THE WAITE GROUP®

C++ PRIMER PLUS®



TEACH YOURSELF OBJECT-ORIENTED PROGRAMMING

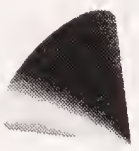
BASED ON AT&T C++ 2.0 AND 2.1

COMPATIBLE WITH ALL C++ COMPILERS

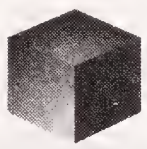
REQUIRES NO KNOWLEDGE OF C

STEPHEN PRATA

WAITE GROUP PRESS™



The Waite Group's®



C++ Primer Plus®

Stephen Prata



TEACH YOURSELF
OBJECT-ORIENTED
PROGRAMMING



WAITE GROUP PRESS ™
A Division of The Waite Group, Inc.
200Tamal Plaza
Corte Madera, California 94925

Development Editor	*Mitchell Waite*
Content Editors	*Mitchell Waite, Harry Henderson*
Technical Reviewers	*David Gerrold, Hank Shiffman, Kent Williams, Harry Henderson*
Design and Production	*Barbara Gelfand*
Cover Illustration	*Steve Epstein*
Text Illustration	*Mary McDermott*
Production Coordination	*Merrill Peterson, Matrix Productions*
Production	*Karyn Kraft, Robin Purcell, Nancy Terzian*
Editorial Director	*Scott Calamar*

© 1991 by The Waite Group, Inc. Published by Waite Group Press, 200 Tamal Plaza, Corte Madera, CA 94925

FIRST EDITION

All rights reserved. No part of this manual shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, desktop publishing, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

All terms mentioned in this book that are known to be trademarks or service marks are listed below. In addition, terms suspected of being trademarks or service marks have been appropriately capitalized. Waite Group Press cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Turbo C++ and Borland C++ are registered trademarks of Borland International.
Sun is a registered trademark of Sun Microsystems.
Zortech C++ is a trademark of Zortech. Inc.
AT&T is a registered trademark of American Telephone and Telegraph.
IBM is a registered trademark of International Business Machines Corp.
MS-DOS is a registered trademark of Microsoft Corporation.
UNIX is a registered trademark of AT&T Bell Laboratories.
Macintosh is a registered trademark of Apple Computer.
The Waite Group is a registered trademark of The Waite Group, Inc.
Primer Plus is a registered trademark of The Waite Group, Inc.

The Waite Group's C++ Primer Plus is distributed to bookstores and book wholesalers by Publishers Group West, Box 8843, Emeryville CA 94662, 800-365-3453 (in California, 415-658-3453).

ISBN: 1-878739-02-6

Printed in the United States of America

92 93 94 • 10 9 8 7 6 5 4 3 2

Library of Congress Cataloging-in-Publication Data

Prata, Stephen.
The Waite Group's C++ primer plus : teach yourself object-oriented programming / Stephen Prata. -- 1st ed.
p. cm.
Includes index.
ISBN 1-878739-02-6 : $26.95
1. C++ (Computer program language) 2. Object-oriented programming. I. Waite Group. II. Title,
III. Title: C++ primer plus. IV. Title: Waite Group's C++ primer plus.
QA76.73.C15P74 1991
005.13'3--dc20 91-28096
CIP

Table of Contents

Preface		xv
Chapter 1	Getting Started	1
Chapter 2	Setting Out to C++	17
Chapter 3	Dealing with Data	51
Chapter 4	Derived Types	95
Chapter 5	Loops and Relational Expressions	153
Chapter 6	Branching Statements and Logical Operators	205
Chapter 7	Functions–C++'s Programming Modules	239
Chapter 8	Adventures in Functions	289
Chapter 9	Objects and Classes	349
Chapter 10	Working with Classes	397
Chapter 11	Class Inheritance	485
Chapter 12	Input, Output, and Files	575
Appendix A	Number Bases	651
Appendix B	C++ Keywords	654
Appendix C	The ASCII Character Set	655
Appendix D	Operator Precedence	660
Appendix E	Other Operators	663
Appendix F	Enumerations	674
Appendix G	Answers to the Review Questions	678
Index		709

Contents

Preface

Chapter 1 Getting Started

Learning C++ 3
A Little History 4
The C Language 5
Object-Oriented Programming 7
C++ 8
Portability and Standards 9
The Mechanics of Creating a Program 11
Creating the Source Code 11
Compilation and Linking 12
Conventions Used in This Book 15
Our System 16

Chapter 2 Setting Out to C++

C++ Initiation 19
The `main()` Function 21
C++ Comments 23
The C++ Preprocessor and the `iostream.h` File 24
C++ Output with `cout` 25
C++ Source-Code Formatting 28
More About C++ Statements 29
Declaration Statements and Variables 30
The Assignment Statement 32
New Trick for `cout` 33
More C++ Statements 34
Using `cin` 35
Function Call Statements 35
More `cout` 36
A Touch of Class 37
Statement Summary 39
Functions 39
Using a Function with a Return Value 39

User-Defined Functions 43
User-Defined Function with a Return Value 47
Summary 49
Review Questions 50

Chapter 3 DEALING WITH DATA
Simple Variables 53
Names for Variables 54
Integer Types 55
The `short`, `int`, and `long` Integer Types 56
Unsigned Types 61
Which Type? 62
Integer Constants 64
The `char` Type: Characters and Small Integers 66
The `const` Qualifier 73
Floating-Point Numbers 75
Writing Floating-Point Numbers 75
Floating-Point Types 76
Floating-Point Constants 79
Floating-Point Advantages and Disadvantages 79
C++ Arithmetic Operators 80
Which Order: Operator Precedence and Associativity 81
Division Diversions 82
The Modulus Operator 84
Type Conversions 86
Summary 91
Review Questions 92

Chapter 4 DERIVED TYPES
Introducing Arrays 97
Strings 101
String Concatenation 104
Using Strings in an Array 104
Adventures in String Input 106
Introducing Structures 113
Other Structure Properties 118
Pointers and the Free Store 120
Declaring and Initializing Pointers 124
Pointers and Numbers 127
Allocating Memory with `new` 128

Chapter 4 (CONTINUED)

Using `new` to Create Dynamic Arrays 130
Pointers, Arrays, and Pointer Arithmetic 133
Pointers and Strings 140
Using `new` to Create Dynamic Structures 143
Freeing Memory with `delete` 146
Automatic Storage, Static Storage, the Free Store 149
Summary 150
Review Questions 151

Chapter 5 LOOPS AND RELATIONAL EXPRESSIONS

Introducing the `for` Loop 155
`for` Loop Parts 157
Back to the `for` Loop 163
Changing the Step Size 165
Inside Strings with the `for` Loop 166
The Increment (++) and Decrement (--) Operators 167
Combination Assignment Operators 169
Compound Statements, or Blocks 170
The Comma Operator (or More Syntax Tricks) 171
Relational Expressions 174
The Mistake You'll Probably Make 176
Comparing Strings 178
The `while` Loop 181
`for` vs. `while` 183
Just a Moment 185
The `do while` Loop 187
Loops and Text Input 189
Using Unadorned `cin` for Input 189
`cin.get()` to the Rescue 191
The End-of-File Condition 193
Yet Another `cin.get()` 195
Nested Loops and Two-Dimensional Arrays 198
Initializing a Two-Dimensional Array 200
Summary 202
Review Questions 203

Chapter 6 BRANCHING STATEMENTS AND LOGICAL OPERATORS
The `if` Statement 207
Formatting Your `if else` Statements 211
The `if else if else` Construction 212
Logical Expressions 214
The Logical `OR` Operator: || 214
The Logical `AND` Operator: && 216
The Logical `NOT` Operator: ! 221
Logical Operator Facts 222
The `ctype.h` Library of Character Functions 224
The `?:` Operator 225
The `switch` Statement 228
`switch` and `if else` 232
The `break` and `continue` Statements 233
Summary 235
Review Questions 236

Chapter 7 FUNCTIONS—C ++'S PROGRAMMING MODULES
Function Review 241
Defining a Function 242
Function Prototyping and Function Calls 245
Function Arguments and Passing by Value 249
Multiple Arguments 250
Another Two-Argument Function 253
Functions and Arrays 256
Arrays and Pointers (Again) 258
Implications of Using Arrays as Arguments 259
More Array Functions 261
Pointers and `const` 267
Functions and Strings 269
Functions That Return Strings 272
Functions and Structures 274
Passing and Returning Structures 274
Another Example 276
Passing Structure Addresses 282
Recursion 284
Summary 286
Review Questions 287

Chapter 8 ADVENTURES IN FUNCTIONS

Inline Functions 291
Default Arguments 295
Function Polymorphism (Function Overloading) 298
An Overloading Example 299
When to Use Function Overloading 302
Reference Variables 302
Creating a Reference Variable 302
References as Function Parameters 306
Reference Properties and Oddities 309
Using References with a Structure 312
Separate Compilation 316
Storage Classes, Scope, and Linkage 320
Scope and Linkage 320
Automatic Variables 321
The Static Storage Class 326
The `static` Modifier (Local Variables) 331
Linkage, External Variables, and the `static` and `external` Modifier 332
Storage Classes and `const` 336
Storage Classes and Functions 337
Storage Classes and Dynamic Allocation 338
Pointers to Functions 339
Function Pointer Basics 339
Summary 344
Review Questions 344

Chapter 9 OBJECTS AND CLASSES

Procedural and Object-Oriented Programming 351
Data Abstraction and Classes 353
What's a Type? 353
The Class 354
Implementing Class Member Functions 358
Using a Class 364
Our Story to Date 368
Class Constructors and Destructors 369
Declaring and Defining a Constructor 370
Using a Constructor 372
Destructors 372
Improving the `softball` Class 373
Further Refinements: The Default Constructor 378

Constructors and Destructors in Review 383
Knowing Your Objects: The `this` Pointer 384
An Array of Objects 391
Class Scope 393
Summary 394
Review Questions 395

Chapter 10 WORKING WITH CLASSES

Friends 401
Designing Friends 403
Operator Overloading 409
A Vector Class with Operator Overloading 410
Overloading Restrictions 426
Overloading the << Operator 427
Improving the Class by Adding a State Member 431
Yet Another Trick 432
Automatic Conversions and Type Casts for Classes 439
Conversion Functions 443
Dynamic Memory and Classes 447
Review Example and `static` Class Members 448
Back to Softball: Improved Memory Management 457
Pointer Problems in Object Initialization and Assignment 464
Fixing the Object Initialization Problem 467
Fixing the Assignment Problem 468
Reviewing Techniques 475
Using Pointers to Objects 477
Summary 481
Review Questions 483

Chapter 11 CLASS INHERITANCE

An Array Base Class 488
Class Definition 489
Using the Class 500
Deriving a Class 502
Defining a Derived Class 503
Implementing the Derived Class 506
Using the Derived Class 511
Managing Your Include Files 515
A Derived Class with Bounds Checking 517
Extending Inheritance 521

Chapter 11 (CONTINUED)

Which Methods? 529
Class Conversions 530
Pointers and References to Objects 533
Virtual Functions 535
The Virtual Solution 543
Classes with Member Classes 547
Member Classes 550
Member Class Initialization for Classes with Object Members 551
Initialization and Assignment in Constructors 552
Completing the Class Definition 553
Disadvantages of Member Classes 556
Multiple Inheritance 557
Multiple Inheritance with a Common Ancestor 563
Virtual Base Classes 564
Using Virtual Base Classes 568
Which Method? 572
Summary 573
Review Questions 573

Chapter 12 INPUT, OUTPUT, AND FILES

An Overview of C++ Input and Output 577
Streams and Buffers 578
Streams, Buffers, and the `iostream.h` File 581
Redirection 583
Output with `cout` 584
The Overloaded << Operator 585
The Other `ostream` Methods 588
Flushing the Output Buffer 591
Formatting with `cout` 592
Input with `cin` 605
How `cin>>` Views Input 608
Other `istream` Class Methods 610
Other `istream` Methods 617
File Input and Output 620
Simple File I/O 621
Opening Multiple Files 623
Stream Checking 626

File Modes 627
Summary 646
Review Questions 647

Appendix A Number Bases 651
Octal Integers 651
Hexadecimal Integers 652
Binary Numbers 652

Appendix B C++ Keywords 654

Appendix C The ASCII Character Set 655

Appendix D Operator Precedence 660

Appendix E Other Operators 663
Bitwise Operators 663
The Shift Operators 663
The Logical Bitwise Operators 665
A Few Common Bitwise Techniques 669
Member Dereferencing Operators 669

Appendix F Enumerations 674
Setting Enumerator Values 676
Enumerations and Classes 677

Appendix G Answers to the Review Questions 678
Chapter 2 678
Chapter 3 679
Chapter 4 680
Chapter 5 682
Chapter 6 683
Chapter 7 685
Chapter 8 687
Chapter 9 689
Chapter 10 694
Chapter 11 702
Chapter 12 704

Index 709

Dear Reader:

Thank you for considering the purchase of *The Waite Group's C++ Primer Plus*. Readers have come to know products from The Waite Group for the care and quality we put in them. Let me tell you a little about our group and how we make our books.

It started in 1976 when I could not find a computer book that really taught me anything. The books that were available talked down to people, lacked illustrations and examples, were poorly laid out, and were written as if you already understood all the terminology. So I set out to write a good book about microcomputers. This was to be a special book—very graphic, with a friendly and casual style, and filled with examples. The result was an instant best-seller.

Over the years, I developed this approach into a "formula" (nothing really secret here, just a lot of hard work—I am a crazy man about technical accuracy and high-quality illustrations). I began to find writers who wanted to write books in this way. This led to co-authoring and then to multiple-author books and many more titles (over seventy titles currently on the market). As The Waite Group author base grew, I trained a group of editors to manage our products. We now have a team devoted to putting together the best possible book package and maintaining the high standard of our existing books.



THE WAITE GROUP

We are most proud of this particular book, *C++ Primer Plus*. The author, Stephen Prata, is also the coauthor of *The Waite Group's New C Primer Plus* (SAMS), a book that is well known for its clarity and accessibility. In fact, *New C Primer Plus* recently received the Computer Press Association's 1990 Best How-To Book award. Our new title on C++ follows in that tradition—you'll find it a good tutorial on the syntax of C++ and at the same time a comfortable journey through the landscape of object-oriented programming. Prata takes the approach that C++ is a stand-alone language, and therefore you don't have to know C to use this book.

We hope you enjoy this and all the other books upcoming from Waite Group Press. If you wish to be on our mailing list, or if you have any comments to tell us after you have purchased and read the book, please fill out and send in the detachable postpaid card.

Mitchell Waite

Mitchell Waite
President

200 Tamal Plaza Corte Madera California 94925 415-924-2515 Fax. 415-924-2576

Preface

When the Waite Group first released *C Primer Plus* in 1984, the C language had been around for about a decade but was just beginning to boom. We take pride in the important role that our book played in introducing programmers to C. Today the C++ language, which derives from C, has reached a similar stage in its evolution. It's booming because it offers a new paradigm—Object-oriented programming, or OOP—well-suited to modern programming needs. Thus AT&T is rewriting UNIX in C++ because C++ improves the reliability, maintainability, and reusability of the code. Apple is using C++ to develop system software for its Macintosh line for the same reasons and because OOP techniques are a natural match to program features such as windows and dialog boxes. Individual programmers are turning to C++ because its new features bring the thrill back to programming. Naturally, it's time to release *C++ Primer Plus* and help this new boom along.

One difference between now and then is that many more books have been written about C++ than were written about C when it was new. However, none of the new C++ books plays the role that a Waite Group "primer" does. Many C++ books assume that you already know C and know it well. That's of little help to those who wish to move to C++ from, say, Pascal or BASIC, or to those who have enjoyed C recreationally without acquiring expert status. Most of the other C++ books present the full language, not just the new elements, but still assume you are fairly knowledgeable in C and in programming in general. Some C++ books make excellent references but can be rather tough sledding for learning the language. A few C++ titles were rushed out the door. They merely tack on a few new chapters to an old C book and don't fully integrate the new material or really do justice to C++'s exciting new object-oriented features.

Enter *The Waite Group's C++ Primer Plus.* We don't assume you know C, and we integrate discussing the basic C language with presenting the C++ features. We do assume you've had some programming experience, but we don't skip over the basics. We've tried to present C++ in a book instilled with traditional Waite Group primer virtues:

- A primer should be an easy-to-use, friendly guide.
- A primer doesn't assume that you already are familiar with all relevant programming concepts.
- A primer emphasizes hands-on learning with brief, easily typed examples that develop your understanding a concept or two at a time.
- A primer clarifies concepts with illustrations.

- A primer provides exercises to let you test your understanding, making the book suitable for self-learning or for the classroom.

C++ Primer Plus presents C++ fundamentals and illustrates them with short, to-the-point programs that are easy to copy and to experiment with. The book is not intended to provide encyclopedic coverage of all features and nuances of the C++ language, but it does present the most important aspects while laying the foundation for further study. You'll learn about input and output, how to make programs perform repetitive tasks and make choices, the many ways to handle data, and how to use functions. You'll learn about the important object-oriented programming concepts of information hiding (lots of fun), polymorphism (not as bad as it sounds), and inheritance. Besides learning basic techniques, you'll learn about the OOP philosophy. Meanwhile, we'll do our best to keep the presentation short, simple, and fun. Our goal is that by the end you'll be able to write solid, effective programs and enjoy yourself doing so.

Acknowledgements

Many people have contributed to this book. In particular, I wish to thank Mitch Waite for his work in developing, shaping, and reshaping this book, and for reviewing the manuscript. I appreciate Harry Henderson's work in reviewing the last few chapters and in testing programs with the Zortech C++ compiler. Thanks to David Gerrold for reviewing the entire manuscript and for championing the needs of less experienced readers. Also, thanks to Hank Shiffman for testing programs using Sun C++ and to Kent Williams for testing programs with AT&T *cfront* and with G++. Thanks to Nan Borreson of Borland International for her responsive and cheerful assistance with Turbo C++ and Borland C++. Thank you, Ruth Myers and Christine Bush, for handling the relentless paper flow involved with this kind of project. Finally, thanks to Scott Calamar for keeping everything on track.

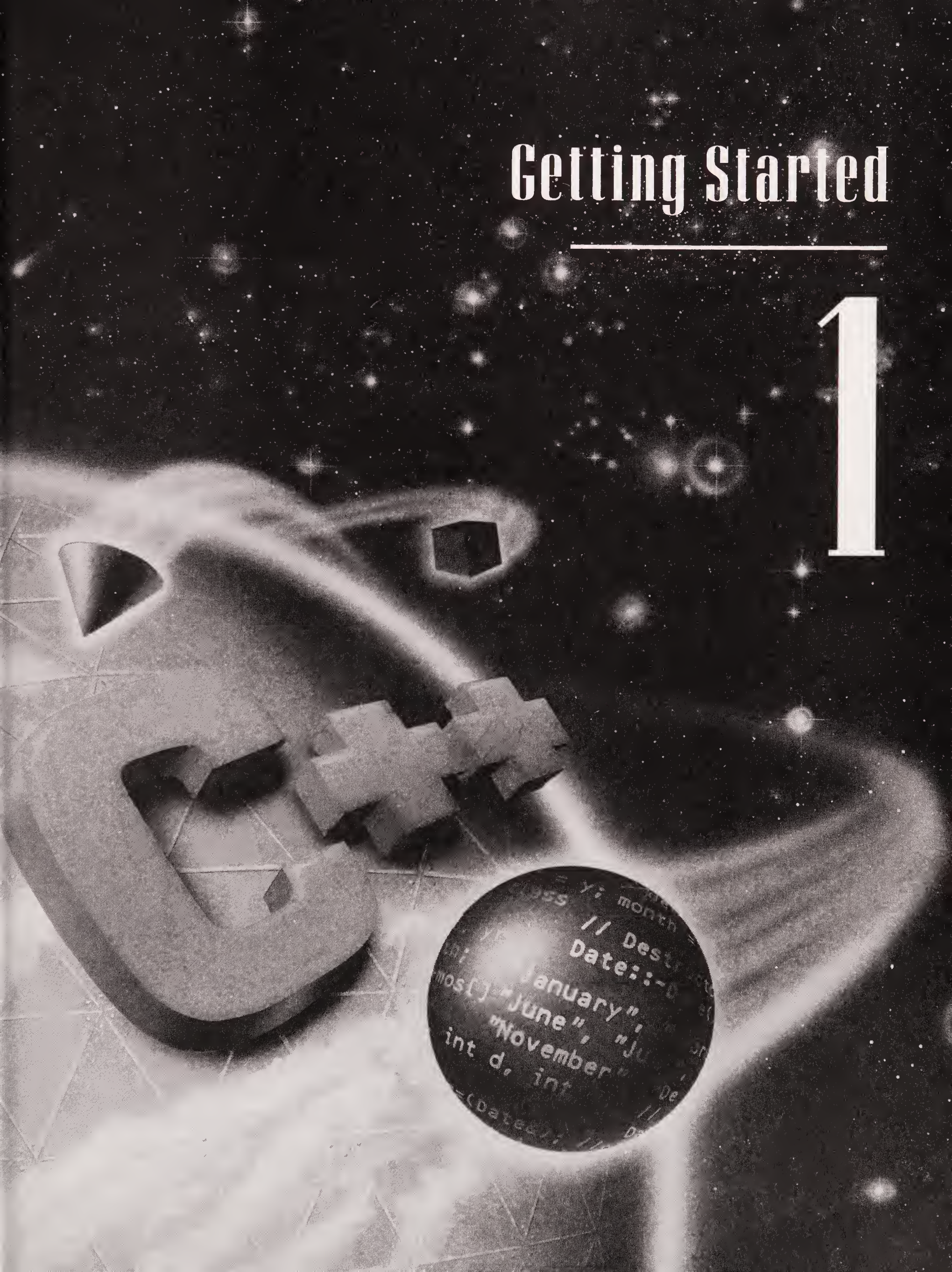
Dedication

To my colleagues at the College of Marin, with whom it is a pleasure to work.

Stephen Prata

Getting Started

1



Welcome to C++! This exciting young language, blending the C language with support for object-oriented programming, seems destined to be one of the most important programming languages of the 1990s. Its C ancestry brings C++ the tradition of an efficient, compact, fast, and portable language. Its object-oriented heritage brings C++ a fresh programming methodology designed to cope with the escalating complexity of modern programming tasks. This dual heritage is both a blessing and a bane. It makes the language very powerful, but it also means there's more to learn. In this chapter we'll explore C++'s background further, then go over some of the ground rules for creating C++ programs. The rest of the book teaches you to use the C++ language, going from the modest basics of the language to the glory of object-oriented programming (OOP) and its supporting cast of new jargon—objects, classes, encapsulation, data hiding, polymorphism, and inheritance. (Of course, as you learn C++, these terms will be transformed from buzz words to the necessary vocabulary of cultivated discourse.)

LEARNING C++

C++ joins together two separate programming traditions—the procedural language tradition, represented by C, and the object-oriented language tradition, represented by the enhancements C++ adds to C. This chapter will look into those traditions shortly. But first, let's consider what this dual heritage implies about learning C++. The main reason to use C++ is to avail yourself of its object-oriented features. But to do so, you need a sound background in standard C, for that language provides the basic types, operators, control structures, and syntax rules. So, if you already know C, you're poised to learn C++. But it's not just a matter of learning a few more keywords and constructs. Going

from C to C++ involves about as much work as learning C in the first place. Also, if you know C, you need to unlearn some programming habits as you make the transition to C++. If you don't know C, you have to master both the C and the OOP components to learn C++, but at least you may not have to unlearn programming habits. You may be gaining the impression that learning C++ may involve some mind-stretching effort on your part, and it will. But this book will guide you through the process in a clear, helpful manner, one step at a time, so the mind-stretching will be sufficiently gentle to avoid leaving you with a sprained brain.

C++ Primer Plus approaches C++ by teaching both its C and its OOP components, so it assumes no prior knowledge of C. You'll start by learning C. Even if you know C, you may find this part of the book a good review. Also, it points out concepts that will become important later, and it indicates where C++ differs from C. Once well-founded in C basics, you'll add the C++ superstructure. At that point you'll learn about objects and classes and how C++ implements them.

This book is not intended to be a complete C++ reference, so it won't explore every nook and cranny of the language. But you will learn all the major features of the language, including some, like multiple inheritance, that are new with Version 2.0 of C++.

Now let's take a short look at some of C++'s background.

A Little History

Computer technology has evolved at an amazing rate during the last few decades. Today a laptop computer can compute faster and store more information than could the mainframe computers of twenty-five years ago. (Quite a few programmers can recall bearing offerings of decks of punched cards to be submitted to a mighty, room-filling computer system having a majestic 100 Kb of memory—not enough memory to run a good personal computer game today.) Computer languages have evolved, too. The changes may not be as dramatic, but they are important. Bigger, more powerful computers spawn bigger, more complex programs which, in turn, raise new problems in program management and maintenance. In the 1970s, languages like C and Pascal helped usher in an era of structured programming, a philosophy that brought some order and discipline to a field badly in need of these qualities. Beside providing the tools for structured programming, C also produced compact, fast-running programs along with the ability to address hardware matters such as managing communication ports and disk drives. These gifts helped make C the dominant programming language in the 1980s. Meanwhile, the 1980s witnessed the growth of a new programming paradigm, object-oriented programming, or OOP, as embodied in languages such as Smalltalk and C++. Let's examine these two developments (C and OOP) a bit more closely.

THE C LANGUAGE

In the early 1970s, Dennis Ritchie of Bell Laboratories was working on a project to develop the UNIX operating system. (An operating system is a set of programs that manages a computer's resources and handles its interactions with users. For example, it's the operating system that puts the system prompt on-screen and that runs programs for you.) For this work Ritchie needed a language that was concise, that produced compact, fast programs, and that could control hardware efficiently. Traditionally, programmers met these needs by using assembly language, which is closely tied to a computer's internal machine language. However, assembly language is a *low-level* language, that is, it is specific to a particular computer processor. So if you want to move a program to a different kind of computer, you may have to completely rewrite the program using a different assembly language. It was a bit as if each time you bought a new car, you found that the designers decided to change where the controls went and what they did, forcing you to relearn how to drive. But UNIX was intended to work on a variety of computer types (or platforms). That suggested using a *high-level* language. A high-level language is oriented towards problem-solving instead of towards specific hardware. Special programs called *compilers* translate a high-level language to the internal language of a particular computer. Thus you can use the same high-level language program on different platforms by using a separate compiler for each platform. Ritchie wanted a language that combined low-level efficiency and hardware access with high-level generality and portability. So, building from older languages, he created C.

C Programming Philosophy

Because C++ grafts a new programming philosophy onto C, we should first take a look at the older philosophy C follows. In general, computer languages deal with two concepts—data and algorithms. The data constitute the information a program uses and processes. The algorithms are the methods the program uses. See Figure 1-1. C, like most mainstream languages to date, is a *procedural* language. That means it emphasizes the algorithm side of programming. Conceptually, procedural programming consists of figuring out the actions a computer should take, then using the programming language to implement those actions. A program prescribes a set of procedures for the computer to follow to produce a particular outcome, much as a recipe prescribes a set of procedures for a cook to follow to produce a cake.

Earlier procedural languages, such as FORTRAN and BASIC, had run into organizational problems as programs grew larger. For example, programs often use branching statements, which route execution to one or another set of instructions depending upon the result of some sort of test. Many older programs had such tangled routing (called "spaghetti programming") that it was virtually impossible to understand

a program by reading it, and modifying such a program was an invitation to disaster. In response, computer scientists developed a more disciplined style of programming loosely called *structured programming*. C includes features to facilitate this approach. For instance, structured programming limits branching (choosing which instruction to do next) to a small set of well-behaved constructions. C incorporates these constructions (the *for* loop, the *while* loop, the *do while* loop, and the *if else* statement) into its vocabulary.

Top-down design was another of the new principles. The idea is to break a large program into smaller, more manageable tasks. If one of these tasks is still too broad, divide it into yet smaller tasks. Continue with this process until the program is compartmentalized into small, easily programmed modules. (Organize your study. Aargh! Well, organize your desk, your table top, your filing cabinet, and your bookshelves. Aargh!

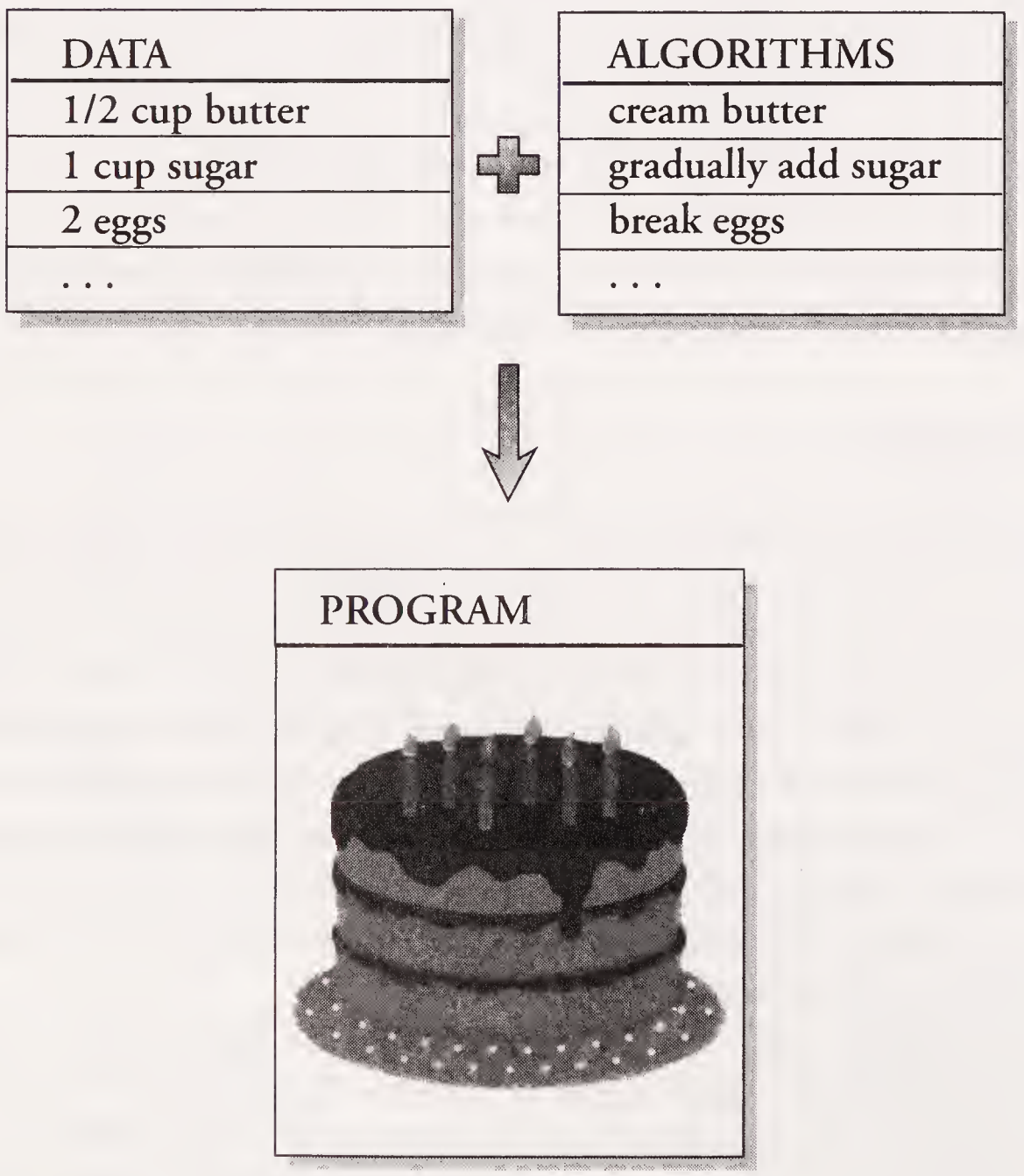


Figure 1-1 *Data + algorithms = program*

Well, start with the desk and organize each drawer, starting with the middle one. Hmmm, perhaps I can manage that task.) C's design facilitates this approach, encouraging you to develop program units called *functions* to represent individual task modules. As you may have noticed, the structured programming techniques reflect a procedural mind-set, thinking of a program in terms of the actions it performs.

OBJECT-ORIENTED PROGRAMMING

Although the principles of structured programming improved the clarity, reliability, and ease of maintenance of programs, large-scale programming still remains a challenge. Object-oriented programming (OOP) brings a new approach to that challenge. Unlike procedural programming, which emphasizes algorithms, OOP emphasizes the data. Rather than trying to fit a problem to the procedural approach of a language, OOP attempts to fit the language to the problem. The idea is to design a data form that corresponds to the essential features of a problem. In C++, a *class* is a specification describing such a new data form, and an *object* is a particular data structure constructed according to that plan. For instance, a class could describe the general properties of a corporation executive (name, title, salary, unusual abilities, etc.), while an object would represent a specific executive (Gilford Sheepblat, vice president, $325,000, knows how to use a CONFIG.SYS file, etc.). In general, a class defines what data is used to represent an object *and* the operations that can be performed upon that data. For example, suppose you were developing a computer drawing program capable of drawing a rectangle. You could define a class to describe a rectangle. The data part of the specification could include such things as the location of the corners, the height and width, the color and style of the boundary line, and the color and pattern used to fill the rectangle. The operations part of the specification could include methods for moving the rectangle, resizing it, rotating it, changing colors and patterns, and copying the rectangle to another location. If you then use your program to draw a rectangle, it will create an object according to the class specification. That object will hold all the data values describing the rectangle, and you can use the class methods to modify that rectangle. If you draw two rectangles, the program will create two objects, one for each rectangle.

The OOP approach to program design is to first design classes that accurately represent those things with which the program deals. A drawing program, for example, might define classes to represent rectangles, lines, circles, brushes, pens, and the like. The class definitions, recall, include a description of permissible operations for each class, such as moving a circle or rotating a line. Then you proceed to design a program using objects of those classes. The process of going from a lower level of organization, such as classes, to a higher level, such as program design, is called *bottom-up* programming.

There's more to OOP programming than the binding of data and methods into a class definition. OOP, for example, facilitates creating reusable code, and that eventu-

ally can save a lot of work. Information hiding safeguards data from improper access. Polymorphism lets you create multiple definitions for operators and functions, with the programming context determining which definition is used. Inheritance lets you derive new classes from old ones. As you can see, object-oriented programming introduces many new ideas and involves a different approach to programming than does procedural programming. Instead of concentrating on tasks, you concentrate on representing concepts. Instead of taking a top-down programming approach, you sometimes take a bottom-up approach. This book will guide you through all these points with plenty of easily grasped examples.

C++

Like C, C++ began its life at Bell Labs, where Bjarne Stroustrup developed the language in the early 1980s. In his own words, "C++ was primarily designed so that the author and his friends would not have to program in assembler, C, or various modern high-level languages. Its main purpose is to make writing good programs easier and more pleasant for the individual programmer."[1] Stroustrup based C++ on C because of C's brevity, its suitability to system programming, its widespread availability, and its close ties to the UNIX operating system. C++'s OOP aspect was inspired by a computer simulation language called Simula67. Stroustrup added OOP features to C without significantly changing the C component. Thus C++ is a superset of C, meaning that any valid C program is a valid C++ program, too. There are some minor discrepancies, but nothing crucial. C++ programs can use existing C software libraries. Libraries are collections of programming modules that you can call up from a program. They provide proven solutions to many common programming problems, thus saving you much time and effort. This has helped the spread of C++.

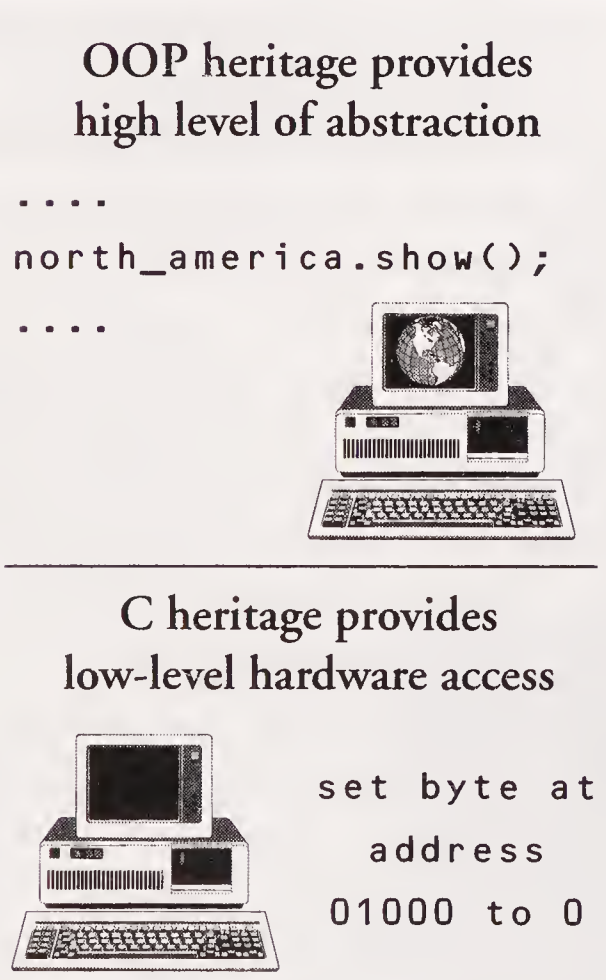


Figure 1-2 *C++ duality*

The name C++ comes from the C increment operator ++, which adds *1* to the value of a variable. The name C++ correctly suggests an augmented version of C.

A computer program translates a real-life problem to a series of actions to be taken by a computer. While the OOP aspect of C++ gives the lan-

[1]Bjarne Stroustrup, *The C++ Programming Language*, Reading, MA: Addison-Wesley Publishing Company, 1986.

guage the ability to relate to concepts involved in the problem, the C part of C++ gives the language the ability to get close to the hardware. See Figure 1-2. This combination of abilities also has helped the spread of C++. It may also involve a mental shift of gears as you turn from one aspect of a program to another. (Indeed, some OOP purists regard adding OOP features to C akin to adding wings to a pig, albeit a lean, efficient pig.) Also, because C++ grafts OOP onto C, you can ignore C++'s object-oriented features. But you'll miss a lot if that's all you do.

Portability and Standards

You've written a handy C++ program for the 286 PC AT computer at work when management decides to replace the machine with a Sun workstation, a computer using a different processor and a different operating system. Can you run your program on the new platform? Of course, you'll have to recompile the program using a C++ compiler designed for the new platform. But will you have to make any changes to the code you wrote? If you can recompile the program without making changes and it runs without a hitch, we say the program is *portable.*

There are a couple of obstacles to portability. One is hardware. A program that is hardware-specific is not likely to be portable. One that takes direct control of an IBM PC VGA video board, for instance, will be speaking gibberish as far as a Sun is concerned. We will avoid that sort of programming in this book. (The Waite Group's *Object-Oriented Programming in Turbo C++* covers more hardware-specific issues.)

The second obstacle to portability is language divergence. Certainly, that can be a problem with spoken languages. A Yorkshireman's description of the day's events may not be portable to Brooklyn, even though English is spoken in both areas. Computer languages, too, can develop dialects. Is the IBM PC C++ implementation the same as the Sun implementation? Although most implementors would like to make their versions of C++ compatible with others, it's difficult to do so without a published standard describing exactly how the language works. Therefore, the American National Standards Institute (ANSI) has created a committee (ANSI X3J16) to develop a standard for C++. (ANSI already has developed a standard for C.) ANSI's decision to create a standard emphasizes that C++ has become an important and widespread language. It also indicates C++ has reached a certain level of maturity, for it's not productive to introduce standards while a language is developing rapidly.

The ANSI C++ standard and its international equivalent, the ISO standard, remain to be developed. But the committee is taking AT&T Version 2.0 of C++ as the base for developing the standard. This book, too, is based on Version 2.0.

The ANSI C++ standard additionally will draw upon the ANSI C standard, since C++ is supposed to be, as far as possible, a superset of C. That means any valid C

program ideally should also be a valid C++ program. There are a few differences between ANSI C and the corresponding rules for Version 2.0 of C++, but they are minor. Indeed, ANSI C incorporates some features first introduced in C++, such as function prototyping and the `const` type qualifier. The ANSI C standard not only defines the C language, it also defines a standard C library that ANSI C implementations must support. C++ also uses that library; we'll refer to it as the *standard C library* or the *standard library.*

Prior to the emergence of ANSI C, the C community followed a *de facto* standard based on the book *The C Programming Language* by Kernigan and Ritchie. This standard often was termed K&R C; with the emergence of ANSI C, the simpler K&R C now sometimes is called classic C.

Before getting to the C++ language proper, let's cover some of the groundwork about creating programs and about using our book.

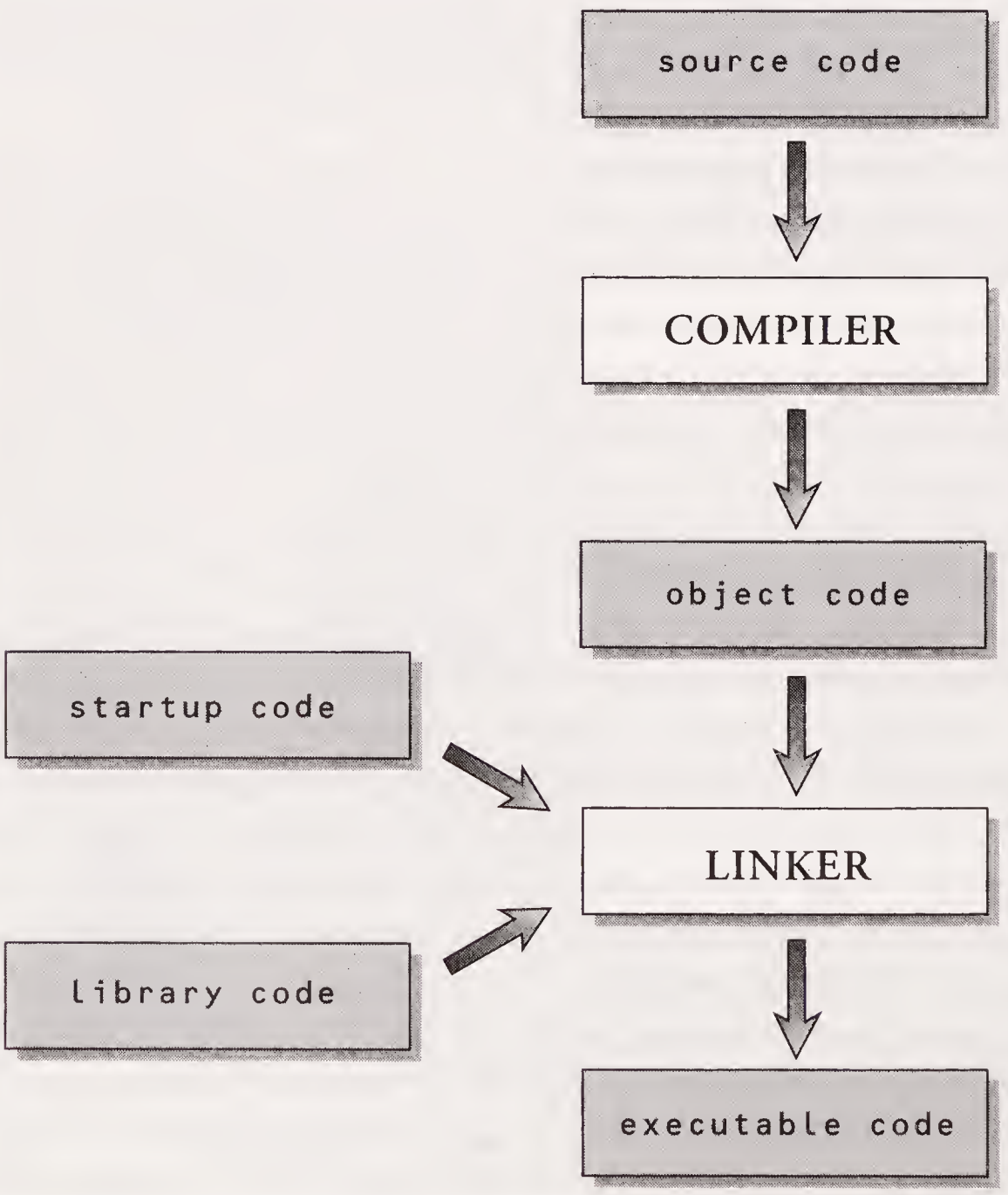


Figure 1-3 *Programming Steps*

The Mechanics of Creating a Program

Suppose you've written a C++ program. How do you get it running? The exact steps depend upon your computer environment and the particular C++ compiler you use, but they will resemble the following steps. (Also see Figure 1-3.)

- Use a text editor of some sort to write the program and save it in a file. This file constitutes the *source* code for your program.
- Compile the source code. This means running a program that translates the source code to the internal language, called *machine language*, used by the host computer. The file containing the translated program is the *object* code for your program.
- Link the object code with additional code. C++ programs, for example, normally use *libraries*. A C++ library contains object code for a collection of computer routines, called *functions*, to perform tasks such as displaying information on the screen or calculating the square root of a number. Linking combines your object code with object code for the functions you use and with some standard start-up code to produce a runnable version of your program. The file containing this final product is called the *executable* code.

We will use the term *source code* throughout the book, so be sure to file that term away in your personal random-access memory.

The programs in this book are generic and should run in any system supporting Version 2.0 of C++, but the steps for putting a program together may differ. Let's look a little further at these steps.

Creating the Source Code

Some C++ implementations, such as Turbo C++ and Zortech C++, provide an integrated environment that lets you manage all steps of program development, including editing, from one master screen. Other implementations, such as AT&T C++ on UNIX, just handle the compilation and linking stages and expect you to type commands on the system command line. In such cases, you can use any available text editor to create and modify source code. On UNIX, for example, you can use *vi* or *ed* or *ex* or *emacs*. On a DOS system, you can use *edlin* or any of several available program editors. You can even use a word processor, providing you save the file as a standard DOS ASCII text file instead of in a special word processor format.

In naming a source file, you need to use the proper suffix to identify the file as being a C++ file. This not only tells you the file is C++ source code, it tells the compiler that, too. (If a UNIX compiler complains to you about a "bad magic number," that's

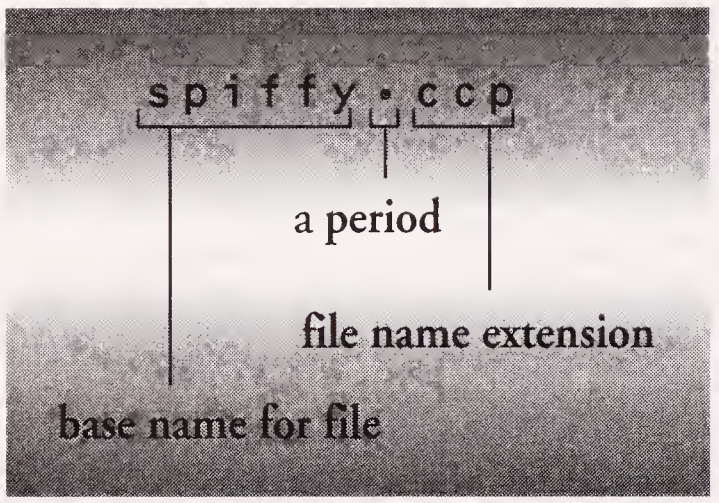


Figure 1-4 *Source file extension*

just its endearingly obscure way of saying that you used the wrong suffix.) The suffix consists of a period followed by a character or group of characters called the *extension*. See Figure 1-4.

The extension you use depends on the C++ implementation. Table 1-1 shows some common choices. For example, `spiffy.C` is a valid AT&T C++ source code file name. Note that UNIX is case-sensitive, meaning you should use an uppercase *C* character. Actually, a lowercase *c* extension also works, but standard C uses that extension. So, to avoid confusion on UNIX systems, use `c` with C programs and `C` with C++ programs. If you don't mind typing an extra character or two, you can also use `cc` and `cxx` extensions with some UNIX systems. DOS, being a bit simple-minded compared to UNIX, doesn't distinguish between uppercase and lowercase, so DOS implementations use additional letters, as shown in Table 1-1, to distinguish between C and C++ programs.

Table 1-1 Source Code Extensions

C++ Implementation	Source Code Extension
UNIX AT&T	C, cc, cxx, c
Zortech	cxx
Borland Turbo	cpp

COMPILATION AND LINKING

Originally, Stroustrup implemented C++ with a C++-to-C *translator* program instead of developing a direct C++ compiler. The translator, called *cfront* (for C front end), translated C++ source code to C source code, which then could be compiled by a standard C compiler. This approach simplified introducing C++ to the C community. Other implementors have used this approach to bring C++ to other platforms. As C++ has developed and grown in popularity, more and more implementors have turned to creating true C++ compilers that generate object code directly from C++ source code. This direct approach speeds up the compilation process and emphasizes that C++ is a separate, if similar, language.

Often the distinction between a translator and compiler is nearly invisible to the user. For instance, on a UNIX system the `CC` command may first pass your program to the *cfront* translator, then automatically pass the translator's output on to the C compiler, which is called *cc*. Henceforth, we'll use the term "compiler" to include translate-and-compile combinations. The mechanics of compiling depend upon the implementation, and we'll outline a few common forms now.

UNIX Compiling and Linking

Suppose, for instance, that you are on a UNIX system using the current AT&T Version 2.0 C++. Use the `CC` command to compile your program. The name is in uppercase letters to distinguish it from the standard UNIX C compiler *cc*. The *CC* compiler is a command-line compiler, meaning you type compilation commands on the UNIX command line. For instance, to compile the C++ source code file `spiffy.C`, you would type this command at the UNIX prompt:

```
CC spiffy.C
```

If, through skill, dedication, or luck, your program has no errors, the compiler generates an object code file with an `o` extension. In this case, the compiler would produce a `spiffy.o` file. Next, the compiler automatically passes the object code file to the system linker, a program that combines your code with library code to produce the executable file. By default, the executable file is called `a.out`. If you used just one source file, the linker also deletes the `spiffy.o` file, for it's no longer needed. To run the program, just type the name of the executable file:

```
a.out
```

Note that if you compile a new program, the new `a.out` executable file replaces the previous `a.out`. (That's because executable files take a lot of space, so overwriting old executable files helps reduce storage demands.) But if you develop an executable program you want to keep, just use the UNIX `mv` command to change the name of the executable file.

In C++, as in C, you can spread a program over more than one file. (Many of the programs in this book from Chapter 8 on do so.) In that case, you can compile a program by listing all the files on the command line:

```
CC my.C precious.C
```

In this case, the compiler does not delete the object code files. That way, if you just change the `my.C` file, you can recompile the program with this command:

```
CC my.C precious.o
```

This recompiles the `my.C` file and links it with the previously compiled `precious.o` file.

Turbo C++ and Borland C++ Compiling and Linking

In Turbo C++'s and Borland C++'s integrated environment, which includes a built-in editor, you use a menu bar, accessible through a mouse or through Alt-key combinations, to make your desires known. For example, the File menu lets you create, save, and open files. The Edit menu assists in editing your source file. The Compile menu offers several compiling options, and the Run menu presents choices for running the program. Once you've used the built-in editor to write a program, the simplest choice is to select Run from the Run menu. This causes Borland C++ or Turbo C++ to compile, link, and run your program. If the compiler catchers errors, of course, it won't run the program, but it will display a list of errors and highlight the offending lines in your source code. Also, the integrated environment includes a debugger that lets you step through the program a line at a time and examine any values you wish to see.

If you develop a program using more than one source code file, use the Project menu to open a new project. Then you can use further menu options to add the relevant files to a project list. The project file lets Borland C++ or Turbo C++ keep track of what's going on. If you change one of the files in the project list, then the compiler knows it should update the executable program. In Borland C++ or Turbo C++ you can have several source code files each in its own window on-screen simultaneously and easily switch from one file to another.

Both Borland C++ and Turbo C++ come with a tutorial program that shows you the ins and outs of the Borland C++ or Turbo C++ environment. And, of course, you can read the manuals.

Zortech C++ Compiling and Linking

Like Borland's Turbo C++, Zortech C++ compiles your C++ source program directly to object code rather than using an intermediate C source file. Starting with Version 2.1, Zortech C++ includes a complete integrated environment, called Zortech Work Bench (ZWB). Before running ZWB, run the memory-resident on-line help program *Ztchelp.exe*. This provides context-sensitive help while you are working with ZWB.

A command-line compiler is also provided, for use with programs that require a large amount of memory. Zortech C++ 2.1 can also be used to create C++ programs that use extended memory, although this requires buying a separate product, the Rational Systems DOS Extender.

ZWB features menu bars, multiple movable and resizable windows, and full mouse support. The menus include File, Edit, Search, Window, Compile, Options, Browse, and Help. ZWB is particularly useful in supporting multiple windows, allowing easy searching of your program and include files, and in using "browsers" to review error messages, source files, and program functions. An enhanced Make facility corresponds to Turbo C++'s Project facility.

To develop a program in Zortech C++, you use the Edit functions to create your source file. You can then compile and run the program in one step by choosing Run Program from the Compile menu, or you can choose Compile to compile the program for later use. When the inevitable errors occur, you can use one of Zortech's four different debuggers: a standard debugger, a remote debugger (which you run on another PC connected by a serial link), a debugger that can reside in extended memory, and a "virtual debugger" that can provide more protection against "crashes" on 80386-based systems. These debuggers provide a wealth of information about what is going on inside your program's classes and functions.

Conventions Used in This Book

To help distinguish between different kinds of text, we've used a few typographic conventions. Italics can denote any of the following:

- Names or values used in a program, such as *x*, *starship*, and *3.14*
- Important words or phrases used for the first time, such as *structured programming*

Monospace "computer" font identifies:

- Language keywords, such as `int` and `if else`
- File names, such as `iostream.h`
- Functions, such as `main()` and `puts()`

C++ source code is presented as follows:

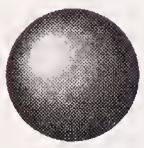
```
#include <iostream.h>
int main(void)
{
    cout << "What's up, Doc!\n";
}
```

Sample program runs use the same format, except user input appears against a screened background:

```
Please enter your name:
Plato
```

Since this book is about object-oriented programming, we've used geometric objects along the way to help you identify various elements of the book. All program listings are indicated by a cone. Rules and suggestions are spheres. And notes are marked with a cube.

You'll find an occasional rule or suggestion in the following format:



You learn by doing, so try the examples and experiment with them.

By the way, you've just read a real and important suggestion, not just an example of what a rule or suggestion looks like.

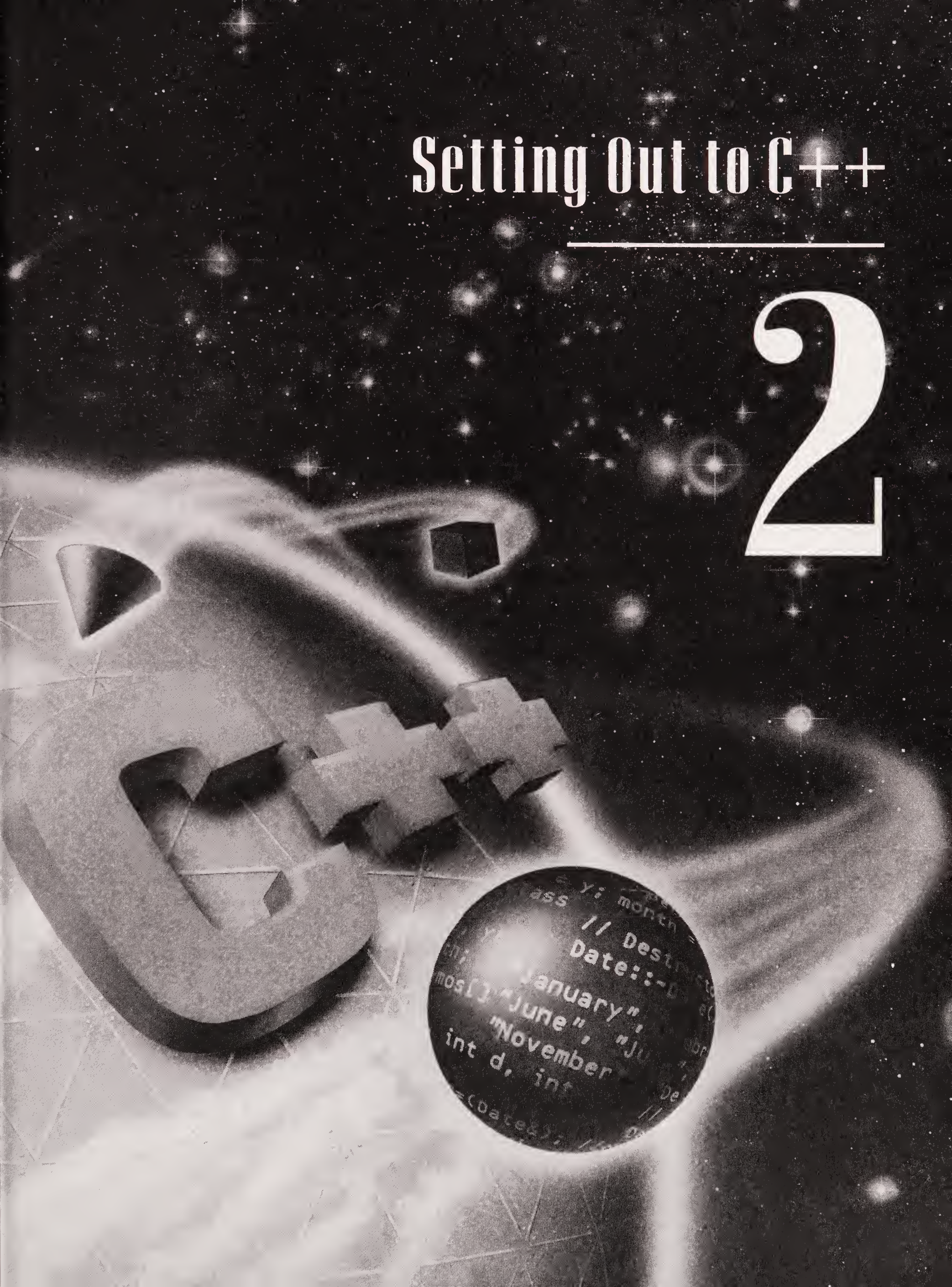
Finally, when you enter program input, you normally have to press the Return key or the Enter key to send the input to the program; some keyboards use one and some the other. This book uses (RETURN) key to mean either the Return key or the Enter key.

Our System

This book describes Version 2.0 of C++, so its examples should work with any C++ implementation compatible with Version 2.0.C++, and with ANSI C. (At least, this is the vision and hope of portability.) However, Version 2.0 is still new, and you may find a few discrepancies. For example, at the time of this writing, some C++ compilers, including Zortech C++, implement the Version 2.0 language features but not its new input and output classes. Systems that use the Version 2.0 (or later) *cfront* translator may then pass the translated code to a C compiler that is not fully ANSI compatible, resulting in some language features being left unimplemented and in some standard ANSI library functions and header files not being supported. Also, some things, such as the number of bytes used to hold an integer, are implementation-dependent. For the record, the examples in this book were developed using Borland's Turbo C++ 2.0 on an IBM-compatible 386 PC with a hard disk and running under MS DOS 3.3. Programs were checked using Zortech's C++ compiler on a similar machine, Sun 2.0 and 2.1 C++ on a Sun computer, the GNU G++ compiler for C++ and AT&T C++ 2.0 and AT&T Cfront 2.01 running UMaXC 4.3 Unix. We'll mention any differences we've found.

Setting Out to C++

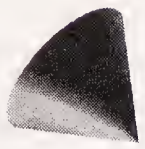
2



When constructing a simple home, you start with the foundation and the framework. If you don't have a solid structure to begin with, you'll have trouble later filling in the details, such as windows, door frames, observatory domes, and parquet ballrooms. Similarly, when you learn a computer language, you should start with learning the basic structure for a program. Only then can you move on to the details, such as loops and objects. This chapter, then, will serve to bring you an overview of the essential structure of a C++ program. While doing so, it will preview some topics—notably functions and classes—that we'll cover in much fuller detail later in the book. (The idea is to introduce at least some of the basic concepts gradually en route to the great awakenings that come later.)

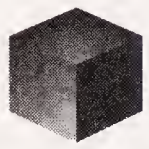
C++ INITIATION

Let's begin with a simple C++ program that displays a message. Listing 2-1 uses the C++ `cout` (pronounced *cee-out*) facility to produce character output. The source code includes several comments to the reader; these lines begin with `//`, and the compiler ignores them. C++ is case-sensitive, that is, it discriminates between uppercase characters and lowercase characters. This means that you must be careful to use the same case as in the examples. For instance, our program uses `cout`. If you substitute `Cout` or `COUT`, the compiler will reject your offering, accusing you of using unknown identifiers. (The compiler also is spelling-sensitive, so don't try `kout` or `coot`, either.) Remember, the `cpp` filename extension is for Turbo C; you may have to use a different extension, as described in Chapter 1.



LISTING 2-1 myfirst.cpp

```
// myfirst.cpp--displays a message
#include <iostream.h>                          // a PREPROCESSOR directive
int main(void)                                 // heading summarizes
                                               // function
{                                              // start of function body
    cout << "Come up and C++ me some time.";   // message
    cout << "\n";                             // start a new line
    return 0;                                  // terminate main()
}                                              // end of function body
```

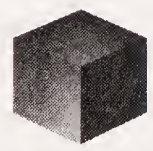


COMPATIBILITY NOTE

Some C++ implementations, including, at the time of this writing, Gnu G++ and Zortech C++, haven't changed over to the *iostream.h* package. If you're using such a compiler, use `#include <stream.h>` instead of `#include <iostream.h>` in this and subsequent programs. Usually, that will be the only change you need to make.

Once you've used your editor of choice to copy this program, you then use your C++ compiler to create the executable code, as Chapter 1 outlined. Here's the output from running the compiled program:

```
Come up and C++ me some time.
```



C INPUT AND OUTPUT

If you're used to programming in C, seeing `cout` instead of the `printf()` function may come as a minor shock. C++ can, in fact, use `printf()`, `scanf()`, and all the other standard C input and output functions, providing you include the usual C `stdio.h` file. But this is a C++ book, so we'll use C++'s new input facilities, which improve in many ways upon the C versions.

You construct C++ programs from building blocks called *functions*. Typically, you organize a program into major tasks, then design separate functions to handle those tasks. The first example is simple enough, as a first example should be, to consist of a single function named `main()`. The `myfirst.cpp` example has the following elements:

- Comments, indicated by the // prefix
- A preprocessor `#include` directive
- A function heading: `int main(void)`

- A function body, delimited by `{` and `}`
- A statement using the C++ `cout` facility to display a message
- A `return` statement to terminate the `main()` function

Let's look at these various elements in more detail now.

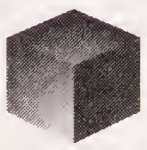
THE MAIN() FUNCTION

Stripped of the trimmings, the example program has the following fundamental structure:

```
int main(void)
{
    statements
    return 0;
}
```

These lines state that we have a function called `main()` and describe how the function behaves. Together they constitute a *function definition.* This definition has two parts: the first line, `int main(void)`, is called the *function heading,* and the portion enclosed in braces (`{` and `}`) is the *function body.* The function heading is a capsule summary of the function's interface with the rest of the program, and the function body represents your instructions to the computer about what the function should do. In C++ each complete instruction is called a s*tatement,* and each statement must be terminated by a semicolon, so don't omit the semicolons when you type the examples.

The final statement in `main()`, called a `return` statement, terminates the function. You'll learn more about the `return` statement later.



STATEMENTS AND SEMICOLONS

A statement represents a complete instruction to a computer. To understand your source code, a compiler needs to know when one statement ends and another begins. Some languages use a statement separator. FORTRAN, for instance, uses the end of the line to separate one statement from the next. Pascal uses a semicolon to separate one statement from the next. In Pascal, you can omit the semicolon in certain cases, such as after a statement just before an END, when you aren't actually separating two statements. (Pragmatists and minimalists will disagree about whether *can* implies *should.*) But C++, like C, uses a *terminator* rather than a separator. Therefore, the semicolon that marks the end of the statement is part of the statement rather than a marker between statements. (The humble semicolon demonstrates that you don't have to look

like Arnold Schwarzenegger to be a terminator.) The practical upshot is that in C++ you can never omit the semicolon.

The Function Heading as Interface

Right now the main point to remember is that C++ syntax requires that you begin the definition of the `main()` function with this heading: `int main(void)` . This chapter will go into details later, but, for those who can't put their curiosity on hold, here's a preview.

In general, a C++ function is activated, or *called*, by another function, and the function heading describes the interface between a function and the function that calls it. The part preceding the function name is called the function *return type*, and it describes information flow from a function to the function that called it. The part within the parentheses following the function name is called the *argument list*, and it describes information flow from the calling function to the function being defined. This general format is a bit confusing when applied to `main()` because `main()` normally isn't called by another function. But you can think of `main()` as being called by the operating system (UNIX, DOS, or whatever) when you start the program. So in the case of `main()`, the function header describes the interface between `main()` and the operating system.

Let's look at the interface for `main()`, beginning with the `int` part. A C++ function called by another function can return a value to the activating (calling) function. That value is called a *return* value. In this case, `main()` can return an integer value, as indicated by the keyword `int`. Next, consider the word `void`. In general, one C++ function can pass information to another function when it calls that function. The portion of the function heading enclosed in parentheses describes that information. In this case, `void` means that the `main()` function takes no information, or, in the usual terminology, `main()` takes no arguments. (Saying `main()` takes no arguments doesn't mean that `main()` is an unreasonable, authoritarian function. Rather, *argument* is the term computer buffs use to refer to information passed from one function to another.)

In short, the heading

```
int main(void)
```

states that the `main()` function can return an integer value *to* the function that calls it and that `main()` takes no information *from* the function that calls it.

Why main() by Any Other Name Is Not the Same

There's an extremely compelling reason to name the function in the `myfirst.cpp` program `main()`: you have to. Every C++ program has to have a function called `main()`. (And not, by the way, `Main()` or `MAIN()` or `mane()`. Remember, case and spelling count.) Since the `myfirst.cpp` program has only one function, that function

must bear the responsibility of being `main()`. When you run a C++ program, execution always begins at the beginning of the `main()` function. Therefore, if you don't have `main()`, you don't have a complete program, and the compiler will point out that you haven't defined a `main()` function.

C++ COMMENTS

The double slash (//) introduces a C++ comment. A comment is a remark from the programmer to the reader, and usually it identifies a section of a program or explains some aspect of the code. The compiler ignores comments. After all, it knows C++ at least as well as you do, and, in any case, it's incapable of understanding comments. So, as far as the compiler is concerned, Listing 2.1 looks comment-free like the following:

```
#include <iostream.h>
int main(void)
{
    cout << "Come up and C++ me some time.";
    cout << "\n";
    return 0;
}
```

C++ comments run from the // to the end of the line. A comment can be on its own line or it can be on the same line as code. Incidentally, note the first line in Listing 2-1:

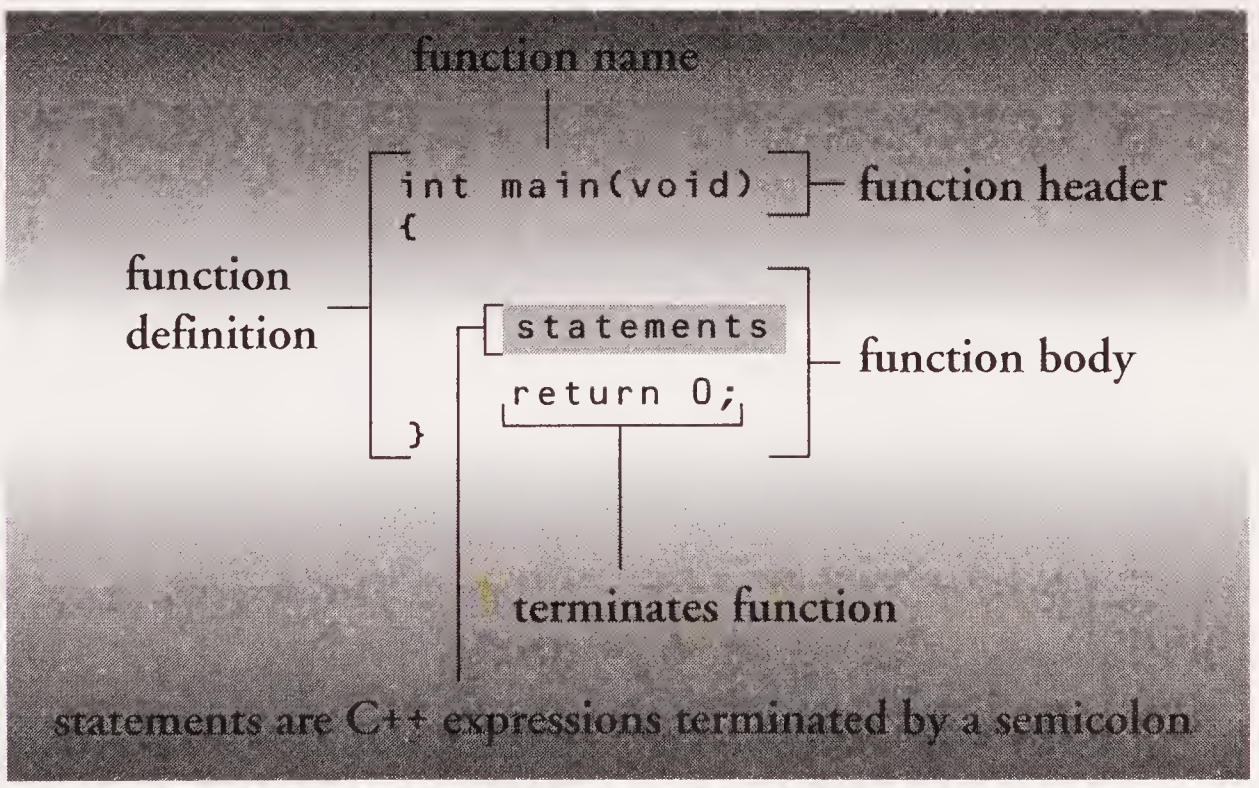
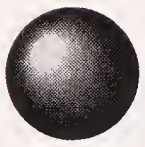


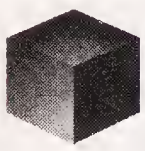
Figure 2-1 *The* `main()` *function*

```
// myfirst.cpp -- displays a message
```

In this book we'll start all our programs with a comment giving the file name we're using for the source code and also giving a brief program summary. As we mentioned in Chapter 1, the file name extension for source code depends on your C++ system. Some systems would use `myfirst.C` or `myfirst.cxx` for names.



You should use comments to document your programs. The more complex the program, the more valuable comments become. Not only do they help others to understand what you have done, they help you understand what you've done, especially if you haven't looked at the program for a while.



C-STYLE COMMENTS

C++ also recognizes C comments, which are enclosed between /* and */ symbols:

```
#include <iostream.h> /* a C-style comment */
```

Because the C-style comment is terminated by */ rather than by the end of a line, it can be spread over more than one line. You can use either or both styles in your programs. We'll stick to the C++ style. That way, a C programmer glancing over your shoulder will know you've advanced to a higher level of programming.

THE C++ PREPROCESSOR AND THE IOSTREAM.H FILE

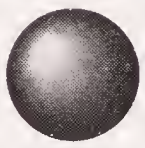
C++, like C, uses a preprocessor. This is a program that processes a source file before the main compilation takes place. (Some C++ implementations, you may recall from Chapter 1, use a translator program to convert a C++ program to C. The translator also is a form of preprocessor, but we're not talking about *that* preprocessor; we're talking about the one that handles directives whose names begin with #.) You don't have to do anything special to invoke this preprocessor. It operates automatically when you compile the program.

Our listing uses the *#include* feature:

```
#include <iostream.h>    // a PREPROCESSOR directive
```

This directive causes the preprocessor to add the contents of the `iostream.h` file to your program. This is a typical preprocessor action—adding or replacing text in the source code before it's compiled.

This raises the question, why add the contents of the `iostream.h` file to the program? The answer concerns communication between the program and the outside world. The *io* in `iostream.h` refers to *input*, which is information brought into the program, and to *output*, which is information sent out from the program. C++'s input/output scheme involves several definitions found in the `iostream.h` file. The program needs these definitions to use the `cout` facility. The `#include` directive causes the contents of the `iostream.h` file to be sent along with the contents of your file to the compiler. In essence, the `#include <iostream.h>` line in the program is replaced by the contents of the `iostream.h` file. Your original file is not altered, but a composite file formed from your file and `iostream.h` goes on to the next stage of compilation.



Programs that use `cin` and `cout` for input and output must include the `iostream.h` file.

Prior to C++ 2.0, C++ programs used a header file called `stream.h` instead of `iostream.h`. The `iostream.h` file reflects improvements in C++ input/output handling. This books sticks to the 2.0 version, so if you use an older implementation of I/O, you may have to make some adjustments to the examples.

C++ OUTPUT WITH COUT

Now let's look at how you display a message. The *myfirst.cpp* program used the following C++ statement:

```
cout << "Come up and C++ me some time.";
```

The part enclosed within the double quotation marks is the message to print. In C++, any series of characters enclosed in double quotation marks is called a *string*, presumably because it consists of several characters strung together into a larger unit. The `<<` notation indicates that the statement is sending this string to `cout`; the symbols point the way the information flows. And what is `cout`? It's a predefined object that knows how to display a variety of things, including strings, numbers, and individual characters. (An object, as you may remember from Chapter 1, is the embodiment of a class definition that describes how data is stored and used.)

Well, this is a bit awkward. We won't be in a position to explain objects for several more chapters, yet here we have to use it. Actually, this reveals one of the strengths of objects. You don't have to know the innards of an object in order to use it. All you need to know is its interface, that is, how it's used. The `cout` object has a simple interface. If *string* represents a string, then do the following to display the string:

```
cout << string;
```

This is all you need to know to display a string, but let's see how the C++ conceptual view represents the process. In this view, the output is a *stream*, that is, a series of characters flowing from the program—a rather bucolic image. The `cout` object, whose properties are defined in the `iostream.h` file, represents that stream. The object properties for `cout` include an *insertion operator* (`<<`) that inserts the information on its right into the stream. So the statement (note the terminating semicolon)

```
cout << "Come up and C++ me some time.";
```

inserts the string "*Come up and C++ me some time.*" into the output stream. Thus, instead of saying that our program displays a message, we can say that it inserts a string into the output stream. Somehow, that sounds more impressive. See Figure 2-2.

If you're coming to C++ from C, you probably noticed that the insertion operator (`<<`) looks just like the bitwise left-shift operator (`<<`). This is an example of *operator overloading*, by which the same operator symbol can have different meanings. Your program uses the context to figure out which meaning is intended. C itself has some operator overloading. For instance, the `&` symbol represents both the address operator and the bitwise `AND` operator. And the `*` symbol represents both multiplication and

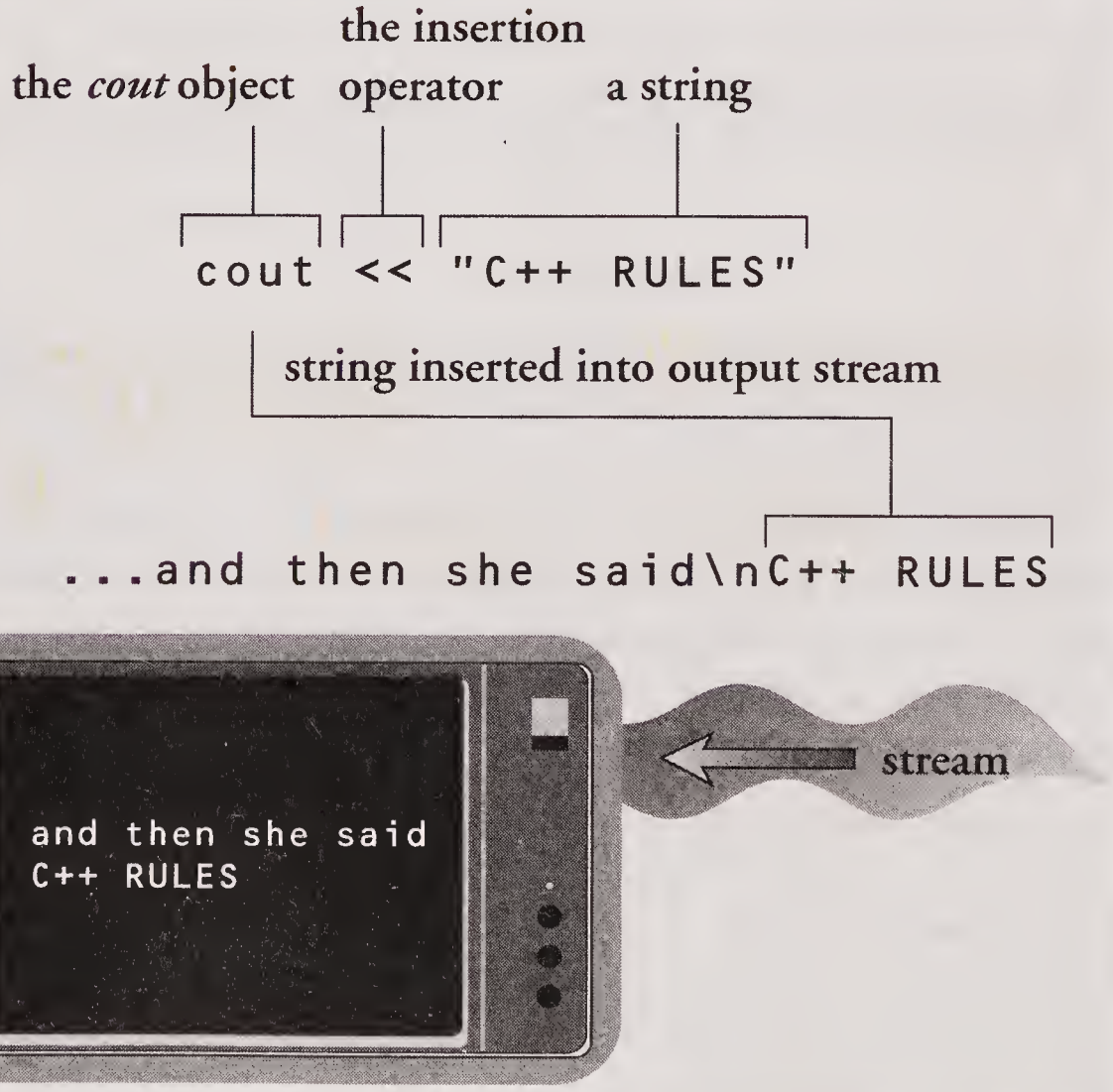


Figure 2-2 *Displaying a string with* `cout`



dereferencing a pointer. The important point here is not the exact function of these operators but that the same symbol can have more than one meaning, with the compiler determining the proper meaning from the context. (You do much the same when you determine the meaning of "shoot" in "shoot the breeze" versus "shoot the piano player.") C++ extends the operator overloading concept by letting you redefine operator meanings for the user-defined types called classes.

The Newline Character (\n)

Now let's examine an odd-looking notation that appeared in the second output statement:

```
cout << "\n";
```

The `\n` is a special C++ (and C) notation representing an important concept dubbed the *newline character*. Although you type the newline character by using two characters (a \ and an *n*), it counts as a single character. Note that you use the backslash (\), not the regular slash (/). Displaying a newline character moves the screen cursor to the beginning of the next line, and sending the newline character to a printer moves the print head to the beginning of the next line. So the newline character earns its name.

Note that the `cout` facility does not move to the next line automatically when it prints a string, so the first `cout` statement in Listing 2-1 leaves the cursor positioned just after the period at the end of the output string. To move the cursor to the beginning of the next line, you must send a newline character to the output. Or, practicing C++ lingo, you must insert a newline character into the output stream.

You can use the newline character just like any ordinary character. Listing 2-1 used it in a separate string, but the listing could have used it in the original string. That is, you can replace the original two output statements with the following:

```
cout << "Come up and C++ me some time.\n";
```

You can even place the newline character in the midst of a string. For instance, consider the following statement:

```
cout << "I am a mighty stream\nof lucid\nclarity.\n";
```

Each newline character moves the cursor to the beginning of the next line, making the output as follows:

```
I am a mighty stream
of lucid
clarity.
```

By leaving out newlines, you can make successive `cout` statements print on the same line. For instance, the statements

```
cout << "The Good, the";
cout << "Bad, ";
cout << "and the Ukelele\n";
```

produce the following output:

```
The Good, theBad, and the Ukelele
```

Note that the beginning of one string comes immediately after the end of the preceding string. If you want a space where two strings join, you have to include it in one of the strings. (Keep in mind that to try out these output examples, you have to place them in a complete program, with a `main()` function heading and opening and closing braces.)

C++ SOURCE-CODE FORMATTING

Some languages, such as FORTRAN, are line-oriented, with one statement to a line. For these languages the carriage return serves to separate statements. But in C++, the semicolon marks the end of each statement. This leaves C++ free to treat the carriage return the same as a space or a tab. That is, in C++, you usually can use a space where you would use a carriage return, and vice versa. This means you can spread a single statement over several lines or place several statements on one line. For instance, you could reformat `myfirst.cpp` as follows:

```
#include <iostream.h>
    int
main
(void) { cout
           <<
"Come up and C++ me some time.";
    cout << "\n"; return 0; }
```

This is ugly, but valid, code. You do have to observe some rules. In particular, in C and C++ you can't put a space, tab, or carriage return in the middle of an element such as a name or a string:

```
int ma  in(void) // INVALID -- space in name
re
turn 0;    // INVALID -- carriage return in word
cout << "Behold the Beans
 of Beauty!";    // INVALID -- carriage return in string
```

The indivisible elements in a line of code are called *tokens*. See Figure 2-3. Generally, you must separate one token from the next by a space, tab, or carriage return,

which collectively are termed *white space*. However, C++ recognizes some special tokens, such as parentheses, without the need of separating white space:

```
returnO;              // INVALID, must be return 0;
return(0);            // VALID, white space omitted
return (0);           // VALID, white space used
int main(void)        // VALID, white space omitted
int main ( void )     // ALSO VALID, white space used
```

C++ Source Code Style

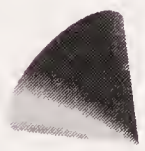
Although C++ gives you much formatting freedom, your programs will be easier to read if you follow some sensible style. You don't want to be nicknamed Old Valid-But-Ugly. Most programmers use the style of Listing 2-1, which observes these rules:

- One statement per line
- An opening and a closing brace for a function each on its own line
- Statements in a function indented in from the braces
- No white space around the parentheses associated with a function name

The first three rules have the simple intent to keep the code clean and readable. The fourth helps one differentiate functions from some built-in C++ structures, like loops, that also use parentheses. We'll mention other rules as we come across them.

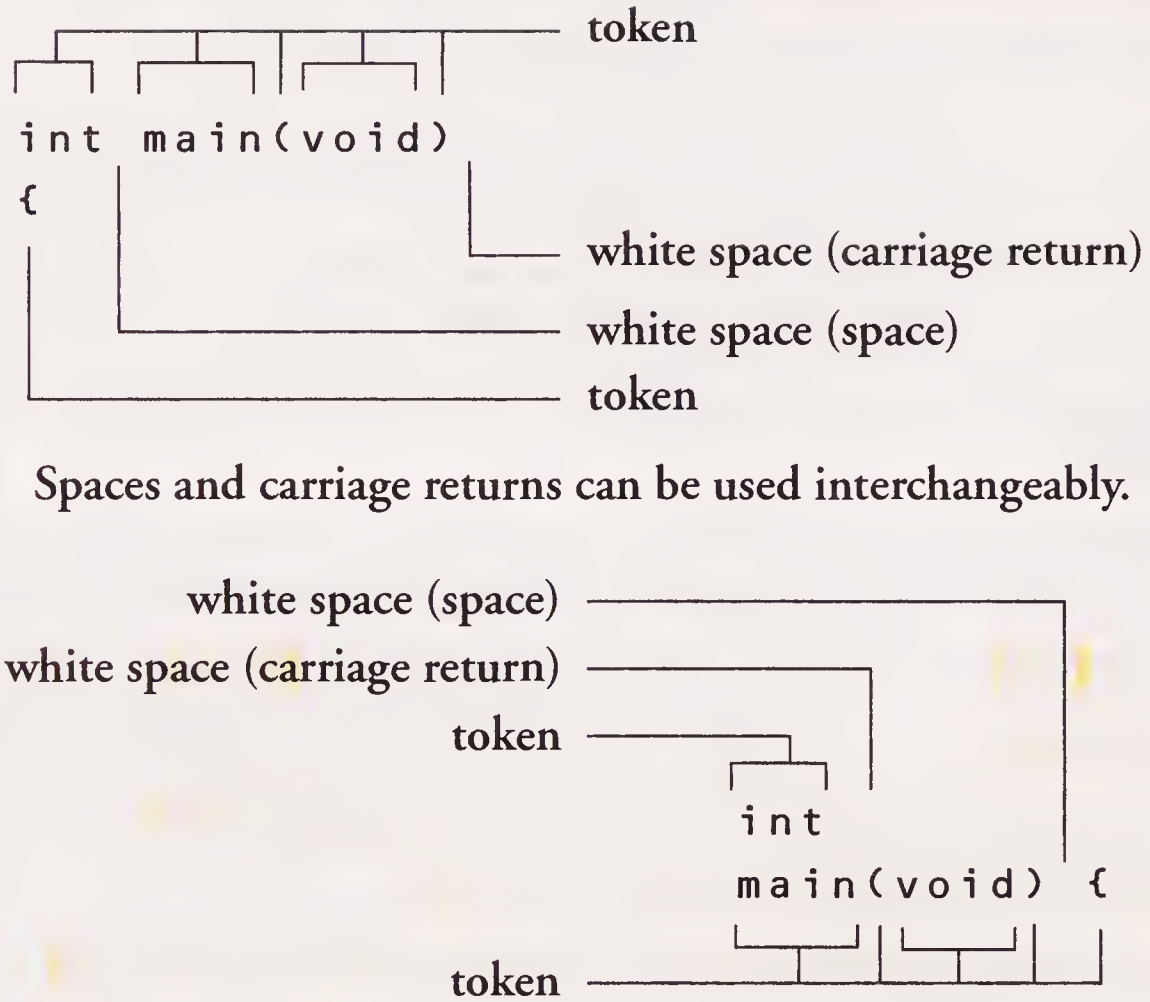
More About C++ Statements

A C++ program is a collection of functions, and each function is a collection of statements. C++ has several kinds of statements, so let's see some of the possibilities. Listing 2-2 provides two new kinds of statements. First, a *declaration statement* creates a *variable*. Second, an *assignment statement* provides a value for that variable. Also, the program shows a new ability for `cout`.



LISTING 2-2 fleas.cpp

```
// fleas.cpp -- display the value of a variable
#include <iostream.h>
int main(void)
{
    int fleas;      // create an integer variable
    fleas = 28;      // give a value to the variable
    cout << "My cat has ";
```



Spaces and carriage returns can be used interchangeably.

Figure 2-3 *Tokens and white space*

```
    cout << fleas;                  // display the value of fleas
    cout << " fleas.\n";
    return 0;
}
```

A blank line separates the declaration from the rest of the program. This practice is the usual C convention, but it's somewhat less common in C++. Here is the program output:

```
My cat has 28 fleas.
```

We will study this program in the next few pages.

DECLARATION STATEMENTS AND VARIABLES

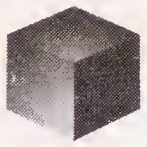
Computers are precise, orderly machines. To store an item of information in a computer, you need to identify how much memory storage space is required for that information, and you need a way to identify the storage location. One relatively painless way to do this in C++ is to use a declaration statement to indicate the type of storage and to

provide a label for the location. For example, our program has this declaration statement (note the semicolon):

```
int fleas;
```

This statement declares that the program will use enough storage to hold what is called an *int*. The compiler takes care of the details of allocating and labeling memory for that task. C++ can handle several kinds, or *types*, of data, and the `int` is perhaps the most basic data type. It corresponds to an *integer*, or whole number. The C `int` type can be positive or negative, but the size range depends on the implementation. Chapter 3 provides the details on `int` and the other basic types.

Besides giving the type, the declaration statement declares that henceforth the program will use the name *fleas* to identify the value stored at that location. We call *fleas* a *variable* because we can change its value. In C++ you must declare all variables. If you were to omit the declaration in `fleas.cpp`, the compiler would report an error when the program attempts to use *fleas* further on. (In fact, you might want to try omitting the declaration just to see how your compiler responds. Then, if you see that response in the future, you'll know to check for omitted declarations.)



WHY MUST VARIABLES BE DECLARED?

Some languages, notably BASIC, create a new variable whenever you use a new name, without the aid of explicit declarations. That may seem friendlier to the user, and it is—in the short run. The problem is that by mistyping the name of a variable, you can inadvertently create a new variable without realizing it. That is, in BASIC, you can do something like the following:

```
CastleDark = 34
...
CastleDank = CastleDank + MoreGhosts
...
PRINT CastleDark
```

Because *CastleDank* is mistyped, the changes made to it leave *CastleDark* unchanged. This kind of error can be hard to trace because it breaks no rules in BASIC. But in C++, the equivalent code breaks the rule about using a variable without declaring it, so the compiler catches the error and stomps the potential bug.

In general, then, a declaration indicates the type of data to be stored and the name we'll use for the data that's stored there. In this particular case, we create a variable called *fleas* in which we can store an integer. See Figure 2-4.

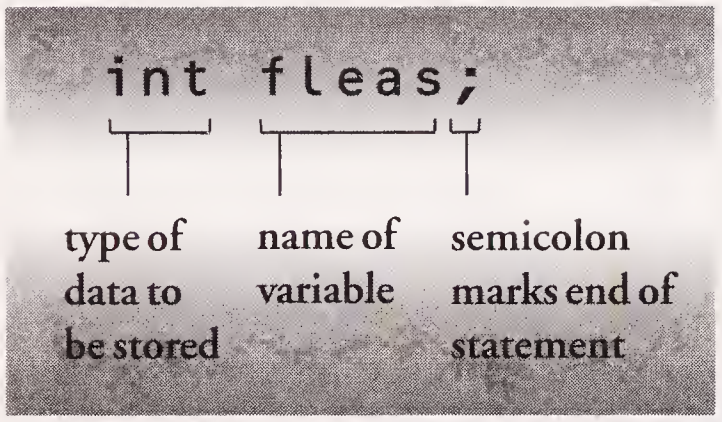


Figure 2-4 *A variable declaration*

The declaration statement in our program is called a *defining* declaration statement, or *definition*, for short. That means its presence causes the compiler to allocate memory space for the variable. In more complex situations, you can also have *reference* declarations. These tell the computer to use a variable that has already been defined elsewhere. So, in general, a declaration need not be a definition, but in our example, it is.

If you're familiar with C or Pascal, you're already familiar with variable declarations. You also may have a modest surprise in store for you. In C and Pascal, all variable declarations normally come at the very beginning of a function or procedure. But C++ has no such restriction. Indeed, the usual C++ style is to declare a variable just before it is first used. That way, you don't have to rummage back through a program to see what the type is. You'll see an example later in this chapter. This style does have the disadvantage of not gathering all your variable names in one place; thus you can't tell at a glance what variables a function uses.



The C++ style for declaring variables is to declare a variable as close as possible to its first use.

THE ASSIGNMENT STATEMENT

An assignment statement assigns a value to a storage location. For instance, the statement

```
fleas = 28;
```

assigns the integer 28 to the location represented by the variable *fleas*. The = symbol is called the *assignment* operator. One unusual feature of C++ (and C) is that you can use the assignment operator serially. That is, the following is valid code:

```
int steinway;
int baldwin;
int yamaha;
yamaha = baldwin = steinway = 88;
```

The assignment works from right to left. First, *88* is assigned to *steinway*, then the value of *steinway*, which is now *88*, is assigned to *baldwin*, then *baldwin*'s value of *88* is assigned to *yamaha*. (C++ follows C's penchant for allowing weird-appearing code.)

NEW TRICK FOR COUT

Prior to now we've given `cout` strings to print. Listing 2-2 additionally gives `cout` a variable whose value is an integer:

```
cout << fleas;
```

The program doesn't print the word *fleas*; instead, it prints the integer value stored in *fleas*, which is *28*. Actually, this is two tricks in one. First, `cout` has to replace `fleas` with its current numeric value of 28. Second, it has to translate the value to the proper output characters.

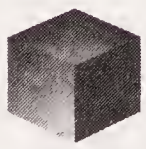
As you see, `cout` works both with strings and with integers. This may not seem particularly remarkable to you, but keep in mind that the integer *28* is something quite different from the string *"28"*. The string holds the characters with which we write the number; that is, a 2 character and an 8 character. Internally, the program stores the code for the 2 character and the code for the 8 character. To print the string, `cout` simply prints each character in the string. But the integer 28 is stored as a numeric value. Instead of storing each digit separately, the computer stores 28 as a *binary* number. (Appendix A discusses this representation.) The main point here is that `cout` has to translate a number in integer form into a character form before it can print it. Furthermore, `cout` is smart enough to recognize that *fleas* is an integer requiring conversion.

Perhaps the contrast with old C will indicate how clever `cout` is. To print the string *"28"* and the integer *28* in C, you could use C's multipurpose output function `printf()`:

```
printf("Printing a string: %s\n", "28");
printf("Printing an integer: %d\n", 28);
```

Without going into the intricacies of `printf()`, note that you have to use special codes (*%s* and *%d*) to indicate whether you are going to print a string or an integer. And if you tell `printf()` to print a string but give it an integer by mistake, `printf()` is too dumb to notice your mistake. It just goes ahead and displays garbage.

The intelligent way in which `cout` behaves stems from C++'s object-oriented features. In essence, the C++ insertion operator (<<) adjusts its behavior to fit the type of data that follows it. In later chapters, when we take up function overloading and operator overloading, you'll learn how to implement such smart designs yourself.



COUT AND PRINTF()

If you are used to C and `printf()`, you may think `cout` looks odd. You may even prefer to cling to your hard-won mastery of `printf()`. But `cout` is really no stranger in appearance than `printf()` with all its conversion specifications.

More important, `cout` has significant advantages. Its ability to recognize types reflects a more intelligent and foolproof design. Also, it is extensible. That is, you can teach `cout` to recognize and display new data types you develop. And if you relish the fine control `printf()` provides, you can accomplish the same effects with more advanced uses of `cout` (Chapter 12).

More C++ Statements

Let's look at a few more examples of statements. The program in Listing 2-3 expands on the preceding example by allowing you to enter a value while the program is running. To do so, it uses `cin` (pronounced *cee-in*), the input counterpart to `cout`. Also, the first two examples didn't illustrate how a function like `main()` can call another function, so this time we have `main()` call the `puts()` function. This function is from the standard C library, which C++ has inherited, and it displays a string. Finally, the program shows yet another way to use that master of versatility, the `cout` object.

LISTING 2-3 yourcat.cpp

```
// yourcat.cpp -- input and output
#include <iostream.h>
#include <stdio.h>      // used for old C I/O functions
int main(void)
{
      int fleas;

      puts("How many fleas does your cat have?");   // Old C
      cin >> fleas;                                 // C++ input
// next line concatenates output
      cout << "Well, that's " << fleas << " fleas too many.\n";
      return 0;
}
```

Here is a sample output:

```
How many fleas does your cat have?
112
Well, that's 112 fleas too many.
```

The new features in this program are using `cin` to read keyboard input, calling a function `puts()`, and combining three output statements into one. Now let's examine these new features.

USING CIN

As the output demonstrates, the value we typed in at the keyboard (*112*) wound up being assigned to the variable *fleas* . Here is the statement that performed that wonder:

```
cin >> fleas;
```

Looking at this statement, you can practically see information flowing from `cin` into *fleas*. Naturally, there is a slightly more formal description of this process. Just as C++ considers output as a stream of characters flowing out of the program, it considers input as a stream of characters flowing into the program. The `iostream.h` file defines `cin` as an object representing this stream. For output, the `<<` operator inserts characters into the output stream. For input, `cin` uses the `>>` operator to *extract* characters from the input stream. Typically, you would provide a variable to the right of the operator to receive the extracted information. (The symbols `<<` and `>>` were chosen to suggest the direction that information flows.)

Like `cout`, `cin` is a smart object. It converts input, which is just a series of characters typed from the keyboard, into a form acceptable to the variable receiving the information. In this case, we've declared *fleas* to be an integer variable, so the input is converted to the numerical form the computer uses to store integers.

FUNCTION CALL STATEMENTS

The second new feature of the `yourcat.cpp` program was calling a library function. C++ programs often use statements that *call* functions to perform various feats and tasks. Functions come from two sources: you and function libraries. You'll soon see some examples of user-supplied functions. Library functions come in library files that the compiler accesses when compiling a program. The compiler joins (*links*) your code with the relevant library code to produce the final program. The `puts()` function in Listing 2-3 is a standard library function available in both C and C++. The `stdio.h` header file contains information relating to this function, just as `iostream.h` contains information relating to `cout` and `cin`. The `puts()` function displays a string, then moves the cursor to the next line. It's a simple function to use, but it is limited. For example, it doesn't print integers. We used `puts()` because it is a simple example of a function, but we'll stick with *cout* in the future.

Let's look at the mechanics of calling a function. Listing 2-3 calls the `puts()` function:

```
puts("How many fleas does your cat have?");
```

Here's what happens. When the program begins execution, the computer goes along, happily executing the statements in `main()`. When it reaches the function call, the

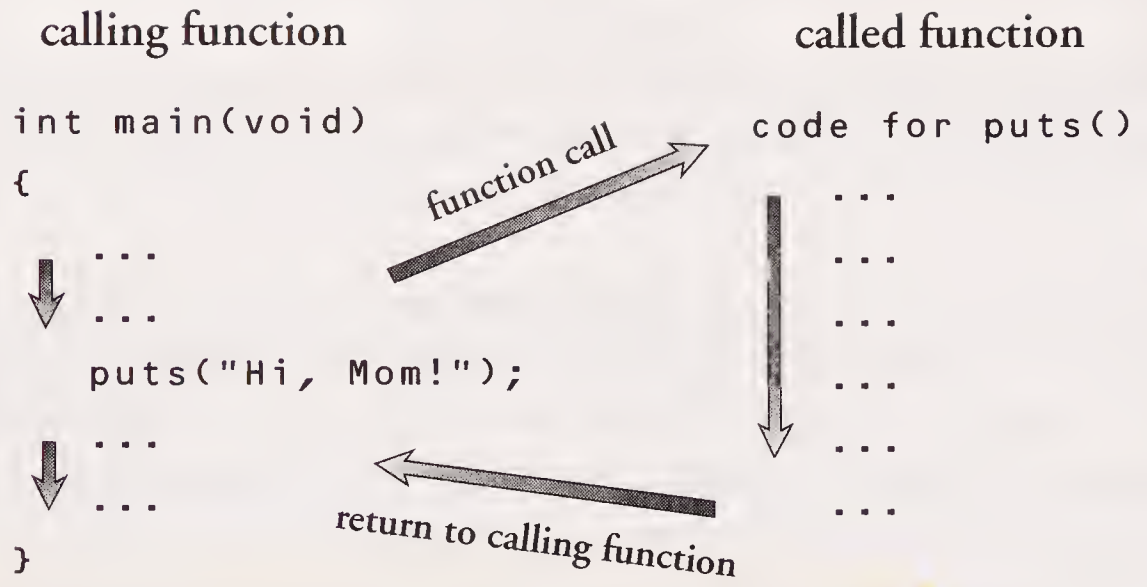


Figure 2-5 *Calling a function*

computer temporarily abandons `main()` and executes the statements in `puts()`. When it finishes with `puts()`, the computer then resumes executing statements in `main()`. Because the function call for `puts()` is in `main()`, we term `main()` the *calling function* and `puts()` the *called function*. (For some reason, calling them the *caller* and the *callee* never caught on.) See Figure 2-5.

To call a function, use the function name followed by parentheses. Many functions require information from the calling function. The `puts()` function, for example, requires a string, which it then displays. To pass such information to a function, place it within the parentheses. We call this passed information an *argument*. See Figure 2-6. An argument can be a string, integer, or other data type. Or there can be no argument at all. In that case, you still need to use parentheses:

```
showtime();   // calling a function with no arguments
```

Multiple arguments in C++ are separated by commas:

```
add_em_up( x, y, z, q, r); // function with list of arguments
```

MORE COUT

The third new feature of `yourcat.cpp` is combining three output statements into one. The `iostream.h` file defines the << operator so that you can combine (that is, *concatenate*) output as follows:

```
cout << "Well, that's " << fleas << " fleas too many.\n";
```

This lets us combine string output and integer output in a single statement. The resulting output is the same as what the following code produces:

```
cout << "Well, that's ";
cout << fleas;
cout << " fleas too many.\n";
```

While you're still in the mood for `cout`ish advice, you also can rewrite the concatenated version this way:

```
cout << "Well, that's "
     << fleas
     << " fleas too many.\n";
```

That's because C++'s free format rules treat newlines and spaces between tokens interchangeably. This last technique is convenient when the line width cramps your style.

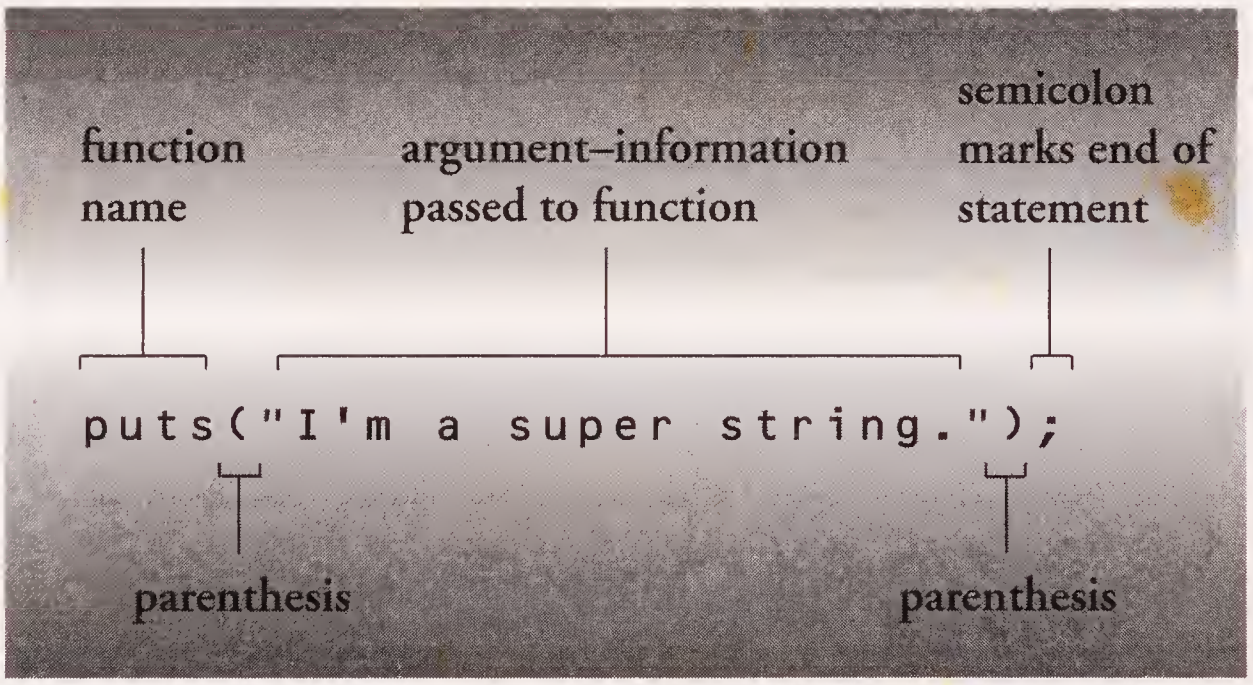


Figure 2-6 *Function call statement*

A TOUCH OF CLASS

You've seen enough of `cin` and `cout` to justify exposing you to a little object lore, in particular, let's talk some more about the notion of *classes*. Classes are one of the core concepts for object-oriented programming in C++.

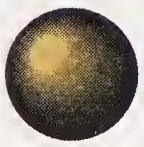
A class is a data type defined by the user. To define a class you describe what sort of information it can represent and what sort of actions you can perform with that data. A class bears the same relationship to an object that a type does to a variable. That is, a class definition describes a data form and how it can be used, while an object is an entity created according to the data form specification. Or, in noncomputer terms, if a class is analogous to a category such as famous actors, then an object is analogous to a particular

example of that category, such as Kermit the Frog. To extend the analogy, a class representation of actors would include definitions of possible actions relating to the class, such as Casting, Expressing Sorrow, Projecting Menace, Hyping, and the like. If you've been exposed to different OOP terminology, it may help to know that the C++ class corresponds to what some languages term an *object type*, and the C++ object corresponds to an *object instance* or *instance variable*.

Let's get a little more specific. Recall this declaration of a variable:

```
int fleas;
```

This creates a particular variable (*fleas*) that has the properties of the `int` type. That is, *fleas* can store an integer and can be used in particular ways—for addition and subtraction, for instance. Now consider `cout`. It is an object created to have the properties of the *ostream* class. The *ostream* class definition (another inhabitant of the `iostream.h` file) describes the sort of data an *ostream* object represents and the operations you can perform with and to it, such as insert a number or string into an output stream. Similarly, `cin` is an object created with the properties of the *istream* class, also defined in `iostream.h`.



The class describes all the properties of a data type, and an object is an entity created according to that description.

We've said that classes are user-defined types, but as users we certainly didn't design the *ostream* and *istream* classes. Just as functions can come in function libraries, classes can come in *class libraries*. That's the case for the *ostream* and *istream* classes. Technically, they are not built into the C++ language, but are examples of classes that happen to come with the language. The class definitions are laid out in the *iostream* file and are not built into the compiler. You can even modify these class definitions if you like, although that's not a good idea. The `iostream.h` family of classes and the related `fstream.h` (or file I/O) family are the only sets of class definitions that come with all implementations of C++. But many implementations, including Turbo C++ and Zortech C++, do provide additional class definitions as part of the package. Indeed, much of the current appeal of C++ is the existence of useful class libraries supporting UNIX, Macintosh, and Windows programming.

The class description specifies all the operations that can be performed on objects of that class. To perform such an allowed action on a particular object, you send a *message* to the object. For instance, if you want the `cout` object to display a string, you send it a message that says, in effect, "Object! Display this!" C++ provides a couple of ways to send messages. One way, called using a *class method*, essentially boils down to a function call like the ones we've seen. The other, and the one used with `cin` and `cout`, is to redefine an operator. Thus the statement

```
cout << "I am not a crook."
```

uses the redefined << operator to send the "display message" to `cout`. In this case, the message comes with an argument, which is the string to be displayed. See Figure 2-7.

STATEMENT SUMMARY

We've used and discussed several kinds of C++ statements, so let's summarize them now.

- Declaration statements announce the name and the kind of variables used in a function.
- Assignment statements use the assignment operator (=) to assign a value to a variable.
- Function calls activate a function. When the called function terminates, the program returns to the statement in the calling function immediately following the function call.
- Message statements send a message to an object, initiating some sort of action.

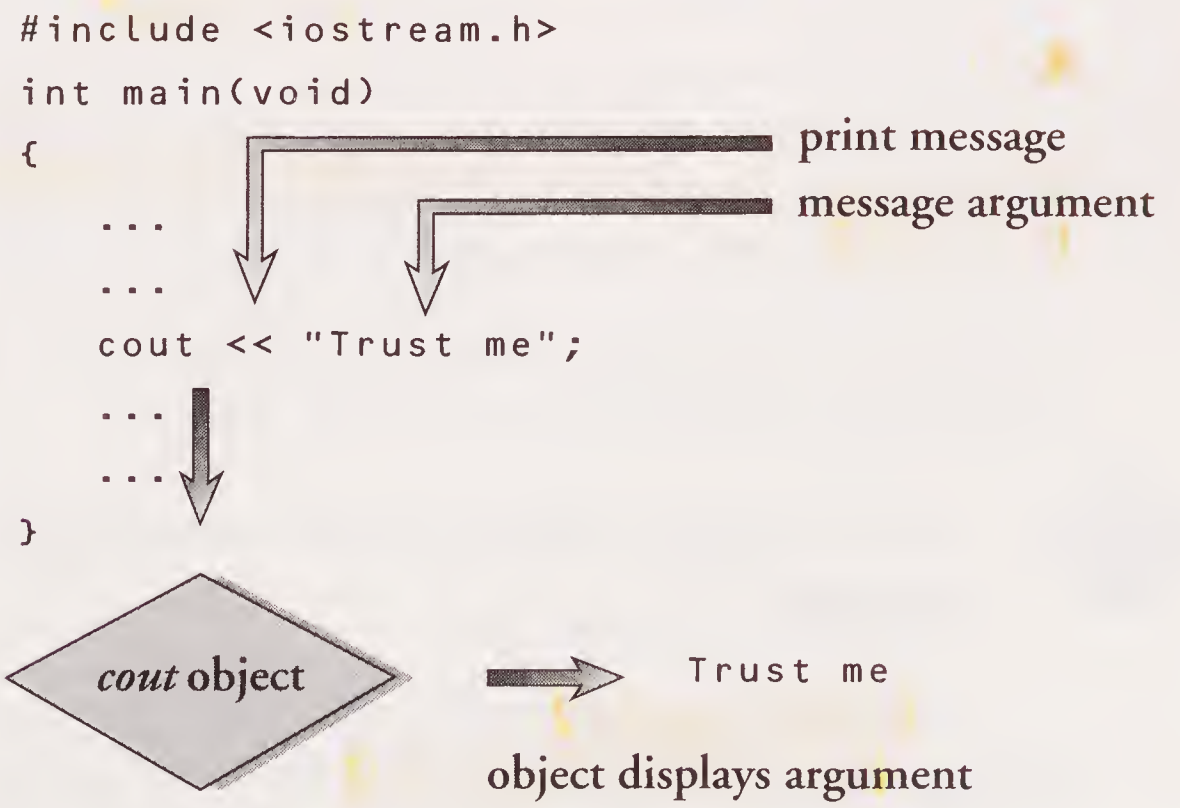


Figure 2-7 *Sending a message to an object*

FUNCTIONS

Because functions are the modules from which C++ programs are built and because they are essential to C++ OOP definitions, you should become thoroughly familiar with them. Because some aspects of functions are advanced topics, the main discussion of functions comes later, in Chapters 7 and 8. However, by dealing with some basic characteristics of functions now, you'll be more at ease and more practiced with functions later. The rest of this chapter introduces you to these function basics.

C++ functions come in two varieties: those with return values and those with none. You can find examples of each kind in the standard C++ library of functions, and you can create your own functions of each type. We'll look at a library function that has a return value, and then we'll examine how you can write your own simple functions.

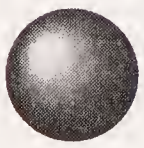
USING A FUNCTION WITH A RETURN VALUE

A function having a return value produces a value that can be assigned to a variable. For instance, the standard C++ library includes a function called `sqrt()` that returns the square root of a number. Suppose, for instance, you want to assign the square root of 6.25 to a variable *x*. (This is a normal, even rational, desire, so don't feel embarrassed about it.) Then you could use the following statement in your program:

```
x = sqrt(6.25); // returns the value 2.5 and assigns it to x
```

The expression `sqrt(6.25)` calls the `sqrt()` function and passes it the value *6.25* as an argument. The function calculates the answer to be *2.5* and returns that value to the calling program. You can think of the return value as being substituted for the function call in the statement.

That's practically all there is to it, except that before using a function, the C++ compiler must know what kind of arguments the function uses and what kind of return value it has. That is, does the function return an integer? A character? A number with a decimal fraction? A guilty verdict? Or something else? Lacking this information, the compiler won't know how to interpret the return value. The C++ way for conveying this information is using a *function prototype* statement.



A C++ program should provide a prototype for each function used in the program.

A function prototype does for functions what a variable declaration does for variables—it tells what types are involved. For instance, the C++ library defines the `sqrt()` function to take a number with (potentially) a fractional part (like 6.25) as an argument and to return a number of the same type. Some languages refer to such numbers as *real*

numbers, but the name C++ uses for this type is `double`. (You'll see more of `double` in Chapter 3.) The function prototype for `sqrt()` looks like this:

```
double sqrt(double);   // function prototype
```

The initial `double` means `sqrt()` returns a type `double` value. The `double` in the parentheses means `sqrt()` requires a `double` argument. So this prototype describes `sqrt()` exactly as we used it. By the way, the terminating semicolon identifies this line of text as a statement, making it a prototype instead of a function heading. If you omit the semicolon, the compiler will interpret the line as a function heading and expect you to follow it with a function body defining the function.

When you use `sqrt()` in a program, you also need to provide the prototype. You can do this in either of two ways:

- You can type the function prototype into your source code file yourself.
- You can include the `math.h` header file, which has the prototype in it.

The second way is better, for the header file is even more likely than you to get the prototype right. Every function in the C++ library has a prototype in one or more header files. Just check the function description in your manual or with on-line help, if you have it, and the description tells you which header file to use. For instance, the description of the `sqrt()` function should tell you to use the `math.h` header file.

Don't confuse the function *prototype* with the function *definition*. The prototype, as you've seen, only describes the function interface. The definition includes the code for the function workings. C and C++ divide these two features—prototype and definition—for library functions. The library files contain the compiled code for the functions, while the header files contain the prototypes.

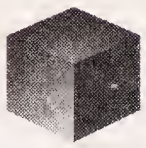
A function prototype should be placed ahead of where the function is first used. The usual practice is place prototypes just before the definition of the `main()` function. Listing 2-4 demonstrates using the library function `sqrt()`; it provides a prototype by including the `math.h` file.

LISTING 2-4

```
// sqrt.cpp -- use a square root function
#include <iostream.h>
#include <math.h>                  // use with math functions
int main(void)
{
    double cover;                  // use double for real numbers

    cout << "How many square feet of sheets do you have?\n";
```

```
    cin >> cover;
    double side;          // create another variable
    side = sqrt(cover);  // call function, assign return value
    cout << "You can cover a square with sides of " << side;
    cout << " feet\nwith your sheets.\n";
    return 0;
}
```



USING LIBRARY FUNCTIONS

C++ library functions are stored in library files. When the compiler compiles a program, it has to search the library files for the functions you've used. Compilers differ on which library files they search automatically. If you try to run Listing 2-4 and get a message `_sqrt` being an undefined external (sounds like a condition to avoid!), chances are that your compiler doesn't search the math library automatically. (Compilers like to add an underscore prefix to function names—another subtle reminder that they have the last say about your program.) If you get such a message, you'll have to check your compiler documentation to see how to have the compiler search the correct library. The usual UNIX implementations, for example, require that you use a `-lm` option (for *l*ibrary *m*ath) at the end of the command line:

```
CC sqrt.C -lm
```

Merely including the `math.h` header file provides the prototype but does not necessarily cause the compiler to search the correct library file.

Here's a sample run:

```
How many square feet of sheets do you have?
123.21
You can cover a square with sides of 11.1 feet
with your sheets.
```

Because `sqrt()` works with type `double` values, we made our variables that type. Note that declaring a type `double` variable uses the same form, or syntax, as declaring a type `int` variable:

```
typename variablename;
```

Being type `double` allows the variables *cover* and *side* to hold values with decimal fractions, such as *123.21* and *1.1*. As you'll see in Chapter 3, type `double` encompasses a much greater range of values than type `int`.

C++ does allow you to declare new variables anywhere in a program, so



`sqrt.cpp` didn't declare *side* until just before using it. C++ also allows you to assign a value to a variable when you create it, so you could also have done this:

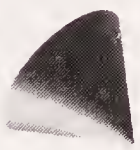
```
double side = sqrt(cover);
```

We'll return to this process, called *initialization*, in Chapter 3.

Note that `cin` knows how to convert information from the input stream to type `double`, and `cout` knows how to insert type `double` into the output stream. As we've said, these objects are smart.

USER-DEFINED FUNCTIONS

The standard C library provides over 140 predefined functions. If one fits your needs, by all means use it. But often you'll have to write your own, particularly when you design classes. Anyway, it's a lot more fun to design your own functions, so let's examine that process. You've already used several user-defined functions, and they all have been named `main()`. Every C++ program must have a `main()` function, which the user must define. Let's see how to add a second user-defined function. Just as with a library function, you can call a user-defined function by using its name. And, as with a library function, you need to provide a function prototype before using the function, typically by placing the prototype above the `main()` definition. The new element is that you also have to provide the source code for the new function. The simplest way is to place the code in the same file following the code for `main()`. Listing 2-5 illustrates these elements.



LISTING 2-5 our func.cpp

```
// ourfunc.cpp -- defining your own function
#include <iostream.h>
void simon(int);            // function prototype for simon()
int main(void)
{
simon(3);                   // call the simon() function
cout << "Pick an integer: ";
int count;
cin >> count;
simon(count);               // call it again
return 0;
}

void simon(int n)           // define the simon() function
{
```



```
cout << "Simon says touch your toes " << n << " times.\n";
}
                                        // void functions don't need return
```

The `main()` function calls the `simon()` function twice, once with an argument of 3, once with a variable argument *count*. In between, the user enters an integer that's used to set the value of *count*. We didn't use a newline character in the `cout` prompting message. This results in the user input appearing on the same line as the prompt. Here is a sample run:

```
Simon says touch your toes 3 times.
Pick an integer: 512
Simon says touch your toes 512 times.
```

Function Form

The definition for the `simon()` function follows the same general form as the definition for `main()`. First, there is a function header. Then, enclosed in braces, comes the function body. We can generalize the form for a function definition as follows:

```
type functionname(argumentlist)
{
    statements
}
```

Note that the source code defining `simon()` follows the closing brace of `main()`. Like C, and unlike Pascal, C++ does not allow you to embed one function definition inside another. Each function definition stands separately from all others; all functions are created equal. See Figure 2-8.

Function Headings

The `simon()` function had this heading:

```
void simon(int n)
```

The initial void means that `simon()` has no return value. So calling `simon()` doesn't produce a number that you can assign to a variable in `main()`. Thus the first function call looked like this:

```
simon(3);       // ok for void functions
```

Because poor `simon()` lacks a return value, you can't use it this way:

```
simple = simon(3);   // not allowed for void functions
```

The `int n` within the parentheses means that you are expected to use `simon()` with a single argument of type `int`. The *n* is a new variable to which is assigned the value passed during a function call. Thus the function call

```
simon(3);
```

assigns the value *3* to the *n* variable defined in the `simon()` heading. When the `cout` statement in the function body uses *n*, it uses the value passed in the function call. That's why `simon(3)` displays a *3* in its output. The call to `simon(count)` in our sample run caused the function to display *512* because that's the value we gave to `count`. In short, the heading for `simon()` tells us that this function takes a single type `int` argument and that it doesn't have a return value.

Let's review `main()`'s function header:

```
int main(void)
```

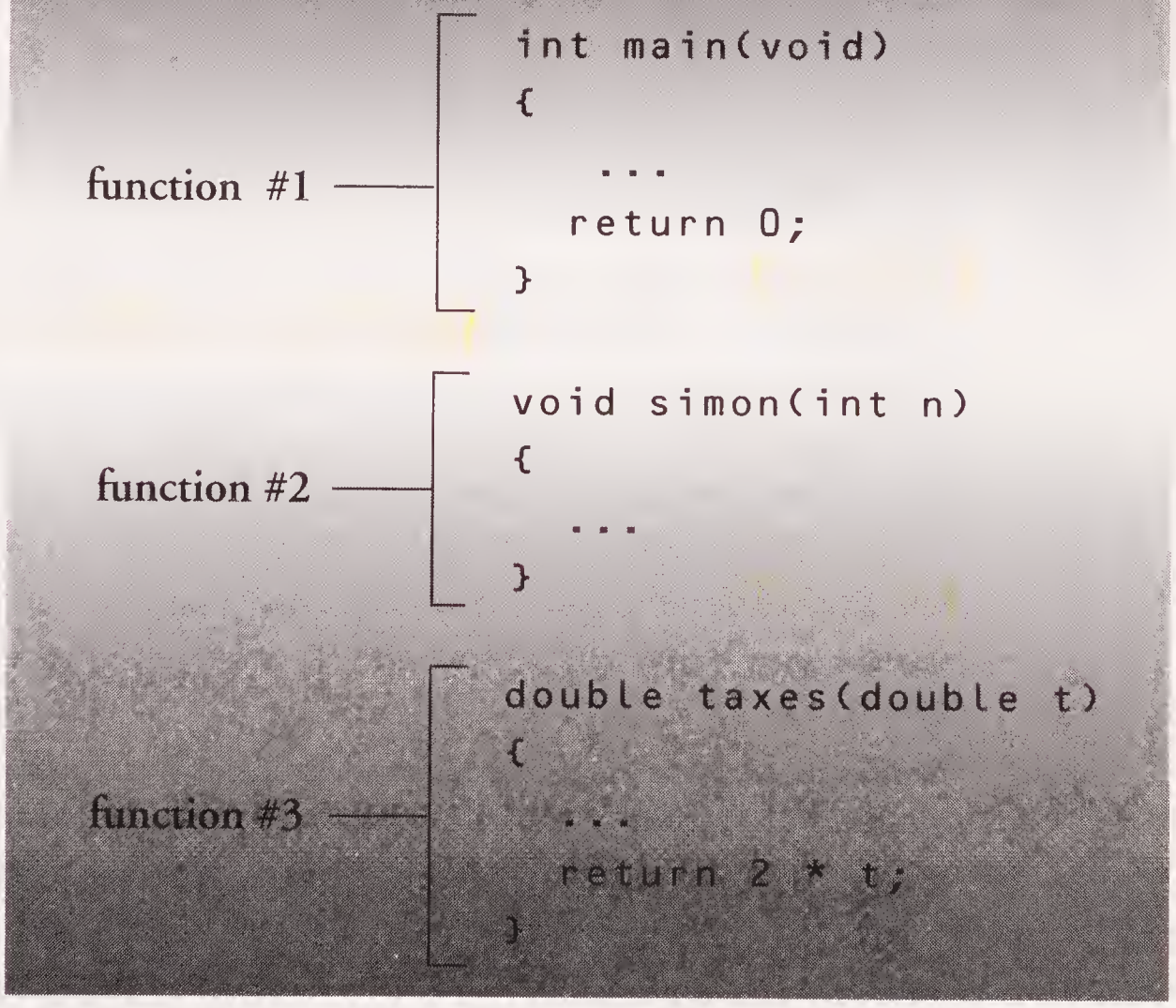


Figure 2-8 *Function definitions occur sequentially in a file*

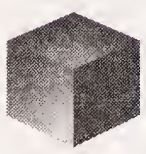
This header reverses the roles that `int` and `void` play in `simon()`. The initial `int` means that `main()` returns an integer value. The `void` means that `main()` has no arguments. Functions that have return values should use the keyword `return` to provide the return value and to terminate the function. That's why we've been using the following statement at the end of `main()`:

```
return 0;
```

This is logically consistent: `main()` is supposed to return a type `int` value, and we have it return the integer *0*. But, you may wonder, to what are we returning a value? After all, nowhere in any of our programs have you seen anything calling `main()`:

```
squeeze = main();   // absent from our programs
```

The answer is that you can think of your computer's operating system (UNIX, say, or DOS) as calling your program. So `main()`'s return value is returned not to another part of the program but to the operating system. Many operating systems can use the program's return value. For example, UNIX shell scripts and DOS batch files can be designed to run programs and test their return values, usually called *exit values*. The normal convention is that an exit value of zero means the program ran successfully, while a nonzero value means there was a problem. Thus you can design your C++ program to return a nonzero value if, say, it fails to open a file. Then you can design a shell script or batch file to run that program and to take some alternative action if the program signals failure.



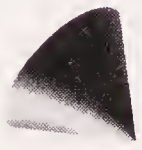
KEYWORDS

Keywords are the vocabulary of a computer language. In this chapter, we've used four C++ keywords: `int`, `void`, `return`, and `double`. Because these keywords are special to C++, you cannot use them for other purposes. That is, you can't use `return` as the name for a variable or `double` as the name of a function. But you can use them as part of a name, as `painter` or `return_aces`. Appendix B provides a complete list of C++ keywords. Incidentally, `main` is not a keyword, for it's not part of the language. Rather, it is the name of a required function. So you could use `main` as a variable name. (That could cause a problem in circumstances too esoteric to describe here, and it would be confusing in any case, so you'd best not.) Similarly, other function names and object names are not keywords. However, using the same name, say `cout`, for both an object and a variable in a program will confuse the compiler. That is, you can use `cout` as a variable name in a function that doesn't use the `cout` object for output, but you can't use `cout` both ways in the same function.



USER-DEFINED FUNCTION WITH A RETURN VALUE

Let's go one step further and write a function that uses the `return` statement. The `main()` function already illustrates the plan for a function with a return value: give the return type in the function heading and use `return` at the end of the function body. We'll use this form to solve a weighty problem for those visiting the United Kingdom. In the U.K., many bathroom scales are calibrated in *stone* instead of in familiar U.S. pounds. The word stone is both singular and plural in this context. (The English language does lack the internal consistency of, say, C++.) One stone is 14 pounds, and the program in Listing 2-6 uses a function to make this conversion.



LISTING 2-6 convert.cpp

```
// convert.cpp -- converts stone to pounds
#include <iostream.h>
int stonetolb(int);     // function prototype
int main(void)
{
    int stone;
    cout << "Enter the weight in stone: ";
    cin >> stone;
    int pounds = stonetolb(stone);
    cout << stone << " stone are ";
    cout << pounds << " pounds.\n";
    return 0;
}

int stonetolb(int sts)
{
    return 14 * sts;
}
```

Here's a sample run:

```
Enter the weight in stone: 14
14 stone are 196 pounds.
```

In `main()`, we use `cin` to provide a value for the integer variable *stone*. This value is passed to the `stonetolb()` function as an argument and is assigned to the variable *sts* in that function. Then we use the `return` keyword to return the value of *14 * sts* to `main()`. This illustrates that you aren't limited to following `return` with a simple number. Here, by using a more complex expression, we avoided the bother of creating a

new variable to which to assign the value before returning it. The program calculates the value of that expression (196 in our example) and returns the resulting value. If returning the value of an expression bothers you, you can take the longer route:

```
int stonetolb(int sts)
{
    int pounds = 14 * sts;
    return pounds;
}
```

Either version produces the same result, but the second version takes slightly longer to do so.

In general, you can use a function with a return value wherever you would use a simple constant of the same type. For instance, `stonetolb()` returns a type `int` value. This means you can use the function in the following ways:

```
int aunt = stonetolb(20);
int aunts = aunt + stonetolb(10);
cout << "Ferdie weighs " << stonetolb(16) << " pounds.\n";
```

In each case, the program will calculate the return value, then use that number in these statements.

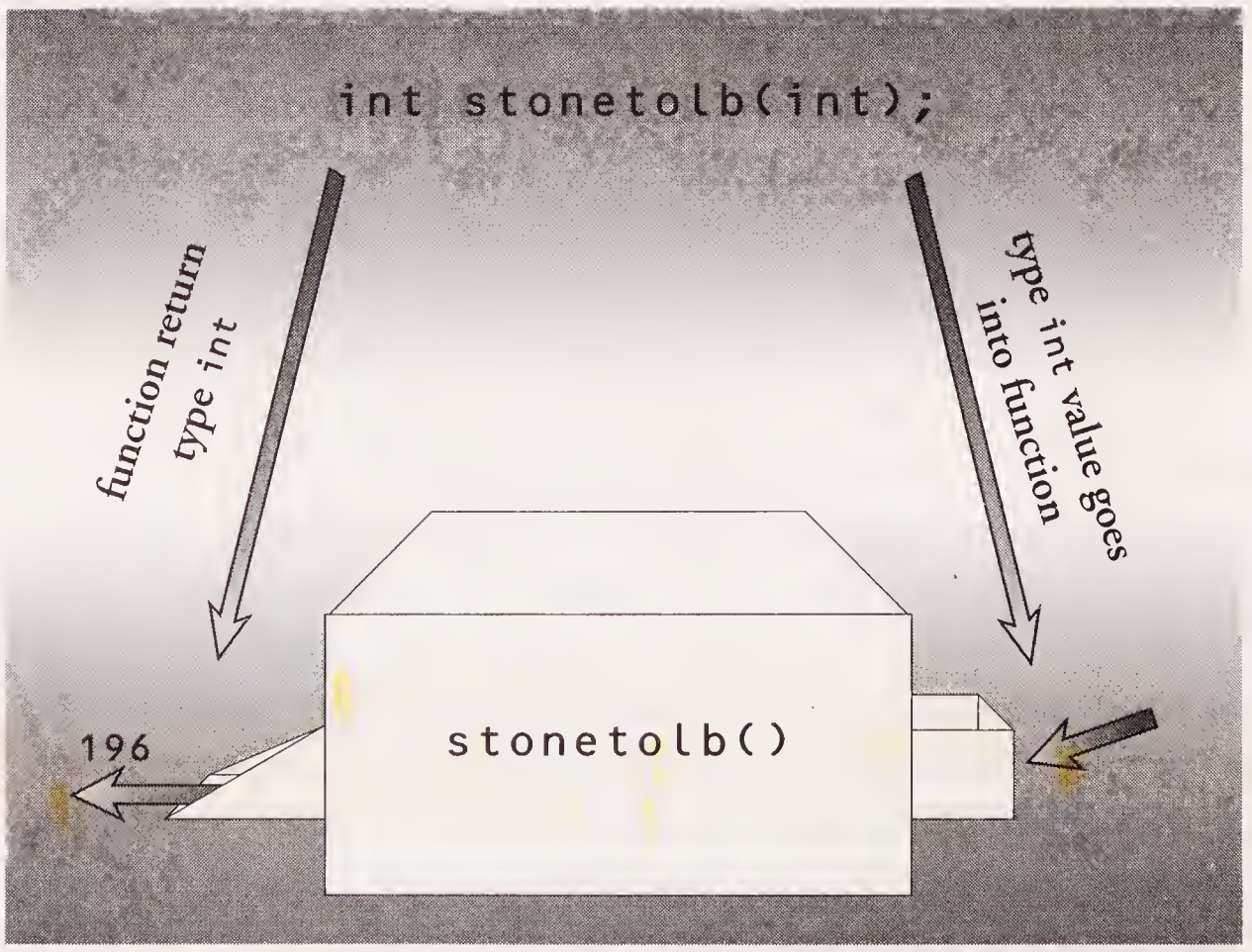


Figure 2-9 *The function prototype and the function as a black box*



As these examples show, the function prototype describes the function *interface*, that is, how it interacts with the rest of the program. The argument list shows what sort of information goes into the function, and the function type shows the type of value returned. Programmers sometimes describe functions as *black boxes* (a term from electronics) specified by the flow of information into them and out of them. The function prototype perfectly portrays that point of view. See Figure 2-9.

The `stonetolb()` function is short and simple, yet it embodies a full range of functional features:

- It has a heading and a body.
- It accepts an argument.
- It returns a value.
- It requires a prototype.

You can consider `stonetolb()` a template for function design. We'll go further into functions in Chapters 7 and 8. In the meantime, the material in this chapter should give you a good feel for how functions work and how they fit into C++.

SUMMARY

A C++ program consists of one or more modules called functions. Programs begin executing at the beginning of the function called `main()`, so you should always have a function by this name. A function, in turn, consists of a heading and a body. The function heading tells us what kind of return value, if any, the function produces and what sort of information it expects to be passed to it by arguments. The function body consists of a series of C++ statements enclosed in paired braces: `{}` .

C++ statement types include declaration statements, assignment statements, function call statements, object message statements, and return statements. The declaration statement announces the name of a variable and establishes the type of data it can hold. An assignment statement gives a value to a variable. A function call passes program control to the called function. When the function finishes, control returns to the statement in the calling function immediately following the function call. A message instructs an object to perform a particular action. A return statement is the mechanism by which a function returns a value to its calling function.

A class is a user-defined specification for a data type. This specification details how information is to be represented and also the operations that can be performed with the data. An object is an entity created according to a class prescription, just as a simple variable is an entity created according to a data type description.

C++ provides two predefined objects (`cin` and `cout`) for handling input and output. They are examples of the *istream* and *ostream* classes, which are defined in the

`iostream.h` file. These classes view input and output as streams of characters. The insertion operator (`<<`), which is defined for the *ostream* class, lets you insert data into the output stream, and the extraction operator (`>>`), which is defined for the *istream* class, lets you extract information from the input stream. Both `cin` and `cout` are smart objects, capable of automatically converting information from one form to another according to the program context.

Now that you have an overall view of simple C++ programs, we can go on in the next chapters to fill in detail and expand horizons.

Review Questions

Note: you'll find the answers to these and subsequent review questions in Appendix G.

1. What are the modules of C++ programs called?
2. What does the following preprocessor directive do?

```
#include <iostream.h>
```

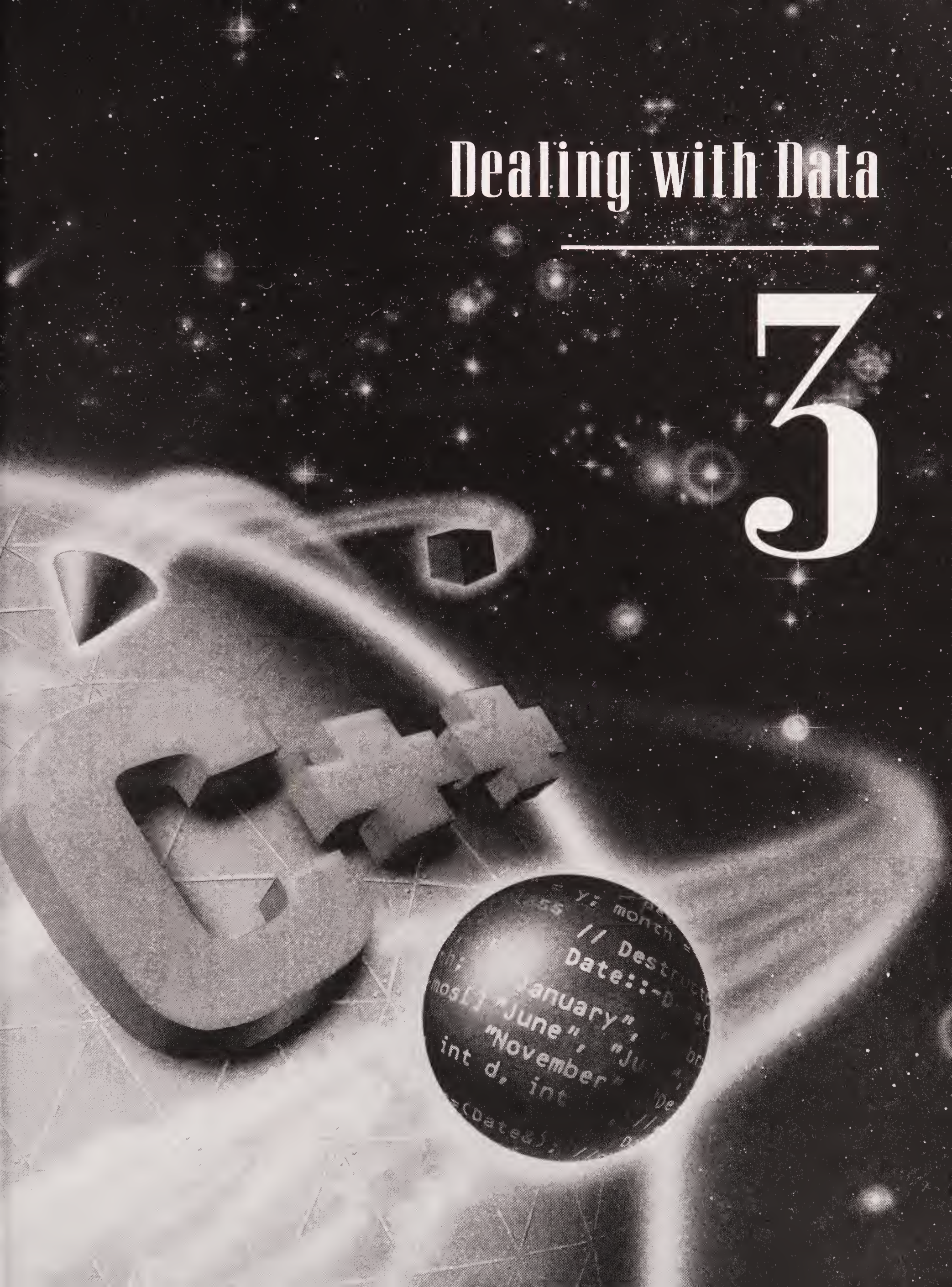
3. What statement would you use to print the phrase "Hello, world" and then start a new line?
4. What statement would you use to create an integer variable with the name *cheeses*?
5. What statement would you use to assign the value 32 to the variable *cheeses*?
6. What statement would you use to read a value from keyboard input into the variable *cheeses*?
7. What statement would you use to print "We have X varieties of cheese", where X will be replaced with the current value of the *cheeses* variable?
8. What does the following function header tell you about the function?

```
int froop(double t)
```

9. When do you not use the keyword `return` when defining a function?
10. Write a C++ program that asks for a distance in furlongs and converts it to yards (one furlong is 220 yards).
11. Redo question 10 by having `main()` call a user-defined function that takes the furlong value as an argument and then returns the yard value.

Dealing with Data

3



The essence of object-oriented programming is designing and extending your own data types. Designed types are not some frivolous trendy thing like designer togs; rather, they represent an effort to make a type match the data. If you do this properly, you'll find it much simpler to work with the data later. But before you can create your own types, you need to know and understand the types built into C++, for these types will be your building blocks.

The built-in C++ types come in two groups: fundamental types and derived types. In this chapter we'll look at the fundamental types, which represent integers and floating-point numbers. That may sound like just two types, however, C++ recognizes that no one integer type and no one floating-point type match all programming requirements, so it offers several variants on these two data themes. The next chapter follows up by covering several types derived from the basic types; these additional derived types include arrays, strings, pointers, and structures.

Of course, a program also needs a means to identify stored data. You'll examine one method for doing so—using variables. Next, you'll look at how to do arithmetic in C++. Finally, you'll see how C++ converts values from one type to another.

Simple Variables

Programs typically need to store information—perhaps the current price of IBM stock, perhaps the average humidity in New York City in August, perhaps the most common letter in the Constitution and its relative frequency, perhaps the number of available Elvis imitators. To store an item of information in a computer, the program needs to keep track of three fundamental properties:

- Where the information is stored
- What value is kept there
- What kind of information is stored

The strategy we've been using so far is to declare a variable. The type used in the declaration describes the kind of information, and the variable name represents the value symbolically. For instance, suppose Chief Lab Assistant Igor uses the following statements:

```
int braincount;
braincount = 5;
```

These statements tell the program that it is storing an integer and that the name *braincount* represents the integer's value, *5* in this case. These statements don't tell `us` (or Igor) where in memory the value is stored, but the program does keep track of that information, too. Indeed, you can use the & operator to retrieve *braincount's* address in memory. We'll take up that operator in the next chapter when we investigate a second strategy—using pointers—for representing data.

NAMES FOR VARIABLES

C++ encourages you to use meaningful names for variables. If a variable represents the cost of a trip, call it *costoftrip* or *costOfTrip*, not just *x* or *cot*. You do have to follow a few simple C++ naming rules:

- The only characters you may use in names are alphabetic characters, digits, and the underscore (_) character.
- The first character in a name cannot be a digit.
- Uppercase characters are considered distinct from lowercase characters.
- You can't use a C++ keyword for a name.
- C++ places no limits on the length of a name, and all characters in a name are significant.

The last point makes C++ different from ANSI C, which guarantees only that the first 31 characters in a name are significant. (That is, in ANSI C, two names having the same first 31 characters would be considered identical, even if the 32nd characters differed.) However, note that while the language standard may pose no length limits, some implementations may.

Here are some valid and invalid C++ names:

```
int poodle;       // valid
int Poodle;       // valid and distinct from poodle
```

```
int POODLE;       // valid and even more distinct
Int terrier;      // invalid -- has to be int, not Int
int my_stars3     // valid
int _mystars3;    // valid -- can start with underscore
int 4ever;        // invalid because starts with a digit
int double;       // invalid -- double is a C++ keyword
int begin;        // valid -- begin is a Pascal keyword
int the_very_best_variable_i_can_be_version_112;  // valid
int honky-tonk;  // invalid -- no hyphens allowed
```

Although you can begin a variable name with an underscore, as with `_mystars3`, you're better off if you don't. That's because C and C++ implementations and libraries use identifiers beginning with one or two underscore characters for their own purposes.

If you want to form a name from two or more words, the usual practice is to separate the words with an underscore character, as in `my_onions`, or to capitalize the initial character of each word after the first, as in `myEyeTooth`. (C veterans tend to use the underscore method in the C tradition, while Pascalians prefer the capitalization approach.)

Some C++ implementations have more restrictive rules for naming a variable if that variable is shared by more than one file in a multifile program. That's because some operating systems, such as PC DOS, have their own rules about variable names shared among files. For instance, PC DOS limits such shared names (called *external identifiers*) to eight characters and doesn't discriminate between uppercase and lowercase. Language designers regard such limitations as pathetic irritations.

INTEGER TYPES

Integers are whole numbers such as 2, 98, -5286, and 0. They have no fractional part. There are lots of integers, assuming you consider an infinite number to be a lot, so no finite amount of computer memory can represent all possible integers. Thus a language can only represent a subset of all integers. Some languages, such as Standard Pascal, offer just one integer type (one type fits all!), but C++ provides several choices. That gives you the option of choosing the integer type that best meets a program's particular requirements. This concern with matching type to data presages the designable data types of OOP.

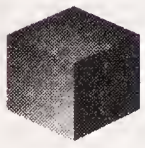
The various C++ integer types differ in the amount of memory they use to hold an integer. A larger block of memory can represent a larger range in integer values. Also, some types (*signed* types) can represent both positive and negative values, while others (*unsigned* types) can't represent negative values. C++'s basic integer types, in order of increasing size, are called `char`, `short`, `int`, and `long`. Each comes in both signed and unsigned versions. That gives you a choice of eight different integer types! Let's look at

these integer types in more detail. Because the `char` type has some special properties (it's most often used to represent characters instead of numbers), we'll discuss the other types first.

THE SHORT, INT, AND LONG INTEGER TYPES

Computer memory is organized into units called bytes. (See the Bits and Bytes note.) By using different numbers of bytes to store values, the C++ types of `short`, `int`, and `long` can represent up to three different integer sizes . It would be convenient if each type were always some particular size for all systems, for example, if `short` were always two bytes, `int` always four bytes, and so on. But life (even life++) is not that simple. The reason is that no one choice is suitable for all computer designs. So C++ offers a flexible standard with some guaranteed minimum sizes. Here's what you get:

- A `short` integer is at least two bytes.
- An `int` integer is at least as big as `short`.
- A `long` integer is at least four bytes and at least as big as `int`.



BITS AND BYTES

Memory capacity is measured in *bytes*. A byte consists of eight *bits*, where a bit is the smallest unit of memory. You can think of a bit as an electronic switch that can be set to either off or on. Off represents the value 0, and on represents the value 1. The eight-bit byte can be set to 256 different combinations. The number 256 comes from the fact that each bit has two possible settings, making the total number of combinations for 8 bits be $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$, or 256. Thus a byte can be used to represent, say, the values 0 through 255 or the values from -128 through 127. Each additional bit doubles the number of combinations. This means a two-byte unit can be set to 65536 different values, and a four-byte unit can be set to 4,294,672,296 different values.

Many systems currently use the minimum guarantee, making `short` two bytes and `long` four bytes. This still leaves several choices open for `int`. It could be two, three, or four bytes in size and meet the standard. Typically, `int` is two bytes (the same as `short`) for IBM PC implementations and four bytes (the same as `long`) for Macintosh, VAX, and many other minicomputer implementations. (What does your implementation use? The next example will show you how to determine the limits for your system without opening a manual.) The differences between implementations for type sizes can cause

problems when you move a C++ program from one environment to another. But a little care, as discussed later, can minimize those problems.

You use these type names just like int to declare variables:

```
short score;          // creates a type short integer variable
int temperature;      // creates a type int integer variable
long position;        // creates a type long integer variable
```

Actually, `short` is short for `short int` and `long` is short for `long int`, but hardly anyone uses the lengthier forms. (Don't extend the pattern too far—`int` is not short for `int int`.)

The three types `int`, `short`, and `long` are signed types, meaning each splits its range approximately equally between positive and negative values. For example, a two-byte `int` might run from -32768 to +32767.

If you want to know how your system's integers size up, C++ offers tools to let you investigate type sizes with a program. First, the `sizeof` operator returns the size, in bytes, of a type or a variable. (An operator is a built-in language element that operates on one or more items to produce a value. For instance, the addition operator, represented by +, adds two values.) Second, the `limits.h` header file contains information about integer type limits. In particular, it defines symbolic names to represent different limits. For instance, it defines `INT_MAX` to be the largest possible `int` value. Listing 3-1 demonstrates how to use these facilities. The program also illustrates *initialization*, which is using a declaration statement to assign a value to a variable.

LISTING 3-1 limits.cpp

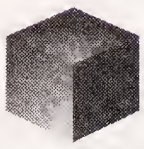
```
// limits.cpp -- some integer limits
#include <iostream.h>
#include <limits.h>          // defines limits for types
int main(void)
{
   int n_int = INT_MAX;      // initialize n_int to max int value
   short n_short = SHRT_MAX; // symbols defined in limits.h file
   long n_long = LONG_MAX;

   // sizeof operator yields size of type or of variable
   cout << "int is " << sizeof (int) << " bytes.\n";
   cout << "short is " << sizeof n_short << " bytes.\n";
   cout << "long is " << sizeof n_long << " bytes.\n\n";

   cout << "Maximum values:\n";
   cout << "int: " << n_int << "\n";
   cout << "short: " << n_short << "\n";
```

```
    cout << "long: " << n_long << "\n\n";

    cout << "Minimum int value = " << INT_MIN << "\n";
    return 0;
}
```



COMPATIBILITY NOTE

The `limits.h` header file is an ANSI C addition. Some C++ implementaions don't yet provide full ANSI C support and thus may not provide the `limits.h` header file. If you're using such a system, you'll have to limit yourself to experiencing this example in spirit only.

Here is a sample output:

```
int is 2 bytes.
short is 2 bytes.
long is 4 bytes.

Maximum values:
int: 32767
short: 32767
long: 2147483647

Minimum int value = -32768
```

PROGRAM DETAILS

Let's summarize the chief programming features for this program.

The `sizeof` operator reports that `int` is two bytes on our system. You can apply the `sizeof` operator to a type name or to a variable name. When you use the `sizeof` operator with a type name, such as `int`, you enclose the name in parentheses. But if you use the operator with the name of the variable, such as `n_short`, parentheses are optional:

```
cout << "int is " << sizeof (int) << " bytes.\n";
cout << "short is " << sizeof n_short << " bytes.\n";
```

The `limits.h` header file defines symbolic constants (see the Symbolic Constants box) to represent type limits. As mentioned, `INT_MAX` represents the largest value type `int` can hold; this turned out to be 32767 for our system. The compiler manufacturer provides a `limits.h` file that reflects the values appropriate to that compiler. For instance, the `limits.h` file for a system using a four-byte `int` would define the

INT_MAX to represent 2147483647. Both C++ and ANSI C use the limits.h file, so it's designed to be compatible with both languages. Table 3-1 summarizes the symbolic constants defined in this file; some pertain to types we have not yet discussed.

Initialization combines assignment with declaration. For instance, the statement

```
int n_int = INT_MAX;
```

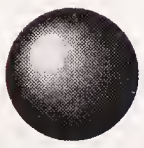
declares the n_int variable and sets it to the largest possible type int value. You can also use regular constants to initialize values. You can initialize a variable to another variable, providing the other variable has been defined first. You can even initialize a variable to an expression, providing all the values in the expression are known at compilation time:

```
int uncles = 5;                          // initialize to 5
int aunts = uncles;                      // initialize to 5
int chairs = aunts + uncles + 4;         // initialize to 14
```

TABLE 3-1 Symbolic constants from limits.h

SYMBOLIC CONSTANT	REPRESENTS
CHAR_BIT	number of bits in a char
CHAR_MAX	maximum char value
CHAR_MIN	minimum char value
SCHAR_MAX	maximum signed char value
SCHAR_MIN	minimum signed char value
UCHAR_MAX	maximum unsigned char value
SHRT_MAX	maximum short value
SHRT_MIN	minimum short value
USHRT_MAX	maximum unsigned short value
INT_MAX	maximum int value
INT_MIN	minimum int value
UINT_MAX	maximum unsigned int value
LONG_MAX	maximum long value
LONG_MIN	minimum long value
ULONG_MAX	maximum unsigned long value

Moving the `uncles` declaration to the end of this list of statements would invalidate the other two initializations, for then the value of `uncles` wouldn't be known at the time the compiler tried to initialize the other variables.

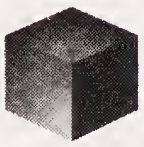


If you don't initialize a variable defined inside a function, the variable's value is *undefined*. That means the value will be whatever happened to be sitting at that memory location prior to creating the variable.

If you know what the initial value of a variable should be, initialize it. True, separating the declaring of a variable from assigning it a value can create momentary suspense:

```
short year;              // what could it be?
year = 1492;             // oh
```

But initializing the variable when you declare it protects you from forgetting to assign the value later.



SYMBOLIC CONSTANTS THE PREPROCESSOR WAY

The `limits.h` file contains lines similar to the following:

```
#define INT_MAX 32767
```

The C++ compilation process, recall, first passes the source code through a preprocessor. Here `#define`, like `#include`, is a preprocessor directive. What this particular directive tells the preprocessor is this: Look through the program for instances of `INT_MAX` and replace each occurrence with `32767`. So the `#define` directive works like a global search-and-replace command in an editor or word processor. The altered program is compiled after these replacements occur. The preprocessor looks for independent tokens (separate words), skipping embedded words. That is, the preprocessor doesn't replace `PINT_MAXIM` with `P32767IM`. You can use `#define` to define your own symbolic constants, too. (See Listing 3-2.) However, the `#define` directive is a C relic. C++ has a better way for creating symbolic constants (the `const` keyword, coming up soon), so we won't be using `#define` much. But some header files, particularly those designed to be used with both C and C++, do use it.

UNSIGNED TYPES

Each of the three integer types we just discussed comes in an unsigned variety, which can't hold negative values. This has the advantage of increasing the largest value the variable can hold. For instance, if `short` represents the range -32768 to +32767, then the unsigned version can represent the range 0 to 65535. Of course, you should use unsigned types only for quantities that are never negative, such as populations, inventory counts, and happy face manifestations. To create unsigned versions of the basic integer types, just use the keyword `unsigned` to modify the declarations:

```
unsigned short change;          // unsigned short type
unsigned int rovert;            // unsigned int type
unsigned quarterback;           // also unsigned int
unsigned long gone;             // unsigned long type
```

Note that unsigned by itself is short for `unsigned int`.

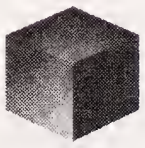
Listing 3-2 illustrates using unsigned types. It also shows what may happen if your program tries to go beyond the limits for integer types. Finally, it gives you one last look at the preprocessor `#define` statement.

LISTING 3-2 exceed.cpp

```
// exceed.cpp -- exceeding some integer limits
#include <iostream.h>
#define ZERO 0              // makes ZERO symbol for 0 value
#include <limits.h>         // defines INT_MAX as largest int value
int main(void)
{
    int sam = INT_MAX;      // initialize a variable to max value
    unsigned sue = sam;     // okay if variable sam already defined

    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\nAdd $1 to each account.\nNow ";
    sam = sam + 1;
    sue = sue + 1;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\nPoor Sam!\n";
    sam = ZERO;
    sue = ZERO;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\n";
    cout << "Take $1 from each account.\nNow ";
    sam = sam - 1;
    sue = sue - 1;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
```

```
    cout << " dollars deposited.\nLucky Sue!\n";
    return 0;
}
```



COMPATIBILITY NOTE

This example, like Listing 3-1, uses the `limits.h` file, which may not be available on C++ systems not providing full ANSI C support.

Here's the output:

```
Sam has 32767 dollars and Sue has 32767 dollars deposited.
Add $1 to each account.
Now Sam has -32768 dollars and Sue has 32768 dollars deposited.
Poor Sam!
Sam has 0 dollars and Sue has 0 dollars deposited.
Take $1 from each account.
Now Sam has -1 dollars and Sue has 65535 dollars deposited.
Lucky Sue!
```

The program sets an int variable (*sam*) and an unsigned int variable (*sue*) to the largest int value, which is 32767 on our system. Then it adds 1 to each value. This causes no problems for *sue*, for the new value is still much less than the maximum value for an unsigned integer. But *sam* goes from 32767 to -32768! Similarly, subtracting 1 from 0 creates no problems for *sam*, but it makes the unsigned variable sue go from 0 to 65535. As you can see, these integers behave much like an odometer or a VCR counter. If you go past the limit, the values just start over at the other end of the range. See Figure 3-1. C++ doesn't guarantee that integer types can exceed their limits (overflow and underflow) without complaint, but that is the most common behavior on current implementations.

WHICH TYPE?

With this richness of C++ integer types, which should you use? Generally, `int` is set to the most "natural" integer size for the target computer. Natural size means the integer form the computer handles most efficiently. If there is no compelling reason to choose another type, use `int`.

Now let's look at reasons you might use another type. If a variable represents something that never is negative, such as the number of words in a document, you can use an unsigned type; that way the variable can represent higher values.

If you know that the variable may have to represent integer values too great for a two-byte integer, use `long`. This is true even if `int` is four bytes on your system. That way, if you transfer your program to a system with two-byte `int`, your program won't embarrass you by suddenly failing to work properly. See Figure 3-2.

Using `short` can conserve memory if `short` is smaller than `int`. Most typically, this will be important only if you have a large array of integers. (An array is a data structure that stores several values of the same type sequentially in memory.) If it is important to conserve space, you should use `short` instead of `int` even if the two are the same size. Suppose, for example, you move your program from a two-byte `int` PC system to a four-byte `int` VAX system. That will double the amount of memory needed to hold an `int` array, but it won't affect the requirements for a `short` array. Remember, a byte saved is a byte earned.

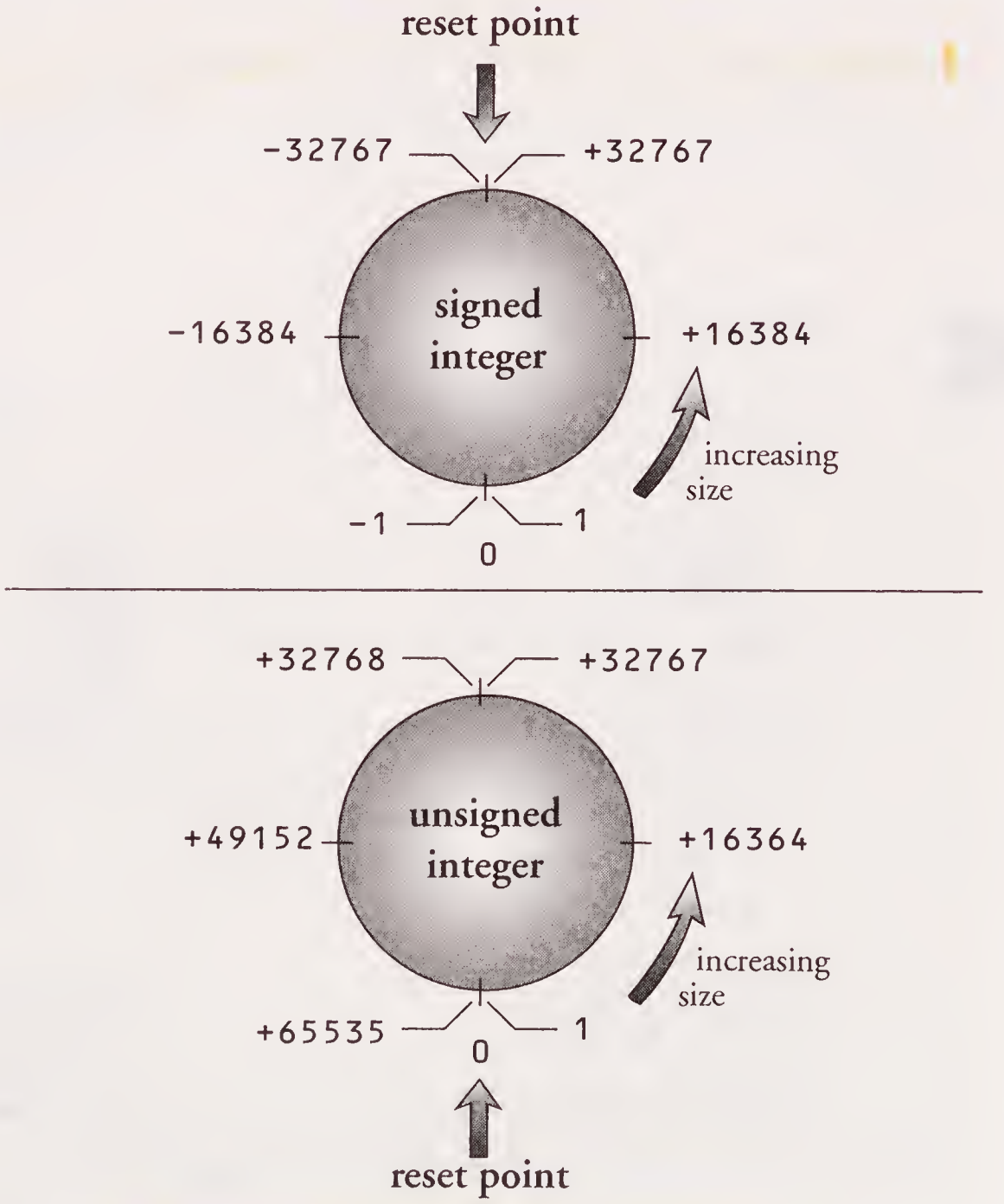


Figure 3-1 *Typical overflow behavior for integers*

If you need only a single byte, you can use `char`. We'll examine that possibility soon.

INTEGER CONSTANTS

An integer constant is one you write out explicitly, such as 212 or 1776. C++, like C, lets you write integers in three different number bases: base ten (the public favorite), base eight (the old UNIX favorite), and base 16 (the hardware hacker's favorite). Appendix A describes these bases; here we'll look at the C++ representations. C++ uses the first digit or two to identify the base of a number constant. If the first digit is in the range 1–9, the number is base 10 (decimal); thus 93 is base 10. If the first digit is 0 and the second digit is in the range 1–7, the number is base 8 (octal); thus 042 is octal and equal to 34 decimal. If the first two characters are 0x or 0X, the number is base 16 (hexadecimal); thus 0x42 is hex and equal to 66 decimal. For hexadecimal values, the characters a–f and A–F represent the hexadecimal digits corresponding to the values 10–15. So 0xF is 15 and 0xA5 is 165 (10 sixteens plus 5 ones). Listing 3-3 is tailor-made to show the three bases.

LISTING 3-3 hexoct.cpp

```
// hexoct.cpp -- shows hex and octal constants
#include <iostream.h>
int main(void)
{
    int chest = 42;       // decimal integer constant
    int waist = 0x42;     // hexadecimal integer constant
    int inseam = 042;     // octal integer constant

    cout << "Monsieur cuts a striking figure!\n";
    cout << "chest = " << chest << "\n";
    cout << "waist = " << waist << "\n";
    cout << "inseam = " << inseam << "\n";
    return 0;
}
```

By default, `cout` displays integers in decimal form, regardless of how they are written in a program, as the following output shows:

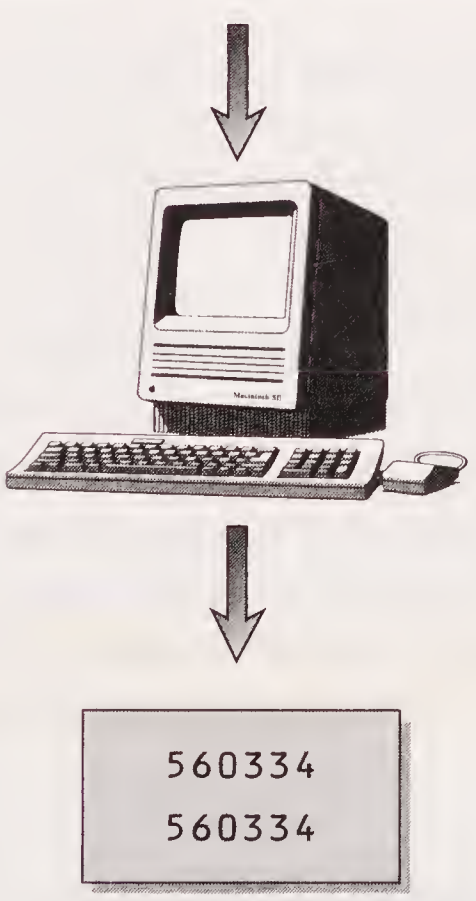
```
Monsieur cuts a striking figure!
chest = 42
```

```
waist = 66
inseam = 34
```

Keep in mind that these notations are merely notational conveniences. For instance, if you read that the CGA video memory segment is B000 in hexadecimal, you don't have to convert the value to base ten 45056 before using it in your program. Instead, simply use 0×B000. But whether you write the value ten as 10, 012, or 0×A, it's stored the same way in the computer—as a binary (base two) value.

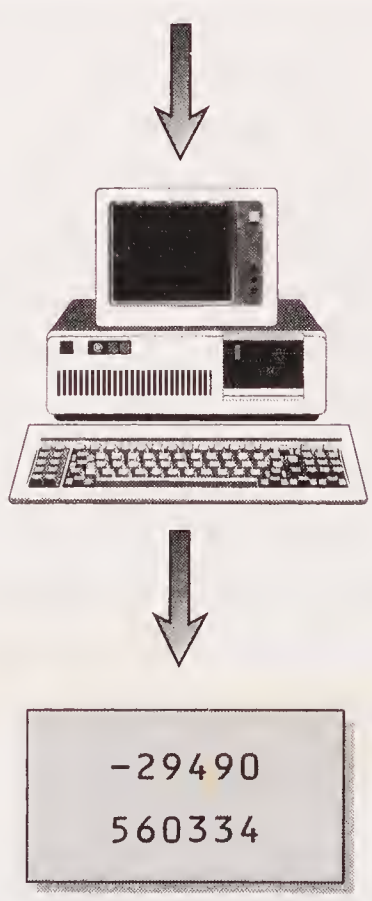
By the way, if you want to display a value in hexadecimal or octal form, you can use some special features of `cout`. We really don't want to get into that now, but you'll find the information in Chapter 12. (You can just skim the chapter for that information

```
// myprofit.cpp
...
int receipts = 560334;
long also = 560334;
cout << receipts << "\n";
cout << also << "\n";
...
```



Type `int` worked on this computer.

```
// myprofit.cpp
...
int receipts = 560334;
long also = 560334;
cout << receipts << "\n";
cout << also << "\n";
...
```



Type `int` failed on this computer.

Figure 3-2 *For portability, use* `long` *for big integers*

and ignore the explanations.) On input, `cin` recognizes the C++ notation for different number bases. So if you type 0×20, `cin` will interpret that as a hex value, the equivalent of 32.

How C++ Decides What Type a Constant Is

A program's declarations tell the C++ compiler the type of a particular integer variable. But what about constants? That is, suppose you represent a number with a constant in a program:

```
cout << "Year = " << 1492 << "\n";
```

Does the program store 1492 as an `int`, a `long`, or as some other integer type? The answer is that C++ stores integer constants as type `int` unless there is a reason to do otherwise. Two such reasons are if you use a special suffix to indicate a particular type or if a value is too large to be an `int`.

First, let's look at the suffixes. These are letters placed at the end of a numeric constant to indicate the type. An `l` or `L` suffix on an integer means the integer is a type `long` constant, a `u` or `U` suffix indicates an `unsigned int` constant, and `ul` (in any combination of orders and uppercase and lowercase) indicates a type `unsigned long` constant. (Because a lowercase `l` can look much like the digit `1`, we recommend using the uppercase `L` for suffixes.) For example, on a system using a two-byte `int` and a four-byte `long`, the number *22022* is stored in two bytes as an `int`, and the number *22022L* is stored in four bytes as a `long`.

Next, let's look at size. C++ has slightly different rules for decimal integers than it has for hexadecimal and octal integers. (Here decimal means base 10, just as hexadecimal means base 16; the term does not necessarily imply a decimal point.) A decimal integer without a suffix is represented by the smallest of the following types that can hold it: `int`, `long`, `unsigned long`. On a computer system using a two-byte `int` and a four-byte `long`, 20000 is represented as type `int`, 40000 is represented as `long`, and 3000000000 is represented as `unsigned long`. A hexadecimal or octal integer without a suffix is represented by the smallest of the following types that can hold it: `int`, `unsigned int`, `long`, `unsigned long`. So the same computer system that represents 40000 as `long` represents the hexadecimal equivalent 0×9C40 as an `unsigned int`. That's because hexadecimal frequently is used to express memory addresses, which intrinsically are unsigned. So `unsigned int` is more appropriate than `long` for a two-byte address.

The char Type: Characters and Small Integers

It's time to turn to the final integer type, type `char`. The `char` type, as you probably suspect from its name, was designed to store characters, such as letters and digits. Now, while

storing numbers is no big deal for computers, storing letters is another matter. Programming languages take the easy way out by using a number code for letters. Thus the `char` type is really another integer type. It's guaranteed to be large enough to represent the entire range of basic symbols—all the letters, digits, punctuation, and the like—for the target computer system. In practice, most systems support fewer than 256 kinds of characters, so a single byte can represent the whole range. Therefore, while `char` most often is used to handle characters, it also can be used as an integer type typically smaller than `short`.

The most common symbol set in the United States is the ASCII (pronounced *askey*) character set described in Appendix C. A numeric code (the ASCII code) represents the characters in the set. For example, 65 is the code for the character A. For convenience, we'll use ASCII code in our examples. However, a C++ implementation will use whatever code is native to its host system, for instance, EBCDIC (pronounced *eb-se-dik*) on an IBM mainframe.

Let's try the `char` type in Listing 3-4.

LISTING 3-4 chartype.cpp

```
// chartype.cpp -- the char type
#include <iostream.h>
int main(void)
{
    char ch;    // declare a char variable

    cout << "Enter a character:\n";
    cin >> ch;
    cout << "Yum! ";
    cout << "Thank you for the " << ch << " character.\n";
    return 0;
}
```

As usual, the `\n` notation is the C++ representation of the newline character. Here's the output:

```
Enter a character:
M
Yum! Thank you for the M character.
```

The interesting thing is that we type an M, not the corresponding character code of 77. And the program prints an M, not a 77. Yet if you were to peer into memory, you'd find that 77 is the value stored in the *ch* variable. The magic, such as it is, lies not in the `char` type, but in `cin` and `cout`. These worthy facilities make conversions on

our behalf. On input, `cin` converts the keystroke input M to the value 77. On output, `cout` converts the value 77 to the displayed character of M; `cin` and `cout` are guided by the type of variable. If we placed the same value of 77 into an `int` variable, then `cout` would display it as 77. (That is, `cout` will display two 7 characters.) Listing 3-5 illustrates this point. It also shows how to write a character constant in C++: enclose the character within two single quotation marks, as in *'M'*. (Note that we didn't use double quotation marks. C++ uses single quotation marks for a character and double quotation marks for a string. As you'll see in Chapter 4, the two are quite different.) Finally, the program introduces a `cout` offshoot, the `cout.put()` function.



LISTING 3-5 morechar.cpp

```
// morechar.cpp -- the char type and int type contrasted
#include <iostream.h>
int main(void)
{
    char c = 'M';           // assign ASCII code for M to c
    int i = c;              // store same code in an int
    cout << "The ASCII code for " << c << " is " << i << "\n";
    cout << "Add one to the character code:\n";
    c = c + 1;
    i = c;
    cout << "The ASCII code for " << c << " is " << i << '\n';

    // using the cout.put() member function to display a char
    cout << "Displaying char c using cout.put(c): ";
    cout.put(c);

    // cout.put() converts int to char on output
    cout << "\nDisplaying int i using cout.put(i): ";
    cout.put(i);

    cout << "\nDone\n";
    return 0;
}
```

Here is the output:

```
The ASCII code for M is 77
Add one to the character code:
The ASCII code for N is 78
Displaying char c using cout.put(c): N
```

```
Displaying int i using cout.put(i): N
Done
```

Program Notes

The notation 'M' represents the numeric code for the M character, so initializing the `char` variable *c* to *'M'* sets *c* to the value 77. The program then assigns the identical value to the `int` variable *i*. So both *c* and *i* have the value 77. Then `cout` displays *c* as M and *i* as 77. As we said, `cout` is guided by a value's type when choosing how to display it—just another example of smart objects.

Because *c* is really an integer, we can apply integer operations to it, such as adding 1. This changes the value of *c* to *78*. Then the program resets *i* to the new value. Again, `cout` displays the `char` version of that value as a character and the `int` version as number.

The fact that C++ represents characters as integers is a real convenience. It makes it easy to manipulate character values. You don't have to use awkward conversion functions to convert characters to ASCII and back.

Finally, the program uses the `cout.put()` function to display both *c* and *i*. Unlike simple `cout`, this function doesn't care whether its argument is type `char` or type `int`. In either case, it displays the character version of the value.

A Member Function: cout.put()

Just what is `cout.put()`? And why does it have a period in its name? The `cout.put()` function is our first example of an important C++ OOP concept, the *member* function. A class, you may remember, defines how to represent data and how to manipulate it. A member function belongs to a class and describes a method for manipulating class data. The `ostream` class, for instance, has a `put()` member function designed to output characters. You can use a member function only with a particular object of that class, such as the `cout` object, in this case. To use a class member function with an object like `cout`, you use a period to combine the object name (`cout`) with the function name (`put()`). The period is called the *membership* operator. The notation `cout.put()` means to use the class member function `put()` with the class object `cout`. Of course, we'll go into more detail when we reach classes. Meanwhile, the only classes we have are the *istream* and *ostream* classes, and we can experiment with their member functions to get more comfortable with the concept.

Using `cout.put()` is not quite the same as using the `<<` operator. As Listing 3-5 shows, `cout<<` displays a `char` value as a character and an `int` value as a number, but `cout.put()` displays both types as characters. At this point you may wonder why there is any need for `cout.put()`. Much of the answer is historical. Prior to Version 2.0 of C++, `cout` would display character *variables* as characters but display character *con-*

stants, such as *'M'* and *'\n'*, as numbers. The problem was that earlier versions of C++, like C, stored character constants as type `int`. That is, the code 77 for 'M' would be stored in a two-byte or four-byte unit. Meanwhile, `char` variables typically occupied one byte. A statement like

```
char c = 'M';
```

copied one byte (the important byte) from the constant *'M'* to the variable *c*. Unfortunately, this meant that *'M'* and *c* looked quite different to `cout`, even though both held the same value. So a statement like

```
cout << '$';
```

would print the ASCII code for the $ character instead of simply displaying $. But

```
cout.put('$');
```

would print the character, as desired. Now, Version 2.0 of C++ stores single character constants as type `char`, not type `int`. That means `cout` now correctly handles character constants. C++ always could use the string *"\n"* to start a new line; now it also can use the character *'\n'*:

Table 3-2 C++ escape sequence codes

CHARACTER NAME	ASCII SYMBOL	C++ CODE	ASCII DECIMAL CODE	ASCII HEX CODE
newline	NL (LF)	\n	10	0xA
horizontal tab	HT	\t	9	0x9
vertical tab	VT	\v	11	0xB
backspace	BS	\b	8	0x8
carriage return	CR	\r	13	0xD
alert	BEL	\a	7	0x7
backslash	\	\\	92	0x5C
question mark	?	\?	63	0x3F
single quote	'	\'	39	0x27
double quote	"	\"	34	0x22

```
cout << "\n"; // using a string
cout << '\n'; // using a character constant
```

A string is enclosed in double quotation marks instead of single quotation marks and can hold more than one character. Strings, even one-character strings, are *not* the same as type `char`. We'll come back to strings in the next chapter.

The `cin` object has a couple of different ways of reading characters from input. These are more easily explored using a program that uses a loop to read several characters, so we'll go into the matter when we cover loops.

CHAR CONSTANTS

You have several options for writing character constants in C++. The simplest choice for ordinary characters such as letters, punctuation, and digits is to enclose the character in single quotes. This notation stands for the numeric code for the character. For instance, an ASCII system has the following correspondences:

```
'A' is 65, the ASCII code for A
'a' is 97, the ASCII code for a
'5' is 53, the ASCII code for the digit 5
' ' is 32, the ASCII code for the space character
'!' is 33, the ASCII code for the exclamation mark
```

Using this notation is better than using the numeric codes explicitly. It's clearer, and it doesn't assume a particular code. If a system uses EBCDIC, then 65 is not the code for A, but '*A*' still represents the character.

Some characters can't be entered into a program directly from the keyboard. For instance, you can't make the newline character part of a string by typing the Return or Enter key; instead, the program editor interprets that keystroke as a request for it to start a new line in your source code file. Other characters have difficulties because the C++ language imbues them with special significance. For instance, the double quote character delimits strings, so you can't just stick one in the middle of a string. C++ has special notations, called *escape sequences*, for several of these characters, as shown in Table 3-2. For instance, `\a` represents the *alert* character, which beeps your terminal's speaker or rings its bell. And `\"` represents the double quotation mark as an ordinary character instead of a string delimiter. You can use these notations in strings or in character constants:

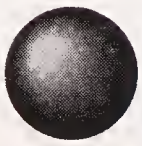
```
char alarm = '\a';
cout << alarm << "Don't do that again!\a\n";
cout << "Ben \"Buggsie\" Hacker was here!\n";
```

The last line produces the following output:

```
Ben "Buggsie" Hacker was here!
```

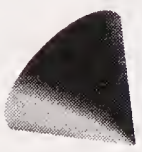
Note that you treat an escape sequence such as `\a` just like a regular character, such as *Q*. That is, you enclose it in single quotes to create a character constant and don't use single quotes when including it as part of a string.

Finally, you can use escape sequences based on the octal or hexadecimal codes for a character. For instance, Control-Z has an ASCII code of 26, which is 032 in octal and 0x1a in hexadecimal. You can represent this character by either of the following escape sequences: \032 or \x1a. You can make character constants out of these by enclosing them in single quotes, as in *'\032'*, and you can use them as parts of a string, as in *"hi\x1a there"*.



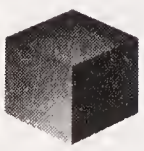
When you have a choice between using a numeric escape sequence or a symbolic escape sequence, as in `\0x8` versus `\b`, use the symbolic code. The numeric representation is tied to a particular code, such as ASCII, but the symbolic representation works with all codes and is more readable.

Listing 3-6 demonstrates a few escape sequences. It uses the alert character to get your attention, it uses the newline character to advance the cursor (one small step for a cursor, one giant step for cursorkind), and it uses the backspace character to back the cursor one space to the left. (Houdini once painted a picture of the Hudson River using only escape sequences; he was, of course, a great escape artist.)



LISTING 3-6 bondini.cpp

```
// bondini.cpp -- using escape sequences
#include <iostream.h>
int main(void)
{
    cout << "\aOperation \"HyperHype\" is now activated!\n";
    cout << "Enter your agent code:________\b\b\b\b\b\b\b\b";
    long code;
    cin >> code;
    cout << "\aYou entered " << code << "...\n";
    cout<<"\aCode verified! Proceed with Plan Z3!\n";
}
return 0;
```



COMPATIBILITY NOTE

Some C++ systems based on pre-ANSI C compilers don't recognize *\a*. You can substitute `\007` for `\a` on systems that use the ASCII character code.

When you start the program, it puts the following text on the screen:

```
Operation "HyperHype" is now activated!
Enter your agent code:________
```

After printing the underscore characters, the program uses the backspace character to back up the cursor to the first underscore. Then you can enter your secret code and continue. Here's a complete run:

```
Operation "HyperHype" is now activated!
Enter your agent code:42007007
You entered 42007007...
Code verified! Proceed with Plan Z3!
```

SIGNED CHAR AND UNSIGNED CHAR

Unlike `int`, `char` is not signed (that is, it allows both positive and negative values) by default. Nor is it unsigned (that is, it has no negative values) by default. The choice is left to the C++ implementation in order to allow the implementor to best fit the type to the hardware properties. If it is vital to you that `char` has a particular behavior, you can use `signed char` or `unsigned char` explicitly as types:

```
char fodo;          // may be signed, may be unsigned
unsigned char bar;  // definitely unsigned
signed char snark;  // definitely signed
```

These distinctions are particularly important if you're using `char` as a numeric type. The `unsigned char` type typically represents the range 0 to 255, and `signed char` typically represents the range -128 to 127. For example, suppose you want to use a `char` variable to hold values that may be as large as 200. That will work on some systems but fail on others. But you can use `unsigned char` successfully for that purpose on any system. On the other hand, if you're using a `char` variable to hold a standard ASCII character, it doesn't really matter whether `char` is signed or unsigned, so you can simply use `char`.

THE CONST QUALIFIER

Let's return to the topic of symbolic names for constants. The symbolic name can suggest what the constant represents. Also, if the program uses the constant in several places and you need to change the value, you can just change the single symbol definition. The box about `#define` statements earlier this chapter promised that C++ had a better way to handle symbolic constants. That way is to use the `const` keyword to modify a variable declaration and initialization. Suppose, for instance, that you want a symbolic constant for the number of months in a year. Enter this line:

```
const int Months = 12;  // Months is symbolic constant for 12
```

Now you can use *Months* in a program instead of *12*. (A bare 12 in a program might represent the number of inches in a foot or the number of donuts in a dozen, but the name *Months* tells you what it represents.) Once *Months* has been initialized, its value is set. The compiler will not let you subsequently change the value *Months*. For instance, Turbo C++ will give an error message stating that an Lvalue is required. This is the same message you'll get if you try, say, to assign the value 4 to 3. (An Lvalue is a value, such as a variable, that can appear on the left side of the assignment operator.) The keyword `const` is termed a *qualifier* because it qualifies the meaning of a declaration.

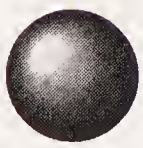
Capitalizing the name helps remind you that *Months* is a constant. This is by no means a universal convention, but it helps separate the constants from the variables when you read a program. Another convention is using all caps for a constant name. We'll follow the usual C convention and reserve all caps for constants created using `#define`, such as *INT_MAX*. Yet another convention is to begin constant names with the letter *k*, as in *kmonths*.

The general form for creating a constant is this:

```
const type name = value;
```

If you omit the type, C++ uses `int`.

If your background is in C, you might feel that the `#define` statement, which we discussed earlier, already does the job adequately. But `const` is better. For one thing, it lets you specify the type explicitly. Second, you can use C++'s scoping rules to limit the definition to particular functions or files. (Scoping rules describe how widely known a name is to different modules; we'll go into more detail in Chapter 8.) Third, you can use `const` with more elaborate types, such as the arrays and structures coming up in the next chapter.



If you are coming to C++ from C and you are about to use `#define` to define a symbolic constant, use `const` instead.

ANSI C also uses the `const` qualifier. If you're familiar with the ANSI C version, you should be aware that the C++ version is slightly different. One difference relates to the scope rules, and Chapter 8 will cover that point. The other main difference is that in C++ (but not in C) you can use a `const` value to declare the size of an array. We'll show examples in the next chapter.

FLOATING-POINT NUMBERS

Now that you have seen the complete line of C++ integer types, let's look at the floating-point types, which comprise the second major group of fundamental C++ types. These numbers let you represent numbers with fractional parts, such as the gas mileage of an M1 tank (0.56 mpg). They also provide a much greater range in values. If a number is too large to be represented as type *long*, for instance, the number of stars in our galaxy (an estimated 400,000,000,000), you can use one of the floating-point types.

With floating-point types, you can represent numbers like 2.5 and 3.14159 and 122442.32, that is, numbers with a fractional part. A computer stores such values in two parts. One part represents a value, and the other part scales that value up or down. Here's an analogy. Consider the two numbers 34.1245 and 34124.5. They're identical except for scale. We can represent the first one as 0.341245 (the base value) and 100 (the scaling factor). We can represent the second as 0.341245 (the same base value) and 100000 (a bigger scaling factor). The scaling factor serves to move the decimal point, hence the term *floating-point*. C++ uses a similar method to represent floating-point numbers internally, except it's based on binary numbers, so the scaling is by factors of 2 instead of by factors of 10. Fortunately, you don't have to know much about the internal representation. The main points are that floating-point numbers let you represent fractional values, they let you represent very large and very small values, and they have an internal representation much different from that of integers.

WRITING FLOATING-POINT NUMBERS

C++ has two ways of writing floating-point numbers. The first is to use the standard decimal-point notation you've been using much of your life:

```
12.34           // floating-point
939001.32       // floating-point
0.00023         // floating-point
8.0             // still floating-point
```

Even if the fractional part is 0, as in 8.0, the decimal point ensures that the number will be stored in floating-point format and not as an integer.

The second method for representing floating-point values is called *E notation*, and it looks like this: 3.45E6. This means that the value 3.45 is multiplied by 1,000,000; the E6 means 10 to the sixth power, which is 1 followed by 6 zeros. Thus 3.45E6 means 3,450,000. The 6 is called an *exponent*, and the 3.45 is termed the *mantissa*. Here are more examples:

```
2.52e+8          // can use E or e, + is optional
8.33E-4          // exponent can be negative
7E5              // same as 7.0E+05
-18.32e13        // can have + or - sign in front
2.857e12         // US public debt, 1989
5.98E24          // mass of Earth in kilograms
9.11e-31         // mass of an electron in kilograms
```

As you may have noticed, E notation is most useful for very large numbers and for very small numbers.

E notation guarantees that the number is stored in floating-point format, even if no decimal point is used. Note that you can use either E or e, and the exponent can have a positive or negative sign. See Figure 3-3. You can't have spaces in the number, however: 7.2 E6 would be invalid.

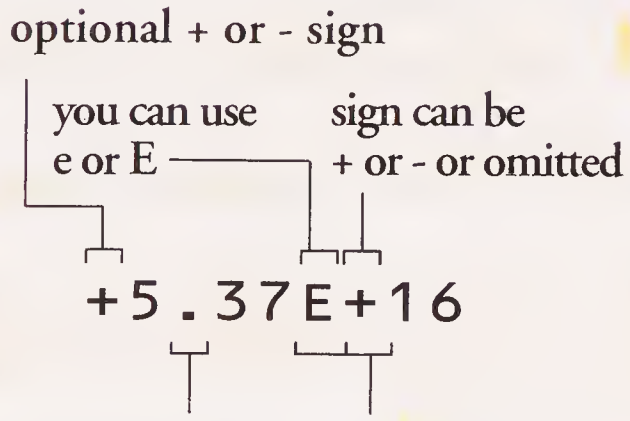
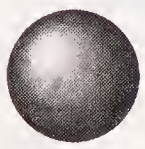


Figure 3-3 *E notation*

Using a negative exponent means to divide by a power of ten instead of multiplying by a power of ten. So 8.33E-4 means 8.33 ÷ 10^4, or 0.000833. Similarly, the electron mass of 9.11e-31 kg means 0.000000000000000000000000000000911 kg. Take your choice. (Incidentally, note that 911 is the usual emergency telephone number in the United States and that telephone messages are carried by electrons. Coincidence or scientific conspiracy? You be the judge.) Note that -8.33E4 means -83300. A sign in front applies to the number value, while a sign in the exponent applies to the scaling.



The form *d.ddd*E+*n* means move the decimal point *n* places to the right, and the form *d.ddd*E-*n* means move the decimal point *n* places to the left.

FLOATING-POINT TYPES

Like ANSI C, C++ has three floating-point types: `float`, `double`, and `long double`. These types are described in terms of the number of significant figures they can represent and the minimum allowable range of exponents. Significant figures are the meaningful digits in a number. For example, writing the height of Mt. Shasta in California as 14,162 feet uses five significant figures, for it specifies the height to the nearest foot. But

writing the height of Mt. Shasta as about 14,000 feet tall uses two significant figures, for the result was rounded to the nearest thousand feet. In this case, the remaining three digits are just place holders. The number of significant figures doesn't depend on the location of the decimal point. For instance, we can write the height as 14.162 thousand feet. Again, this uses five significant digits because the value is accurate to the fifth digit.

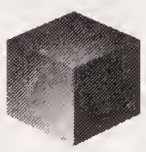
In effect, the C and C++ requirements for significant digits amount to `float` being at least four bytes, `double` being at least six bytes and certainly no smaller than `float`, and `long double` being at least as big as `double`. All three can be the same size. Typically, however, `float` is four bytes, `double` is eight bytes, and `long double` is ten, twelve, or sixteen bytes. Also, the range in exponents for all three types is at least -37 to +37. You can look in the `float.h` file to find the limits for your system. Here, for example, are some annotated entries from the `float.h` file for Turbo C++:

```
// the following are the minimum number of significant digits
#define DBL_DIG  15      // double
#define FLT_DIG  6       // float
#define LDBL_DIG 19      // long double

// the following are the number of bits used to represent
// the mantissa
#define DBL_MANT_DIG   53
#define FLT_MANT_DIG   24
#define LDBL_MANT_DIG  64

// the following are the maximum and minimum exponent values
#define DBL_MAX_10_EXP        +308
#define FLT_MAX_10_EXP        +38
#define LDBL_MAX_10_EXP       +4932

#define DBL_MIN_10_EXP        -307
#define FLT_MIN_10_EXP        -37
#define LDBL_MIN_10_EXP       -4931
```

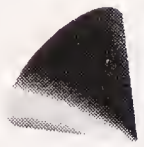


COMPATIBILITY NOTE

Some C++ implementations based on pre-ANSI C compilers don't provide a `float.h` header file.

Listing 3-7 examines types `float` and `double` and how they can differ in the precision to which they represent numbers (that's the significant figure aspect). The program also shows how `cout` will choose to use decimal-point notation or E notation, depending on the magnitude of the number being displayed.





LISTING 3-7 floatnum.cpp

```
// floatnum.cpp -- floating-point types in Turbo C++
#include <iostream.h>
int main(void)
{
    float tub = 10.0 / 3.0;      // good to about 6 places
    double mint = 10.0 / 3.0;    // good to about 15 places
    const float million = 1.0e6;

    cout << "tub = " << tub;
    cout << ", a million tubs = " << million * tub;
    cout << ",\nand ten million tubs = ";
    cout << 10 * million * tub << "\n";

    cout << "mint = " << mint << " and a million mints = ";
    cout << million * mint << "\n";
    return 0;
}
```

Here is the output:

```
tub = 3.333333, a million tubs = 3333333.25,
and ten million tubs = 3.333333e+07
mint = 3.333333 and a million mints = 3333333.333333
```

PROGRAM NOTES

First, note in the second line of output how `cout` switches from decimal notation to E notation when the number of digits to the left of the decimal reaches 7. Second, note that `cout` usually displays six digits to the right of the decimal. The exception is that `cout` drops trailing zeros. That is, it displays 3333333.250000 as 3333333.25. Third, note how `float` has less precision than `double`. Both *tub* and *mint* are initialized to 10.0 / 3.0. That should evaluate to 3.3333333333333333...(etc.). Since `cout` prints six figures to the right of the decimal, we can see that both *tub* and *mint* are accurate that far. But, after the program multiplies each number by a million, we see that `tub` digresses from the proper value after the seventh 3. We describe this by saying `tub` is good to seven significant figures. (Our system guarantees six significant figures for `float`, but that's the worst-case scenario.) The type `double` variable, however, shows thirteen 3s, so it's good to at least thirteen significant figures. The system guarantees fifteen, so we shouldn't be surprised.

The `iostream` class to which `cout` belongs also has class member functions that give you precise control on how the output is formatted—field widths, places to the right of the decimal point, decimal form or E form, and so on. Chapter 12 outlines those choices. This book's examples keep it simple and just use the << operator. Occasionally, this practice displays more digits than necessary, but that causes only esthetic harm. If you do mind, you can skim through Chapter 12 to see how to use the formatting methods. Don't, however, expect to fully follow the explanations at this point.

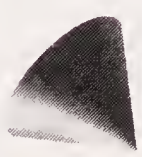
FLOATING-POINT CONSTANTS

When you write a floating-point constant in a program, in which floating-point type does the program store it? By default, floating-point constants like 8.24 and 2.4E8 are type `double`. If you want a constant to be type `float`, use an *f* or *F* suffix. For type `long double`, use an *l* or *L* suffix.

```
1.234f              // a float constant
2.45E20F            // a float constant
2.345324E28         // a double constant
2.2L                // a long double constant
```

FLOATING-POINT ADVANTAGES AND DISADVANTAGES

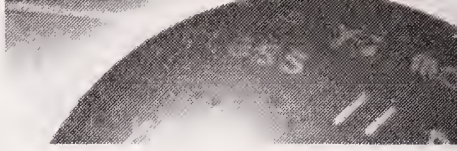
Floating-point numbers have two advantages over integers. First, they can represent values between the integers. Second, because of the scaling factor, they can represent a much greater range of values. On the other hand, floating-point operations are usually slower than integer operations, and you can lose precision. Listing 3-8 illustrates the last point.



LISTING 3-8 fltadd.cpp

```
// fltadd.cpp -- precision problems with float
#include <iostream.h>
int main(void)
{
    float a = 2.34E+22;
    float b = a + 1;

    cout << "a = " << a << "\n";
    cout << "b - a = " << b - a << "\n";
    return 0;
}
```



The program takes a number, adds 1, then subtracts the original number. That should result in a value of one. Does it? Here is the output:

```
a = 2.34e+22
b - a = 0
```

The problem is that 2.34E+22 represents a number with twenty-three digits to the left of the decimal place. By adding 1, we are attempting to add 1 to the twenty-third digit in that number. But type `float` can represent only the first six or seven digits in a number, so trying to change the twenty-third digit has no effect on the value.

C++ Arithmetic Operators

Perhaps you have warm memories of doing arithmetic drills in grade school. You can give that same pleasure to your computer. C++ uses operators to do arithmetic. It provides operators for five basic arithmetic calculations: addition, subtraction, multiplication, division, and taking the modulus. Each of these operators uses two values (called *operands*) to calculate a final answer. Together, the operator and its operands constitute an *expression*. For instance, consider the following statement:

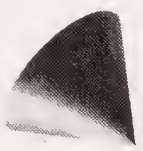
```
int wheels = 4 + 2;
```

The values *4* and *2* are operands, the + symbol is the addition operator, and *4* + *2* is an expression whose value is 6.

Here are C++'s five basic arithmetic operators:

- The + operator adds its operands. For instance, 4 + 20 evaluates to 24.
- The - operator subtracts the second operand from the first. For instance, 12 - 3 evaluates to 9.
- The * operator multiplies its operands. For instance, 28 * 4 evaluates to 112.
- The / operator divides its first operand by the second. For instance, 1000 / 5 evaluates to 200.
- The % operator finds the modulus of its first operand relative to the second. That is, it produces the remainder of dividing the first by the second. For instance, 19% 6 is 1, since 6 goes into 19 three times with a remainder of 1. Both operands must be integer types.

Of course, you can use variables as well as constants for operands. Listing 3-9 does just that. Because the % operator works with integers only, we'll save it for a later example.



LISTING 3-9 arith.cpp

```
// arith.cpp -- some C++ arithmetic
#include <iostream.h>
int main(void)
{
    float hats, heads;

    cout << "Enter a number: ";
    cin >> hats;
    cout << "Enter another number: ";
    cin >> heads;

    cout << "hats = " << hats << "; heads = " << heads << "\n";
    cout << "hats + heads = " << hats + heads << "\n";
    cout << "hats - heads = " << hats - heads << "\n";
    cout << "hats * heads = " << hats * heads << "\n";
    cout << "hats / heads = " << hats / heads << "\n";
    return 0;
}
```

Here's sample output. As you can see, you can trust C++ to do simple arithmetic:

```
Enter a number: 50.25
Enter another number: 11.17
hats = 50.25; heads = 11.17
hats + heads = 61.419998
hats - heads = 39.080002
hats * heads = 561.29248
hats / heads = 4.498657
```

Well, maybe you can't trust it completely. Adding 11.17 to 50.25 should yield 61.42, but the output reports 61.419998. This is not an arithmetic problem, it's a problem with the limited capacity of type `float` to represent significant figures. Remember, C++ guarantees just six significant figures for `float`. If you round 61.419998 to six figures, you get 61.4200, which is the correct value to the guaranteed precision. The moral is that if you need greater accuracy, use `double` or `long double`.

WHICH ORDER: OPERATOR PRECEDENCE AND ASSOCIATIVITY

Can you trust C++ to do complicated arithmetic? Yes, but you need to know the rules C++ uses. For instance, many expressions involve more than one operator. That can raise questions about which operator gets applied first. For example, consider this statement:

```
int flyingpigs = 3 + 4 * 5;  // 35 or 23?
```

The *4* appears to be an operand for both the + and the * operators. C++ uses *precedence* rules to decide which operator goes first. The arithmetic operators follow the usual algebraic precedences, with multiplication, division, and taking the modulus being done before addition and subtraction. Thus 3 + 4 * 5 means 3 + (4 * 5), not (3 + 4) * 5. So the answer is 23, not 35. Of course, you can use parentheses to enforce your own priorities. Appendix D shows the precedences for all the C++ operators. In it, you'll note that *, /, and % are all in the same row. That means they have equal precedence. Similarly, addition and subtraction share a lower precedence.

Sometimes the precedence list is not enough. Consider the next statement:

```
float logs = 120 / 4 * 5;    // 150 or 6?
```

Once again 4 is an operand for two operators. But the / and * operators have the same precedence, so precedence alone doesn't tell the program whether to first divide 120 by 4 or to first multiply the 4 by 5. Since the first choice leads to a result of 150 and the second to a result of 6, the choice is an important one. When two operators have the same precedence, C++ looks at whether the operators have a left-to-right *associativity* or a right-to-left associativity. Left-to-right associativity means that if two operators acting upon the same operand have the same precedence, apply the left-hand operator first. For right-to-left associativity, apply the right-hand operator first. The associativity information, too, is in Appendix D. There you'll see that multiplication and division associate left-to-right. That means we use 4 with the leftmost operator first. That is, we divide 120 by 4, getting 30, then multiply the result by 5 to get 150.

Note that the precedence and associativity rules come into play only when two operators share the same operand. Consider the following expression:

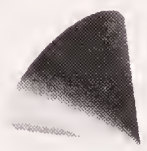
```
int dues = 20 * 5 + 24  *  6;
```

Operator precedence tells us two things: the program must evaluate 20 * 5 before doing addition, and the program must evaluate 24 * 6 before doing addition. But neither precedence nor associativity says which multiplication takes place first. You might think that associativity says to do the leftmost multiplication first, but in this case, the two * operators do not share a common operand, so the rules don't apply. In fact, C++ leaves it to the implementation to decide which order works best on a system. For our example, either order gives the same result, but there are circumstances in which the order can make a difference. You'll see one when Chapter 5 discusses the increment operator.

DIVISION DIVERSIONS

You have yet to see the rest of the story about the division operator. The behavior of this operator depends on the type of the operands. If both operands are integers, C++ per-

forms integer division. That means any fractional part of the answer is discarded, making the result an integer. If one or both operands are floating-point values, then the fractional part is kept, making the result floating-point. Listing 3-10 illustrates how C++ division works with different types of values.



LISTING 3-10 divide.cpp

```
// divide.cpp -- integer and floating-point division
#include <iostream.h>
int main(void)
{
    cout << "Integer division: 9/5 = " << 9 / 5  << "\n";
    cout << "Floating-point division: 9.0/5.0 = ";
    cout << 9.0 / 5.0 << "\n";
    cout << "Mixed division: 9.0/5 = " << 9.0 / 5  << "\n";
    cout << "double constants: 1e7/9.0 = ";
    cout << 1.e7 / 9.0 <<  "\n";
    cout << "float constants: 1e7f/9.0f = ";
    cout << 1.e7f / 9.0f <<  "\n";
    return 0;
}
```



COMPATIBILITY NOTE

Some C++ implementations based on pre-ANSI C compilers don't support the *f* suffix for floating-point constants. If you find yourself facing this problem, you can replace *1.e7f / 9.0f* with *(float) 1.e7 / (float) 9.0*.

Here is the output:

```
Integer division: 9/5 = 1
Floating-point division: 9.0/5.0 = 1.8
Mixed division: 9.0/5 = 1.8
double constants: 1e7/9.0 = 1111111.111111
float constants: 1e7f/9.0f = 1111111.125
```

The first output line shows that dividing the integer 9 by the integer 5 yields the integer 1. The fractional part of 4/5 (or 0.8) is discarded. You'll see a practical use for this kind of division when we discuss the modulus operator. The next two lines show that when at least one of the operands is floating-point, you get a floating-point answer of 1.8. Actually, when you try to combine mixed types, C++ converts all the concerned

types to the same type. We'll discuss these automatic conversions later in the chapter. The relative precisions of the last two lines show that the result is type *double* if both operands are *double* and that it is *float* if both operands are *float*. Remember, floating-point constants are type *double* by default.

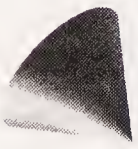


A GLIMPSE OF OPERATOR OVERLOADING

In Listing 3-10, the division operator represents three distinct operations: `int` division, `float` division, and `double` division. C++ uses the context, in this case the type of operands, to determine which operator is meant. The process of using the same symbol for more than one operation is called *operator overloading*. C++ has a few examples of overloading built into the language. C++ also lets you extend operator overloading to user-defined classes, so what we've seen here is a precursor of an important OOP property. See Figure 3-4.

THE MODULUS OPERATOR

Most people are more familiar with addition, subtraction, multiplication, and division than with the modulus operator, so we'll take a moment to illustrate the latter operator in action. The modulus operator, recall, returns the remainder of an integer division. In combination with integer division, the modulus operator is particularly useful in problems that require dividing a quantity into different integral units, such as converting inches to feet and inches or converting dollars to quarters, dimes, nickels, and pennies. In Chapter 2, we converted weight in British stone to pounds. Listing 3-11 reverses the process, converting weight in pounds to stone. The stone, you remember, is 14 pounds, and most British bathroom scales are calibrated in this unit. The program uses integer division to find the largest number of whole stone in the weight, and it uses the modulus operator to find the number of pounds left over.



LISTING 3-11 modulus.cpp

```
// modulus.cpp -- uses % operator to convert lbs to stone
#include <iostream.h>
int main(void)
{
    const int Lbs_per_stn = 14;
    int lbs;
```

```
    cout << "Enter your weight in pounds: ";
    cin >> lbs;
    int stone = lbs / Lbs_per_stn;     // whole stone
    int pounds = lbs % Lbs_per_stn;    // remainder in pounds
    cout << lbs << " pounds are " << stone;
    cout << " stone, " << pounds << " pound(s).\n";
    return 0;
}
```

Here is a sample run:

```
Enter your weight in pounds: 185
185 pounds are 13 stone, 3 pound(s).
```

In the expression `lbs / Lbs_per_stn`, both operands are type `int`, so the computer performs integer division. With a *lbs* value of *185*, the expression evaluates to *13*. The product of 13 and 14 is 182, so the remainder of dividing 14 into 185 is 3, and that's the value of `lbs % Lbs_per_stn`. Now you are prepared, technically if not emotionally, to respond to questions about your weight when you travel in Great Britain.

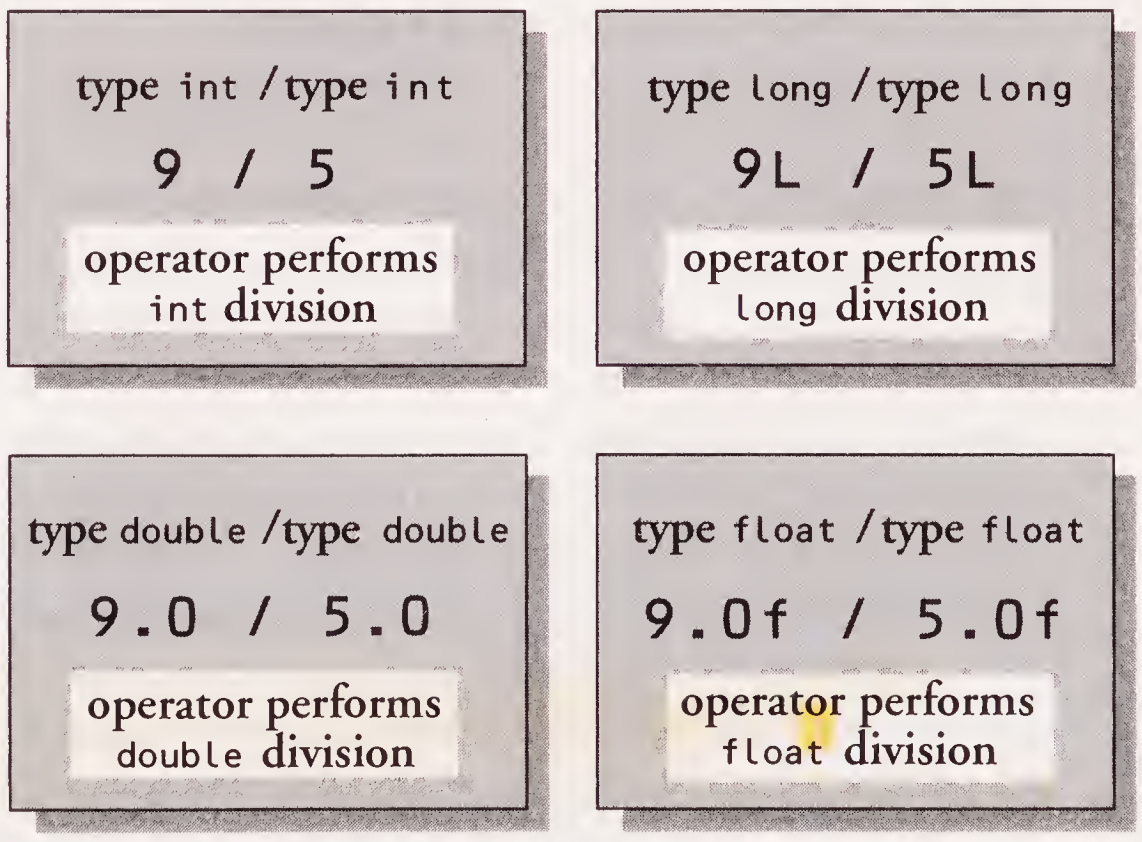


Figure 3-4 *Different divisions*

TYPE CONVERSIONS

C++'s profusion of types lets you match the type to the need. It also complicates life for the computer. For instance, adding two `short` values may involve different hardware instructions than adding two `long` values. With four integer types (counting `char`) and three floating-point types, the computer can have a lot of different cases to handle, especially if you start mixing types. To help deal with this potential mishmash, C++ makes many type conversions automatically:

- C++ converts values when you assign a value of one type to a variable of another type.
- C++ converts values when you combine mixed types in expressions.
- C++ converts values when you pass arguments to functions.

If you don't understand what happens in these automatic conversions, you may find some program results baffling, so we'll outline the rules. C++ also lets you force type conversions, and we'll look at that process, too.

CONVERSION ON ASSIGNMENT

C++ is fairly liberal in allowing you to assign a numeric value of one type to a variable of another type. Whenever you do so, the value is converted to the type of the receiving variable. For instance, suppose *so_long* is type `long`, that *thirty* is type `short`, and you have the following statement in a program:

```
so_long = thirty;        // assigning a short to a long
```

The program will take the value of *thirty* (typically a two-byte value) and expand it to a *long* value (typically a four-byte value) upon making this assignment.

Assigning a value to a type with a greater range usually poses no problem. For instance, assigning a `short` value to a `long` variable doesn't change the value; it just gives the value a few more bytes in which to laze about. However, assigning a large `long` value like 2111222333 to a `float` variable will result in losing some precision. Because `float` may have just six significant figures, the value may be rounded to 2.11122E9. Table 3-3 points out some possible conversion problems.

Assigning floating-point values to integer types poses a couple of problems. First, converting floating-point to integer results in truncating the number (discarding the fractional part). Second, a *float* value may be too big to fit in a cramped `int` variable. In that case, C++ doesn't define what the result should be; that means different implementations can respond differently. Listing 3-12 shows what happens in Turbo C++.

Table 3-3 Potential Conversion Problems

Conversion	Potential Problems
bigger floating-point type to smaller floating-point type, such as `double` to `float`	loss of precision (significant figures) original value may be out of range for target type, in which case result is undefined
floating-point type to integer type	loss of fractional part value may be out of range for target type, in which case result is undefined
bigger integer type to smaller integer type, such as `long` to `short`	original value may be out of range for target type, typically just the low-order bytes are copied

LISTING 3-12 assign.cpp

```
// assign.cpp -- type changes on assignment
#include <iostream.h>
int main(void)
{
    float tree = 3;       // int converted to float
    int guess = 3.9832;   // float converted to int
    int debt = 3.0E12;    // result not defined in C++
    cout << "tree = " << tree << "\n";
    cout << "guess= " << guess << "\n";
    cout << "debt = " << debt << "\n";
    return 0;
}
```

Here is the output:

```
tree = 3
guess= 3
debt = 0
```

Here *tree* is assigned the floating-point value *3.0*. However, because `cout` drops trailing zeros on output, it displays *3.0* as *3*. Assigning *3.9832* to the `int` variable *guess* causes the value to be truncated to *3*; C++ uses truncation (discarding the fractional part) and not rounding (finding the closest integer value) when converting floating-point types to integer types. Finally, note that the `int` variable *debt* is unable to hold the value *3.0E12*. This creates a situation in which C++ doesn't define the result. On our system, *debt* wound up with the value *0*. Well, that's a novel way to solve massive indebtedness!

Conversions in Expressions

Next, let's consider what happens when you combine two different numeric types in one expression. C++ makes two kinds of automatic conversions in that case. First, some types are converted automatically whenever they occur. Second, some types are converted when they are combined with other types in an expression.

First, let's examine the automatic conversions. When evaluating expressions, C++ converts `char`, `unsigned char`, `signed char`, and `short` values to `int`. This is called *integral promotion*. For instance, consider the following fowl statements:

```
short chickens = 20;                    // line 1
short ducks = 35;                       // line 2
short fowl = chickens + ducks;          // line 3
```

To execute the statement on line 3, a C++ program takes the values of *chickens* and *ducks* and converts both to `int`. Then the program converts the result back to type `short` because the answer is assigned to a type `short` variable. You may find this a bit roundabout, but it does make sense. The `int` type generally is chosen to be the computer's most natural type, which means the computer probably does calculations fastest for that type.

There's one more integral promotion: `unsigned short` type is converted to `int` if `short` is smaller than `int`. If the two types are the same size, `unsigned short` is converted to `unsigned int`. This rule ensures that there's no data loss in promoting `unsigned short`.

Then there are the conversions that take place when you combine different types arithmetically, such as adding an `int` to a `float`. When an operation involves two types, the smaller is converted to the larger. For instance, the program in Listing 3-10 divided 9.0 by 5. Because 9.0 is type `double`, the program converted 5 to type `double` before doing the division. More generally, the compiler goes through a checklist to determine which conversions to make in an arithmetic expression. Here's the list—the compiler goes through it in order:

1. If either operand is type `long double`, the other operand is converted to `long double`.
2. Otherwise, if either operand is `double`, the other operand is converted to `double`.
3. Otherwise, if either operand is `float`, the other operand is converted to `float`.
4. Otherwise the operands are integer types and the integral promotions are made.
5. In that case, if either operand is `unsigned long`, then the other operand is converted to `unsigned long`.
6. Otherwise, if one operand is `long int` and the other is `unsigned int`, the

conversion depends on the relative sizes of the two types. If `long` can represent possible `unsigned int` values, `unsigned int` is converted to `long`.

7. Otherwise, both operands are converted to `unsigned long`.
8. Otherwise, if either operand is `long`, the other is converted to `long`.
9. Otherwise, if either operand is `unsigned int`, the other is converted to `unsigned int`.
10. If the compiler reaches this point in the list, both operands should be `int`.

ANSI C follows the same rules as C++, but classic K&R C had slightly different rules. For instance, classic C always promoted `float` to `double` even if both operands were `float`.

Conversions in Passing Arguments

Normally, C++ function prototyping controls type conversions for argument passing, and we'll discuss that in Chapter 7. However, it is possible, although usually unwise, to waive prototype control for argument passing. In that case, C++ applies the integral promotions to the `char` and `short` types (signed and unsigned). Also, to preserve compatibility with huge amounts of code in classic C, C++ promotes `float` arguments to `double` when passing them to a function.

Type Casts

C++ also empowers you to force type conversions explicitly via the *type cast* mechanism. (C++ is such a masterful language.) The type cast comes in two forms. For example, to convert an `int` value stored in a variable called *thorn* to type `long`, you can use either of the following expressions:

```
(long) thorn     // makes the value of thorn type long
long (thorn)   // makes the value of thorn type long
```

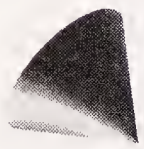
More generally, you can do the following:

```
(typename) value  // converts value to typename type
typename (value)  // converts value to typename type
```

The first form is straight C. The second form is pure C++. The idea behind the new form is to make a type cast look like a function call. This makes type casts for the built-in types look like the type conversions you can design for user-defined classes. Listing 3-13 gives a short illustration of both forms. Imagine that the first section of this listing is part of a powerful ecological modeling program that does floating-point calcu-



lations that get converted to integral numbers of birds and animals. The results you get depend on when you convert. The calculation for *auks* first adds the floating-point values, then converts the sum to `int` upon assignment. But the calculations for *bats* and *coots* first use type casts to convert the floating-point values to `int` then sum the values. The final part of the program shows how you can use a type cast to display the ASCII code for a type `char` value.



LISTING 3-13 typecast.cpp

```
// typecast.cpp -- forcing type changes
#include <iostream.h>
int main(void)
{
    int auks, bats, coots;

    // the following statement adds the values as double,
    // then converts the result to int
    auks = 19.99 + 11.99;

    // these statements add values as int
    bats = (int) 19.99 + (int) 11.99;  // old C syntax
    coots = int (19.99) + int (11.99); // new C++ syntax
    cout << "auks = " << auks << ", bats = " << bats;
    cout << ", coots = " << coots << "\n";

    char ch = 'Z';
    cout << "The code for " << ch << " is ";    // print as char
    cout << int(ch) << "\n";                    // print as int
    return 0;
}
```

Here is the result:

```
auks = 31, bats = 30, coots = 30
The code for Z is 90
```

First, adding 19.99 to 11.99 yields 31.98. When this value is assigned to the `int` variable *auks*, it's truncated to 31. But using type casts truncates the same two values to 19 and 11 before addition, making the *bats* result and the *coots* result both 30. The final `cout` statement uses a type cast to convert a type `char` value to `int` before displaying the result. This causes `cout` to print the value as an integer rather than as a character.

The program illustrates two reasons for using type casting. First, you may have values that are stored as type `double` but are used to calculate a type `int` value. For in-

stance, you may be fitting a position to a grid or modeling integer values, such as populations, with floating-point numbers. You may desire that calculations treat the values as `int`. Type casting lets you do so directly. Notice that you get a different result, at least for these values, when you convert to `int` and add than you do when you add first, then convert to `int`.

The second part of the program shows the most common reason for using a type cast—compelling data in one form to meet a different expectation. In this listing, for instance, the `char` variable *ch* holds the code for the letter Z. Using `cout` with *ch* displays the character Z, because `cout` zeros in on the fact that *ch* is type `char`. But by type casting *ch* to type `int` we get `cout` to shift to `int` mode and print the ASCII code stored in *ch*.

SUMMARY

C++'s basic types fall into two groups. One group consists of values that are stored as integers. The second group consists of values that are stored in a floating-point format. The integer types differ from each other in the amount of memory used to store values and in whether they are signed or unsigned. From smallest to largest, the integer types are `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, and `unsigned long`. C++ guarantees that *char* is large enough to hold any member of the system's character set, that `short` is at least two bytes, that `int` is at least as big as `short`, and that `long` is at least four bytes and at least as large as `int`. The exact sizes depend on the implementation.

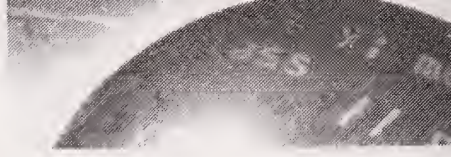
Characters are represented by their numeric codes. The I/O system determines whether a code is interpreted as a character or as a number.

The floating-point types can represent fractional values and values much larger than integers can. The three floating-point types are called `float`, `double`, and `long double`. C++ only guarantees that `float` is no larger than `double`, and that `double` is no larger than `long double`. Typically, however, `float` uses four bytes of memory, `double` uses eight bytes, and `long double` uses ten to sixteen bytes of memory.

By providing a variety of types in different sizes and in both signed and unsigned varieties, C++ lets you match the type to particular data requirements.

C++ uses operators to provide the usual arithmetic support for numeric types: addition, subtraction, multiplication, division, and taking the modulus. When two operators seek to operate on the same value, C++'s precedence and associativity rules determine which operation takes place first.

C++ converts values from one type to another when you assign values to a variable, when you mix types in arithmetic, and when you use type casts to force type conversions. Many type conversions are "safe," meaning they can be made with no loss or



alteration of data. For instance, you can convert an `int` value to a `long` value with no problems. Others, such as converting floating-point types to integer types, require more care.

At first, you may find the large number of basic C++ types a little excessive, particularly when you take the various conversion rules into account. But most likely you'll eventually find occasions when one of the types is just what you need at the time, and you'll thank C++ for having it.

Review Questions

1. Why does C++ have more than one integer type?
2. Define the following:
   a. A `short` integer with the value 80.
   b. An `unsigned int` integer with the value 42110.
   c. An integer with the value 3000000000.
3. What safeguards does C++ provide to keep you from exceeding the limits of an integer type?
4. What is the distinction between 33L and 33?
5. Consider the two C++ statements that follow. Are they equivalent?

```
char grade = 65;
char grade = 'A';
```

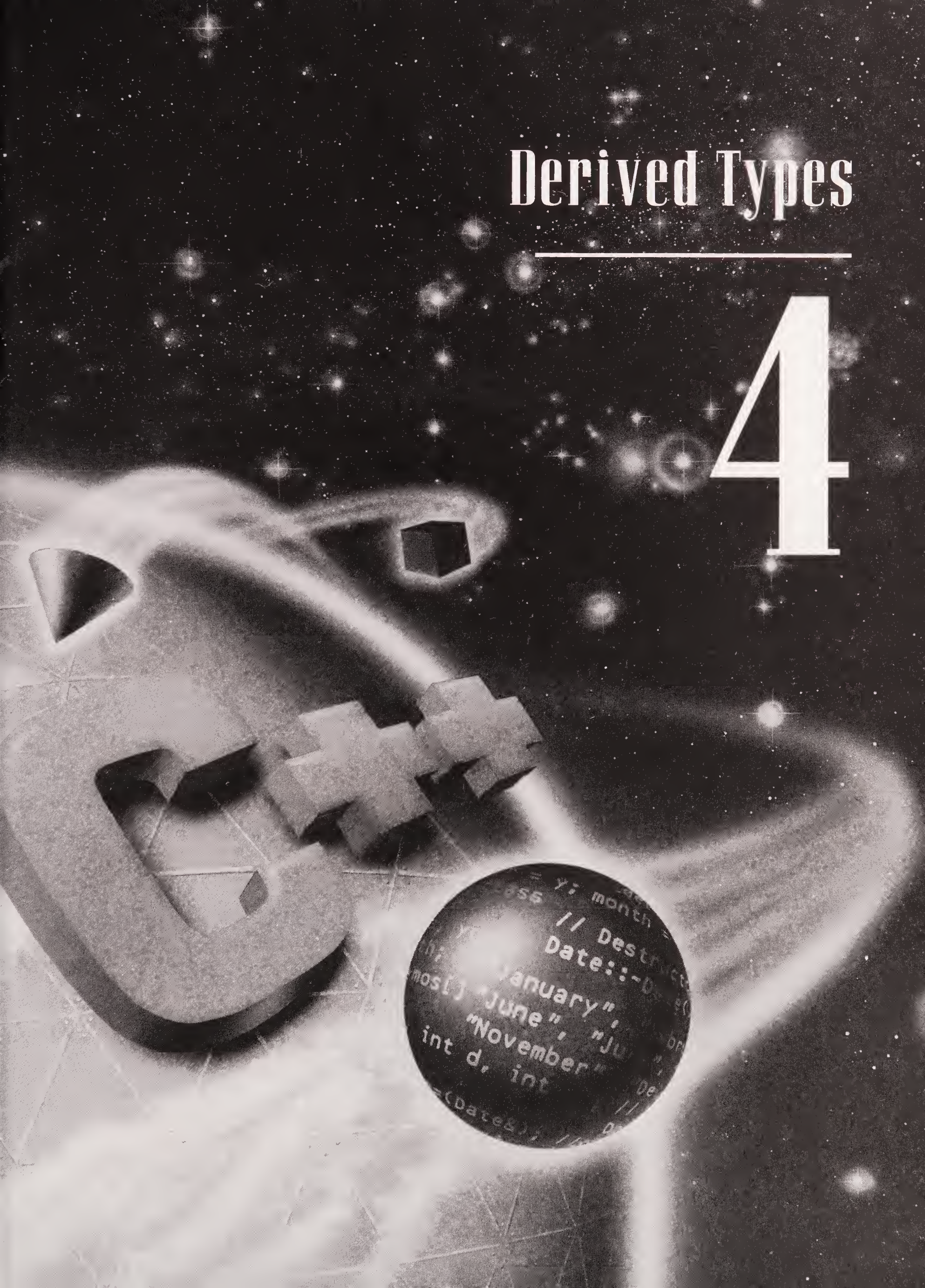
6. How could you use C++ to find out which character the code 88 represents? Come up with at least two ways.
7. Assigning a `long` value to a `float` can result in round-off error. What about assigning `long` to `double`?
8. Evaluate the following expressions as C++ would:
   a. 8 * 9 + 2
   b. 6 * 3 / 4
   c. 3 / 4 * 6
   d. 6.0 * 3 / 4
   e. 15 % 4

9. Suppose *x1* and *x2* are two type `double` variables that you want to add as integers and assign to an integer variable. Construct a C++ statement for doing so.

10. Write a short program that asks for your height in integer inches and then converts your height to feet and inches. Have the program use the underline character to indicate where to type the response. Also use a `const` type.

Derived Types

4



You've developed a computer game called User-Hostile in which one matches wits with a cryptic and abusive computer interface. Now you need to write a program that keeps track of your monthly game sales for a five-year period. Or you want to inventory your accumulation of hacker-hero trading cards. You'll soon conclude you need something more than C++'s simple basic types to meet these data requirements. And C++ offers something more—*derived* types. These are types built from the basic integer and floating-point types. The most far-reaching derived type is the class, that bastion of OOP towards which we are progressing. But C++ also supports several more modest derived types taken from C. The *array*, for example, can hold several values of the same type. A particular kind of array can hold a *string*, which is a series of characters. *Structures* can hold several values of differing types. Then there are *pointers*, which are variables that tell a computer where data is placed. We'll examine all these derived forms (excepting classes) in this chapter and also take a first look at *new* and *delete* and how they can be used to manage data.

Introducing Arrays

An *array* is a data form that can hold several values all of one type. For instance, an array can hold sixty type `int` values to represent five years of game sales data. Or an array can hold twelve `short` values representing the number of days in each month. Or an array can hold 365 `float` values indicating your food expenses for each day of the year. (That sure beats coming up with 365 separate variables!) Each value is stored in a separate array *element*, and the computer stores all the elements consecutively in memory.

To create an array, you use a declaration statement. An array declaration should indicate three things:

- The type of value to be stored in each element
- The name of the array
- The number of elements in the array

For example, the declaration

```
short months[12];    // creates array of 12 short
```

creates an array named *months* having twelve elements, each able to hold a type `short` value. Each element, in essence, is a variable that can be treated like a simple variable.

The general form for declaring an array is this:

```
typename arrayname[arraysize];
```

The expression *arraysize*, which is the number of elements, must be a constant, such as 10 or a `const` value, or a constant expression, such as `8 * sizeof (int)`, for which all values are known at the time compilation takes place. In particular, *arraysize* cannot be a variable whose value is set while the program is running. However, later this chapter will show you how to use the *new* operator to get around that restriction.

THE ARRAY AS DERIVED TYPE

An array is called a derived type because it is based on some other type. You can't simply declare that something is an array; it always has to be an array of some particular type. So there is no generalized array type. Instead, there are many specific array types, such as array of `char` or array of `long`. For example, consider this declaration:

```
float loans[20];
```

The type for *loans* is not "array"; rather, it is "array of `float`." This emphasizes that the *loans* array is derived from the `float` type.

Much of the usefulness of the array comes from the fact that you can access the array elements individually. The way to do so is to use a *subscript*, or *index*, to number the elements. C++ array numbering starts with 0. (This is nonnegotiable; you have to start at 0. Pascal and BASIC users will have to adjust.) C++ uses a bracket notation with the index to specify an array element. For example, *months[0]* is the first element of the *months* array, and *months[11]* is the last element. See Figure 4-1. Thus an array declaration lets you create a lot of variables with a single declaration, and you can then use an index to identify individual elements.

The yam analysis program in Listing 4-1 demonstrates a few properties of arrays, including declaring an array, assigning values to array elements, and initializing an array.

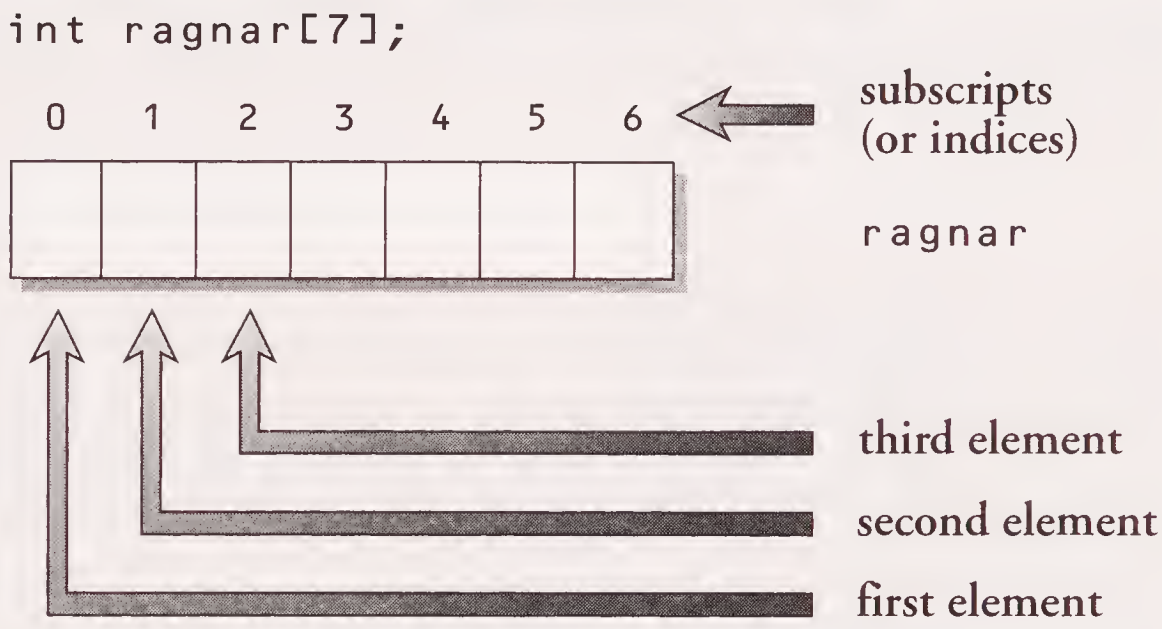


Figure 4-1 *Creating an array*

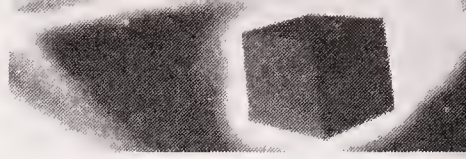
LISTING 4-1 arrayone.cpp

```
// arrayone.cpp -- small arrays of integers
#include <iostream.h>
int main(void)
{
    int yams[3];    // creates array with three elements
    yams[0] = 7;    // assign value to first element
    yams[1] = 8;
    yams[2] = 6;

    int yamcosts[3] = {20, 30, 5}; // create, initialize array
// NOTE:  If your C++ compiler or translator can't initialize
// this array, use static int yamcosts[3] instead of
// int yamcosts[3]

    cout << "Total yams = ";
    cout << yams[0] + yams[1] + yams[2] << "\n";
    cout << "The package with " << yams[1] << " yams costs ";
    cout << yamcosts[1] << " cents per yam.\n";
    int total = yams[0] * yamcosts[0] + yams[1] * yamcosts[1];
    total = total + yams[2] * yamcosts[2];
    cout << "The total yam expense is " << total << " cents.\n";

    cout << "\nSize of yams array = " << sizeof yams;
    cout << " bytes.\n";
```

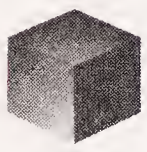


```
    cout << "Size of one element = " << sizeof yams[0];
    cout << " bytes.\n";
    return 0;
}
```

Here is the output:

```
Total yams = 21
The package with 8 yams costs 30 cents per yam.
The total yam expense is 410 cents.

Size of yams array = 6 bytes.
Size of one element = 2 bytes.
```



COMPATIBILITY NOTE

C++ 2.0 as well as ANSI C allow you to initialize ordinary arrays defined in a function. However, in some implementations that use a C++ translator instead of a true compiler, the C++ translator creates C code for a C compiler that is not fully ANSI C compliant. In that case you can get an error message like the following example from a Sun C++ 2.0 system:

```
arrayone.cc", line 10: sorry, not implemented: initialization
of yamcosts (automatic aggregate) Compilation failed
```

The fix is to use the keyword `static` in the array declaration:

```
// pre-ANSI initialization
static int yamcosts[3] = {20, 30, 5};
```

The keyword `static` causes the compiler to use a different memory scheme for storing the array, one that allows initialization even under pre-ANSI C. Chapter 8 discusses `static` in the section about storage classes. The program listings in this book will remind you when `static` may be needed.

PROGRAM NOTES

First, the program creates a three-element array called *yams*. Because *yams* has three elements, the elements are numbered from 0 to 2, and `arrayone.cpp` uses index values of 0–2 to assign values to the three individual elements. Each individual *yam* element is an `int` with all the rights and privileges of an `int` type. So `arrayone.cpp` can, and does, assign values to elements, add elements, multiply elements, and display elements.

The program uses the long way to assign values to the *yam* elements. C++ also lets you initialize array elements within the declaration statement. Listing 4-1 uses this shortcut to assign values to the *yamcosts* array:

```
int yamcosts[3] = {20, 30, 5};
```

Simply provide a comma-separated list of values (the *initialization list*) enclosed in braces. The spaces in the list are optional. If you don't initialize an array that's defined inside a function, the element values remain undefined. That means the element takes on whatever value previously resided at that location in memory.

Next, the program uses the array values in a few calculations. This part of the program looks cluttered with all the subscripts and brackets. The `for` loop, coming up in Chapter 5, provides a powerful way for dealing with arrays and eliminates the need to write each element explicitly. Meanwhile, we'll stick to small arrays.

The `sizeof` operator, you recall, returns the size, in bytes, of a type or data object. Note that if you use the `sizeof` operator with an array *name*, you get the number of bytes in the whole array. But if you use `sizeof` with an array *element*, you get the size, in bytes, of the element. This illustrates that *yams* is an array, but *yams[1]* is just an `int`.

More on Array Initialization

C++ has several rules about initializing an array. They restrict when you can do it, and they determine what happens if the number of array elements doesn't match the number of values in the initializer. Let's examine these rules.

You can use the initialization form only when defining the array. You cannot use it later, and you cannot assign one array wholesale to another:

```
int cards[4] = {3, 6, 8, 10};          // okay
int hand[4];                           // okay

hand[4] = {5, 6, 7, 9};                // not allowed
hand = cards;                          // not allowed
```

However, you can always use subscripts and assign values to the elements of an array individually.

You can provide fewer values than array elements. In that case, just the first part of the array is initialized. For example, the following statement initializes only the first two elements of *hoteltips*:

```
float hoteltips[5] = {5.0, 2.5};
```

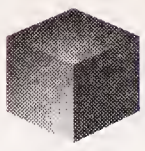
For arrays defined inside a function, the remaining elements remain undefined.

If you leave the square brackets empty when you initialize an array, the C++ com-

piler will count the elements for you. Suppose, for instance, you make this declaration:

```
short things[] = {1, 5, 3, 8};
```

The compiler will make *things* an array of four elements.



LETTING THE COMPILER DO IT

Normally, letting the compiler count the number of elements is poor practice, for its count may be different from what you think it is. However, this approach may be safer for initializing a character array to a string, as you'll see soon. And if your main concern is that the program, not you, know how large an array is, you can do something like this:

```
short things[] = {1, 5, 3, 8};
int num_elements = sizeof things / sizeof (short);
```

Whether this is useful or lazy depends on the circumstances.

STRINGS

A string is a series of characters stored in consecutive bytes of memory. This implies that you can store a string in an array of *char*, with each character kept in its own array element. Strings provide a convenient way to store text information such as messages to the user (*"Please tell me your secret Swiss bank account number:"*) or responses from the user (*"You must be joking"*). C++ strings, like C strings, have a special feature: the last character of every C++ string is the *null* character. This character, written `\0`, is the character with ASCII code 0, and it serves to mark the string's end. For instance, consider the following two declarations:

```
char dog [5] = { 'b', 'e', 'a', 'u', 'x'};  // not a string!
char cat[5] = {'f', 'a', 't', 's', '\0'};   // a string!
```

Both arrays are arrays of `char`, but only the second is a string. The null character plays a fundamental role in C++ strings. For example, C++ has many functions that handle strings, including those used by `cout`. They all work by processing a string character-by-character until they reach the null character. If you ask `cout` to display a nice string like *cat* above, it displays the first four characters, detects the null character, and stops. But if you are ungracious enough to tell `cout` to display the *dog* array above, which is not a string, `cout` would print the five letters in the array, then keep marching through memory byte by byte, interpreting each byte as a character to print, until it reached a null character. Since null characters, which really are bytes set to zero, tend to be com-

mon in memory, the damage usually is contained quickly; nonetheless, you should not treat nonstring character arrays as strings.

The *cat* array example makes initializing an array to a string look tedious—all those single quotes and then having to remember the null character. Don't worry. There is a better way to initialize a character array to a string. Just use a quoted string, called a *string constant* or *string literal*, as in the following:

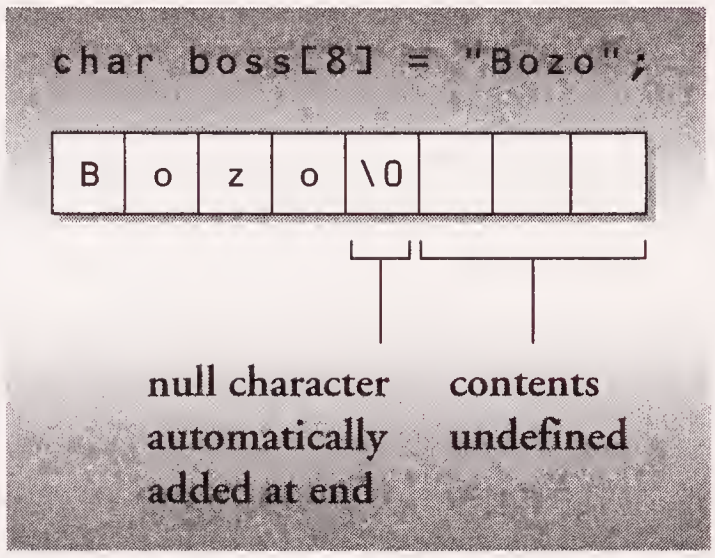
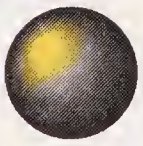


Figure 4-2 *Initializing an array to a string*

```
char bird[10] = "Mr. Cheep";
// the \0 is understood
char fish[] = "Bubbles";       // let the compiler count
```

Quoted strings always include the terminating null character implicitly, so you don't have to spell it out. See Figure 4-2. Also, the various C++ input facilities for reading a string from keyboard input into a `char` array automatically add the terminating null character for you. (If, when running the program in Listing 4-1, you discovered you had to use the keyword `static` to initialize an array, you'll have to use it with these `char` arrays, too.)

Of course, you should make sure the array is large enough to hold all the characters of the string, including the null character. Initializing a character array with a string constant is one case where it may be safer to let the compiler count the number of elements for you. There is no harm, other than wasted space, in making an array larger than the string. That's because functions that work with strings are guided by the location of the null character, not by the size of the array. C++ imposes no limits on the length of a string.



When determining the minimum array size needed to hold a string, remember to include the terminating null character in your count.

Note that a string constant (double quotes) is not interchangeable with a character constant (single quotes). A character constant such as 'S' is a shorthand notation for the code for a character. So, on an ASCII system, '*S*' is just another way of writing 83. Thus the statement

```
char shirt_size = 'S';          // this is fine
```

assigns the value 83 to *shirt_size*. But *"S"* represents the string consisting of two characters, the *S* and the *\0* characters. Even worse, *"S"* actually represents the memory address at which the string is stored. So a statement like

```
char shirt_size = "S";          // illegal type mismatch
```

attempts to assign a memory address to *shirt_size*! Because an address is a separate type in C++, a C++ compiler won't allow this sort of nonsense. (We'll return to this point later, after we've discussed pointers.) But C, which is more lenient about checking type agreement, will let this statement pass with a warning, and the result will be garbage.

STRING CONCATENATION

Sometimes a string may be too long to conveniently fit on one line of code. C++ lets you *concatenate* string constants, that is, combine two quoted strings into one. Indeed, any two string constants separated by white space only (spaces, tabs, and newlines) automatically are joined into one. Thus all the following output statements are equivalent to each other:

```
cout << "I'd give my right arm to be"  " a great violinist.\n";
cout << "I'd give my right arm to be a great violinist.\n";
cout << "I'd give my right ar"
        "m to be a great violinist.\n";
```

Note that the join doesn't add any spaces to the joined strings. The first character of the second string immediately follows the last character, not counting *\0*, of the first string. The *\0* character from the first string is replaced by the first character of the second string.

USING STRINGS IN AN ARRAY

The two most common ways of getting a string into an array are to initialize an array to a string constant and to read keyboard or file input into an array. Listing 4-2 demonstrates these approaches by initializing one array to a quoted string and using `cin` to place an input string in a second array. The program also uses the standard library function `strlen()` to get the length of a string. The standard `string.h` header file provides declarations for this and many other string-related functions.

LISTING 4-2 strings.cpp

```
// strings.cpp -- storing strings in an array
#include <iostream.h>
#include <string.h>     // for the strlen() function
int main(void)
{
    const int Size = 15;
    char name1[Size];                    // empty array
    char name2[Size] = "C++owboy";       // initialized array
// NOTE: some implementations may require the static keyword
// to initialize the array name2

    cout << "Howdy! I'm " << name2;
    cout << "! What's your name?\n";
    cin >> name1;
    cout << "Well, " << name1 << ", your name has ";
    cout << strlen(name1) << " letters and is stored\n";
    cout << "in an array of " << sizeof name1 << " bytes.\n";
    cout << "Your initial is " << name1[0] << ".\n";
    name2[3] = '\0';                     // null character
    cout << "Here are the first 3 characters of my name: ";
    cout << name2 << "\n";
    return 0;
}
```

Here is a sample run:

```
Howdy! I'm C++owboy! What's your name?
Basicman
Well, Basicman, your name has 8 letters and is stored
in an array of 15 bytes.
Your initial is B.
Here are the first 3 characters of my name: C++
```

PROGRAM NOTES

What can you learn from this example? First, note that the `sizeof` operator gives the size of the entire array—fifteen bytes. But the `strlen()` function returns the size of the string stored in the array and not the size of the array itself. Also, `strlen()` counts just the visible characters and not the null character. Thus it returns a value of 8, not 9, for the length of *Basicman*. So if *cosmic* is a string, the minimum array size for holding that string is *strlen(cosmic) + 1*.

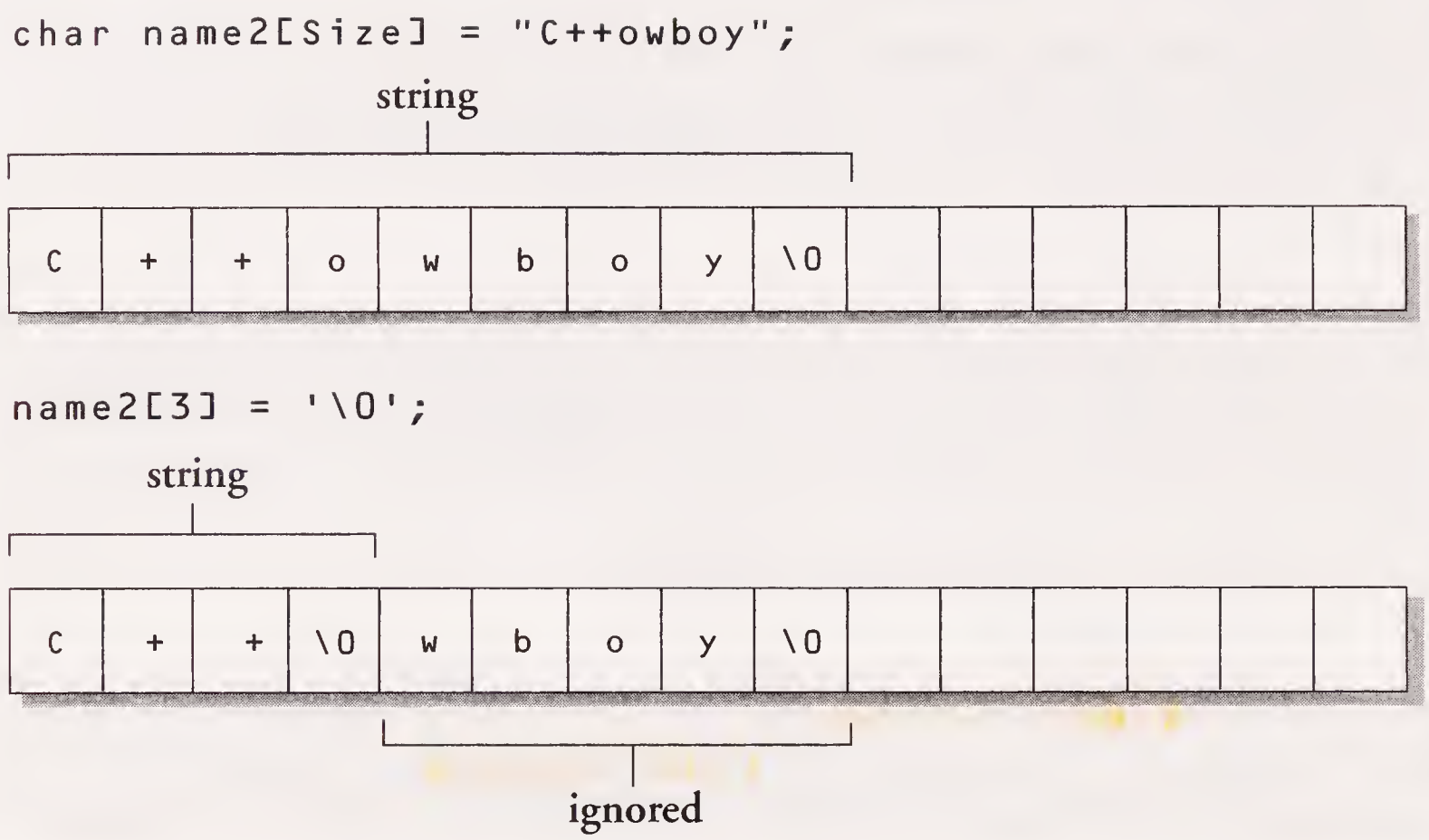


Figure 4-3 *Shortening a string with* `\0`

Because *name1* and *name2* are arrays, you can use an index to access individual characters in the array. For instance, the program uses *name1[0]* to find the first character in that array. Also, the program sets *name2[3]* to the null character. That makes the string end after three characters even though more characters remain in the array. See Figure 4-3.

ADVENTURES IN STRING INPUT

The `strings.cpp` program has a blemish that was concealed through the often useful technique of carefully selected sample input. Listing 4-3 removes the veils and shows that string input can be tricky.

LISTING 4-3 instr1.cpp

```
// instr1.cpp -- reading more than one string
#include <iostream.h>
const int Size = 20;
int main(void)
{
```

```
    char name[Size];
    char dessert[Size];

    cout << "Enter your name:\n";
    cin >> name;
    cout << "Enter your favorite dessert:\n";
    cin >> dessert;
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

The intent is simple: read a user's name and favorite dessert from the keyboard, then display the information. Here is a sample run:

```
Enter your name:
Alistair Dreeb
Enter your favorite dessert:
I have some delicious Dreeb for you, Alistair.
```

We didn't even get a chance to respond to the dessert prompt! The program showed it, then immediately moved on to display the final line.

The problem lies with how `cin` determines when you've finished entering a string. You can't enter the null character from the keyboard, so `cin` needs some other means for locating the end of a string. The `cin` technique is to use white space—spaces, tabs, and newlines—to delineate a string. This means `cin` reads just one word when getting input for a character array. Once it reads this word, `cin` automatically adds the terminating null character when placing the string into the array.

The practical result in our example is that `cin` reads *Alistair* as the entire first string and puts it into the *name* array. This leaves poor *Dreeb* still sitting in the input queue. When `cin` searches the input queue for the response to the favorite dessert question, it finds *Dreeb* still there. Then `cin` gobbles up *Dreeb* and puts it into the *dessert* array. See Figure 4-4.

Another problem, which didn't surface in the sample run, is that the input string might turn out to be longer than the destination array. Using `cin` as we did offers no protection against placing a 30-character string in a 20-character array.

Many programs depend on string input, so it's worth our while to explore this topic further. We'll have to draw upon some of the more advanced features of `cin`, as described in Chapter 12. As you'll see, the path to smooth input handling has a few more pitfalls to bypass. (We could jump directly to the final solution, but you wouldn't learn as much that way.)

LINE-ORIENTED INPUT: GETLINE() AND GET()

To be able to enter whole phrases instead of single words as a string, you need a different approach to string input. Specifically, you need a line-oriented method instead of a word-oriented method. You are in luck, for the *istream* class, of which `cin` is an example, has some line-oriented class member functions. The `getline()` function, for example, reads a whole line, using the newline character transmitted by the Return or Enter key to mark the end of input. You invoke this method by using `cin.getline()` as a function call. The function takes two arguments. The first argument is the name of the array destined to hold the line of input, and the second argument is a limit on the number of characters to be read. If this limit is, say, 20, the function will read no more than 19 characters, leaving room to automatically add the null character at the end. The `getline()` member function stops reading input when it reaches this numeric limit or when it reads a newline, whichever comes first. For example, suppose you want to use `getline()` to read a name into the 20-element array *name*. You would use this call:

```
cin.getline(name,20);
```

This would read the entire line into the *name* array, providing that the line consisted of 19 or fewer characters. (The `getline()` member function also has an optional third argument, which we leave to Chapter 12 to discuss.)

Listing 4-4 modifies Listing 4-3 to use `cin.getline()` instead of a simple `cin`. Otherwise, the program is unchanged.

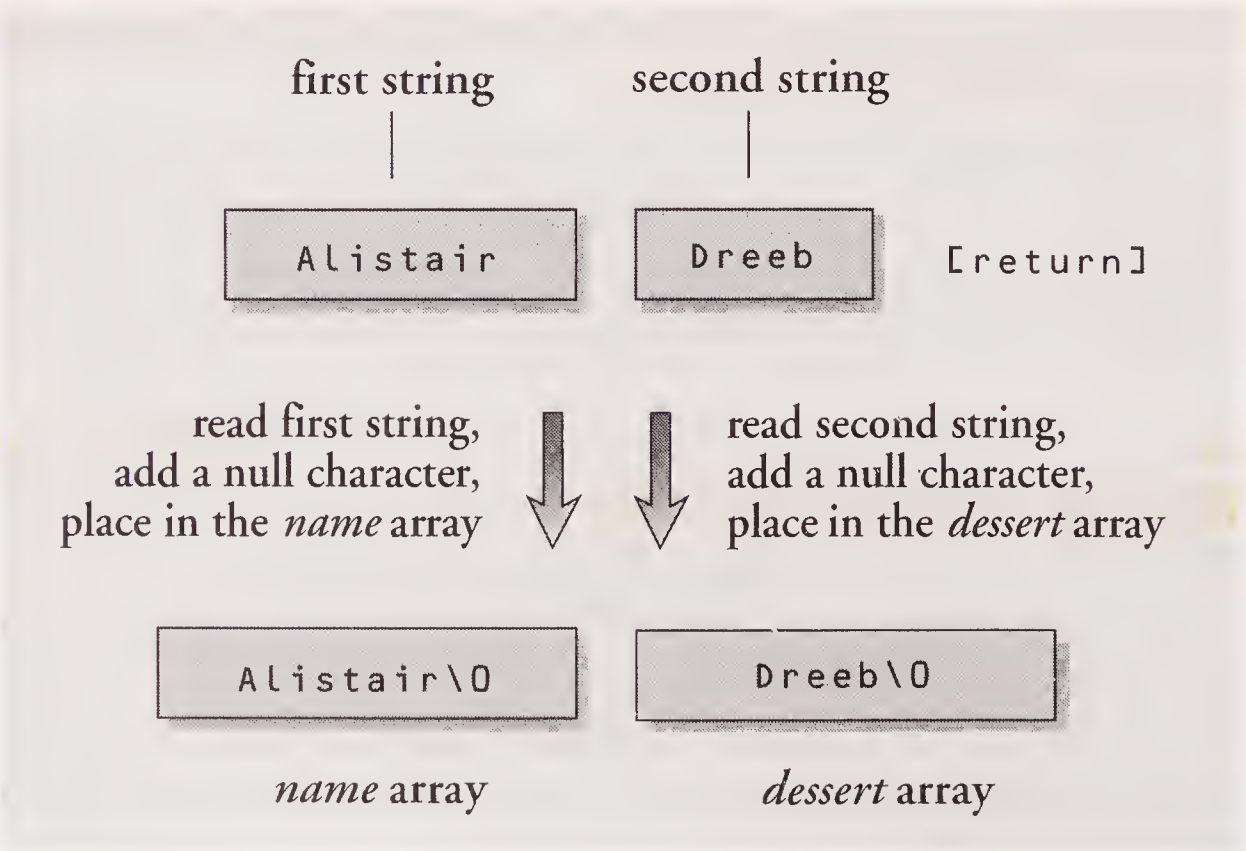
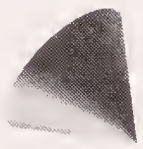


Figure 4-4 *The* `cin` *view of string input*



LISTING 4-4 instr2.cpp

```
// instr2.cpp -- reading more than one word with getline
#include <iostream.h>
int main(void)
{
    const int Size = 20;
    char name[Size];
    char dessert[Size];

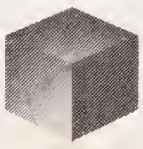
// Zortech and Gnu users please check the Compatibility Note.

    cout << "Enter your name:\n";
    cin.getline(name, Size);      // reads through newline
    cout << "Enter your favorite dessert:\n";
    cin.getline(dessert, Size);
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Here is some sample output:

```
Enter your name:
Melanie Ploops
Enter your favorite dessert:
Raspberry Torte
I have some delicious Raspberry Torte for you, Melanie Ploops
.
```

The program now reads complete names and delivers the user her just desserts! The `getline()` function conveniently gets a line at a time. It reads input through the newline character marking the end of the line, but it doesn't save the newline character. Instead, it replaces it with a null character when storing the string. See Figure 4-5.



COMPATIBILITY NOTE

At the time of this writing, some C++ versions, including Zortech C++ and Gnu G++, haven't fully implemented all facets of the C++ 2.0's revised I/O package. In particular, the `getline()` member function isn't available. If this affects you, just read about this example and go on to the next one, which uses a member function that Zortech and Gnu have implemented. Also, early releases of Turbo C++ implemented `getline()` slightly differently so that it did store the newline character in the string.

Since not everyone has `getline()` available, let's try another approach. The *istream* class has another member function, called `get()`, that comes in several variations. One variant works much like `getline()`. It takes the same arguments, interprets them the same way, and reads to the end of a line. But it doesn't read the newline. In Listing 4-5, we replace `getline()` with `get()`.

LISTING 4-5 instrr3.cpp

```
// instr3.cpp -- reading more than one word with get()
#include <iostream.h>
int main(void)
{
    const int Size = 20;
    char name[Size];
    char dessert[Size];

    cout << "Enter your name:\n";
    cin.get(name, Size);              // reads to newline
    cout << "Enter your favorite dessert:\n";
    cin.get(dessert, Size);
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Here is a sample run:

```
Enter your name:
Sparky Bandersnoot
Enter your favorite dessert:
I have some delicious  for you, Sparky Bandersnoot.
```

Darn! The program gets the name correctly, but it seems to have lost the dessert. The problem is that `cin.get()` reads up to, but not including, the newline character. After the first call to `cin.get()`, the newline character is still in the input queue. The second call to `cin.get()` then sees this lone newline as marking an *empty* line, so it reads the *null string* into the *dessert* array. The null string is a string consisting of one character—the null character. So poor Sparky winds up with the null dessert.

At this point, we seem to have taken a minor step backwards in our tasteful efforts at dessert distribution, although it's quite educational to see how `cin.get()` differs from its `cin.getline()` sibling. As you have seen, `cin.getline()` reads a line, in-

Code:

```
char name[10];
cout << "Enter your name:\n";
cin.getline(name, 10);
```

User responds by typing Jud, then the Return key.

```
Enter your name: Jud (ENTER)
```

`cin.getline()` responds by reading `Jud`, reading the newline generated by the Return key, and replacing it with a null character.



newline replaced with null character

Figure 4-5 `cin.get()` *includes the newline*

cluding the newline, which it discards, while `cin.get()` reads just up to the newline, which it leaves in the input queue for the next input operation. Nonetheless, you can use `cin.get()` to get past the pesty newline by dipping further into the *istream*'s bag of member functions (discussed fully in Chapter 12). The way to success is to use another variant of `cin.get()`. If you use this function with *no* arguments, it reads the next character in the input queue and discards it. This is exactly what we want now. We can use `cin.get` *(name, Size)* to read all the first line of input except for the newline character. Then we can use `cin.get()` to read and discard the newline character. That will cause the call to `cin.get` *(dessert, Size)* to begin at the start of the next line.

There are two ways to add the argument-free `cin.get()` call to the program. The first is as a separate statement:

```
cin.get(name, Size);    // read line
cin.get();              // read, discard next character
```

The second is to *concatenate*, or join, the two class member functions as follows:

```
cin.get(name, Size).get();  // concatenate member functions
```

This is something you can do with C++ member functions, provided they are properly designed. The *istream* member functions are properly designed, so you can use this

technique with the `get()` and `cin.getline()` member functions. For example, the statement

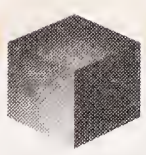
```
cin.getline(name1, Size).getline(name2, Size);
```

will read two consecutive input lines into the arrays *name1* and *name2*; it's equivalent to making two separate calls to `cin.getline()`. We'll use concatenation in Listing 4-6. And when we develop classes, we'll show you how to incorporate this feature into your class definitions.

LISTING 4-6 instr4.cpp

```
// instr4.cpp -- reading more than one word with get & get()
#include <iostream.h>
int main(void)
{
    const int Size = 20;
    char name[Size];
    char dessert[Size];

    cout << "Enter your name:\n";
// NOTE: Zortech users please see Compatibility Note
    cin.get(name, Size).get();          // read string, newline
    cout << "Enter your favorite dessert:\n";
    cin.get(dessert, Size);
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```



COMPATIBILITY NOTE

As of this writing, some C++ versions, including Zortech C++, haven't implemented the `get()` variant having no arguments. They do, however, implement yet another `get()` variant, one that takes a single `char` argument. To use it instead of the argumentless `get()`, you need to declare a `char` variable first:

```
int ch;
cin.get(name, Size).get(ch);
```

You can use this code instead of what is found in Listing 4-6. Chapters 5 and 12 further discuss the `get()` variants.

Here is a sample run:

```
Enter your name:
Mai Parfait
Enter your favorite dessert:
Chocolate Mousse
I have some delicious Chocolate Mousse for you, Mai Parfait.
```

Again, we have a program that delivers the goods!

C++ programs frequently use pointers instead of arrays to handle strings. We'll take up that aspect of strings after learning a bit about pointers. Meanwhile, let's take a look at another derived type, the structure.

Introducing Structures

Suppose you want to store information about a basketball player. You might want to store his or her name, salary, height, weight, scoring average, free-throw percentage, assists, and so on. You'd like some sort of data form that could hold all this information in one unit. An array won't do. Although an array can hold several items, each item has to be the same type. That is, one array can hold twenty `int`s and another can hold ten `float`s, but a single array can't store `int`s in some elements and `float`s in other elements.

The answer to your desire (the one about storing information about a basketball player) is the C++ *structure*. The structure is a more versatile data form than an array, for a single structure can hold several kinds of data. This enables you to unify your data representation by storing all the related basketball information in a single structure variable. If you want to keep track of a whole team, you can use an array of structures. The structure type also is a stepping-stone to that bulwark of C++ OOP, the class. So learning a little about structures now takes us that much closer to the OOP heart of C++.

A structure is a user-definable type, with a template serving to define the type's data properties. Once you define the type, you can create variables of that type. Thus creating a structure is a two-part process. First, you define a *structure template*. It describes and labels the different types of data that can be stored in a structure. Then you create a *structure variable*, or, more generally, a *structure data object*, that follows the template plan.

For example, suppose that Bloataire, Inc., wants to create a type to describe members of its product line of designer inflatables. In particular, the type should hold the name of the item, its volume in cubic feet, and its selling price. Here is a structure template meeting those needs:

```
struct inflatable      // structure template
{
    char name[20];
    float volume;
    double price;
};
```

The keyword `struct` indicates that the code defines a template for a structure. The identifier *inflatable* is the name, or *tag*, we'll use for this template; that makes *inflatable* the name for the new type. Thus you now can create variables of type *inflatable* just as you create variables of type `char` or `int`. Next, between braces, comes the list of data types to be held in the structure. Each list item is a declaration statement. You can use any of the C++ types here, including arrays and other structures. We've used an array of `char`, suitable for storing a string, and a `float` and a `double`. Each individual item in the list is called a structure *member*, so the *inflatable* structure has three members. See Figure 4-6.

Once you have the template, you can create variables of that type:

```
inflatable hat; // hat is a structure variable of type inflatable
inflatable woopie_cushion;    // type inflatable variable
inflatable mainframe;         // type inflatable variable
```

If you're familiar with C structures, you'll notice (probably with pleasure) that C++ allows you to drop the keyword *struct* when declaring structure variables:

```
struct inflatable goose;      // keyword struct required in C
inflatable vincent;           // keyword struct not required in C++
```

In C++, the structure tag is used just like a fundamental type name. This change emphasizes that a structure template defines a new type. It also removes omitting `struct` from the list of curse-inducing errors.

Given that *hat* is type *inflatable*, you use the membership operator (.) to access individual members. For instance, *hat.volume* refers to the *volume* member of the structure, and *hat.price* refers to the *price* member. Similarly, *vincent.price* is the *price* member of a *vincent* variable. Because the *price* member is declared as type `double`, *hat.price* and *vincent.price* both are equivalent to type `double` variables and can be used in any manner an ordinary type `double` variable can. So *hat* is a structure but *hat.price* is a `double`. By the way, the method we used to access class member functions like `cin.getline()` has its origins in the method used to access structure member variables like *vincent.price*.

Listing 4-7 illustrates these points about a structure. Also, it shows how to initialize one.

LISTING 4-7 structur.cpp

```
// structur.cpp -- a simple structure
#include <iostream.h>
struct inflatable      // structure template
{
    char name[20];
    float volume;
    double price;
};

int main(void)
{
    inflatable guest =
    {
            "Glorious Gloria",   // name value
            1.88,                // volume value
            29.99                // price value
    };      // guest is a structure variable of type inflatable
            // It's initialized to the indicated values
    inflatable pal =
    {
            "Audacious Arthur",
            3.12,
            32.99
};      // pal is a second variable of type inflatable
```

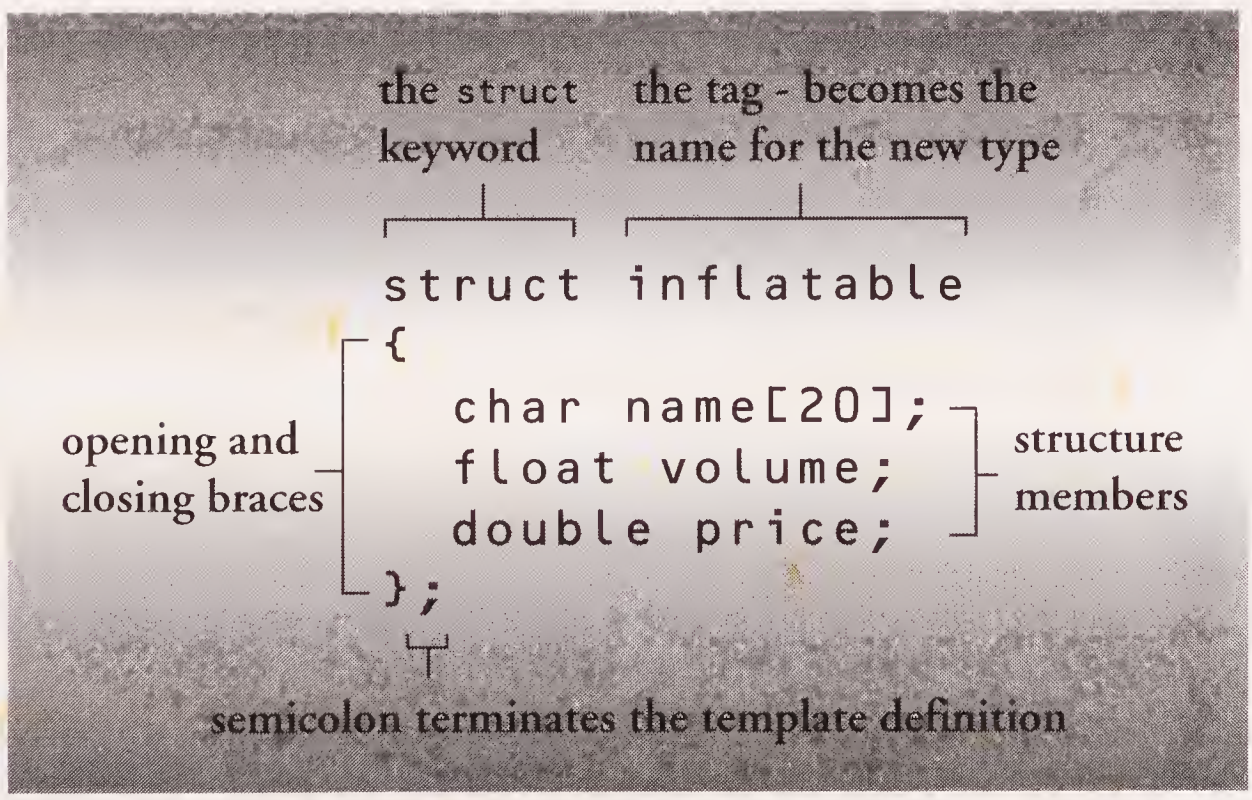
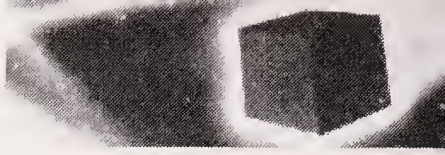
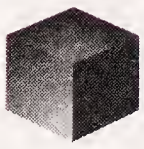


Figure 4-6 *Parts of a structure template*



```
// NOTE: some implementations require using
//  static inflatable guest =

    cout << "Expand your guest list with " << guest.name;
    cout << " and " << pal.name << "!\n";
          // pal.name is the name member of the pal variable
    cout << "You can have both for $";
    cout << guest.price + pal.price << "!\n";
    return 0;
}
```



COMPATIBILITY NOTE

Just as some implementations have not yet implemented the ability to initialize an ordinary array defined in a function, they also haven't implemented the ability to initialize an ordinary structure defined in a function. Again, the solution is to use the keyword `static` in the declaration.

Here is the output:

```
Expand your guest list with Glorious Gloria and Audacious Arthur!
You can have both for $62.98!
```

PROGRAM NOTES

One important matter is where to place the structure template. There are two choices for `structur.cpp`. We could have placed the template *inside* the `main()` function, just after the opening brace. The second choice, and the one we made, is to place the template *outside* of `main()` and preceding it. When a definition occurs outside of any function, it's called an *external* definition. For this program, there is no practical difference between the two choices. But for programs consisting of two or more functions, the difference can be crucial. The external definition can be used by all the functions following it, while the internal definition can be used only by the function in which the definition is found. Most often, you will want an external template definition so that all the functions can use structures of that type. See Figure 4-7.

Variables, too, can be defined internally or externally, with external variables being shared among functions. (Chapter 8 will look further into that topic.) C++ practices discourage using external variables but encourage using external templates.

Next, notice the initialization procedure:

```
inflatable guest =
{
    "Glorious Gloria",          // name value
    1.88,                       // volume value
    29.99                       // price value
};
```

external template—can be used in all functions in file

```
#include <iostream.h>
struct parts
{
  unsigned long part_number;
  float part_cost;
};
void mail(void);
int main(void)
{
```

local template—can be used only in this function

```
  struct perks
  {
    int key_number;
    char car[12];
  };
```

type *parts* variable

```
  parts chicken;
```

type *perks* variable

```
  perks mr_blug;
  ...
  ...
}
void mail(void)
{
```

type *parts* variable

```
  parts studebaker;
```

can't declare a type *perks* variable here

```
  ...
  ...
}
```

Figure 4-7 *Local and external templates*

As with arrays, you use a comma-separated list of values enclosed within a pair of braces. The program placed one value per line, but you can place them all on the same line. Just remember to separate items with a comma:

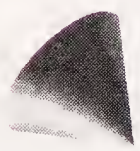
```
inflatable duck = { "Daffny", 0.12,  9.98};
```

Each member of the structure can be initialized to the appropriate kind of data. For instance, the *name* member is a character array, so we can initialize it to a string.

Each structure member is treated as a variable of that type. Thus *pal.price* is a `double` variable and *pal.name* is an array of `char`. And when the program uses `cout` to display *pal.name*, it displays the member as a string. By the way, because *pal.name* is a character array, we can use subscripts to access individual characters in the array. For example, *pal.name[0]* is the character *A*. But *pal[0]* is meaningless, since *pal* is a structure, not an array.

OTHER STRUCTURE PROPERTIES

C++ attempts to make using user-defined types as similar as possible to using built-in types. For example, you can pass structures as arguments to functions, and you can have a function use a structure as a return value. Also, you can assign one structure to another of the same type by using the assignment operator (=). Doing so causes each member of one structure to be set to the value of the corresponding member in the other structure, even if the member is an array. We'll defer passing and returning structures until we discuss functions in Chapter 7, but we can take a quick look at structure assignment now. Listing 4-8 provides an example.



LISTING 4-8 assgn_st.ccp

```
// assgn_st.cpp -- assigning structures
#include <iostream.h>
struct inflatable
{
    char name[20];
    float volume;
    double price;
};
int main(void)
{
    inflatable bouquet =
    {
            "sunflowers",
            0.20,
            12.49
    };
// NOTE: some implementations may require using
//  static inflatable bouquet =
```

```
    inflatable choice;
    cout << "bouquet: " << bouquet.name << " for $";
    cout << bouquet.price << "\n";

    choice = bouquet;    // assign one structure to another
    cout << "choice: " << choice.name << " for $";
    cout << choice.price << "\n";
    return 0;
}
```

Here's the output:

```
bouquet: sunflowers for $12.49
choice: sunflowers for $12.49
```

As you can see, the *choice* structure has its members assigned the same values that were stored in the *bouquet* structure. This is called *memberwise* assignment.

You can combine defining a template with creating structure variables. To do so, follow the closing brace with the variable name or names:

```
struct perks
{
    int key_number;
    char car[12];
} mr_smith, ms_jones;  // two perks variables
```

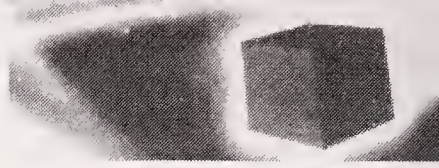
You can even initialize a variable you create in this fashion:

```
struct perks
{
    int key_number;
    char car[12];
} mr_glitz =
{
    7,              // value for mr_glitz.key_number member
    "Packard"       // value for mr_glitz.car member
};
```

However, keeping the template definition separate from the variable declarations usually makes a program easier to read and follow.

Another thing you can do with structures is create a structure with no type name. You do this by omitting a tag name while simultaneously defining a template and a variable:

```
struct              // no tag
{
    int x;          // 2 members
```



```
    int y;
} position;              // a structure variable
```

This creates one structure variable called *position.* You can access its members with the membership operator, as in *position.x*, but there is no general name for the type. So you can't subsequently create other variables of the same type. We won't be using this limited form of structure.

Aside from the fact a C++ program can use the structure tag as a type name, C structures have all the features we've discussed so far for C++ structures. But C++ structures go further. Unlike C structures, for instance, C++ structures can have member functions in addition to member variables. But these more advanced features most typically are used with classes rather than structures, so we'll discuss them when we cover classes. For now, let's continue our survey of derived types.

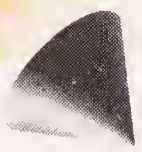
Pointers and the Free Store

At the beginning of Chapter 3 we mentioned three fundamental properties that a computer program must keep track of when storing data. To save the book the wear and tear of your thumbing back to that chapter, here are those properties again:

- Where the information is stored
- What value is kept there
- What kind of information is stored

You've used one strategy for accomplishing these ends: defining a simple variable. The declaration statement provides the type and a symbolic name for the value. It also causes the program to allocate memory for the value and to keep track of the location internally.

Let's look at a second strategy now, one that becomes particularly important in developing C++ classes. This strategy is based on *pointers*, which are variables that store addresses of values rather than the values themselves. But before discussing pointers, let's see how to find addresses explicitly for ordinary variables. Just apply the address operator, represented by &, to a variable to get its location. Listing 4-9 demonstrates this operator.



LISTING 4-9 address.cpp

```
// address.cpp -- using the & operator to find addresses
#include <iostream.h>
int main(void)
{
```

```
    int donuts = 6;
    double cups = 4.5;

    cout << "donuts value = " << donuts;
    cout << " and donuts address = " << &donuts << "\n";
// NOTE: you may need to use unsigned (&donuts)
//       and unsigned (&cups)
    cout << "cups value = " << cups;
    cout << " and cups address = " << &cups << "\n";
    return 0;
}
```

Here is the output:

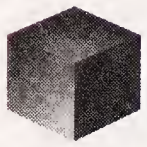
```
donuts value = 6 and donuts address = 0x8566fff4
cups value = 4.5 and cups address = 0x8566ffec
```


COMPATIBILITY NOTE

`cout` is a smart object, but some versions are smarter than others. Thus some implementations fail to recognize pointer types. For instance, at the time of this writing, Zortech C++ requires that you type cast the address to a recognizable type, such as `unsigned int`. The appropriate type cast depends on the memory model. The default DOS memory model uses a two-byte address, hence `unsigned int` is the proper cast. Some DOS memory models, however, use a four-byte address, which requires a cast to `unsigned long`.

When displaying addresses, `cout` uses hexadecimal notation because that is the usual notation used to describe memory. Our implementation stores *cups* at a lower memory location than *donuts*. The difference between the two addresses is 0x8566fff4 - 0x8566ffec, or 8. This makes sense, for *cups* is type `double`, which uses eight bytes. (In case you're interested, these particular representations of addresses reflect the PC method of describing an address by a segment value and by an offset. The segment value, 8*566* in this case, identifies the block of memory being used to store the data; it's the actual address divided by sixteen. The offsets, *ffec* and *fff4* in this case, represent memory position relative to the beginning of the segment. PC programs can use two-byte pointers holding just the offset if all the data is in one segment. Or they can use four-byte pointers, with the first two bytes holding the segment value and the second two bytes holding the offset. Turbo C++ displays the four-byte form even if a program internally uses the two-byte form.)

Using ordinary variables, then, treats the value as a named quantity and the location as a derived quantity. Now let's look at the pointer strategy, one that is essential to the C++ programming philosophy of memory management. (See the note on Pointers and the C++ Philosophy.)



POINTERS AND THE C++ PHILOSOPHY

Object-oriented programming differs from traditional procedural programming in OOP's emphasis on making decisions during *run time* instead of *compile time*. Run time means while a program is running, and compile time means when the compiler is putting a program together. A run-time decision is like, when on vacation, choosing what sights to see depending on the weather and on your mood at the moment, while a compile-time decision is more like adhering to a preset schedule regardless of the conditions. Run-time decisions provide the flexibility to adjust to current circumstances. For instance, consider allocating memory for an array. The traditional way is to declare an array. To declare an array in C++, you have to commit yourself to a particular array size. Thus the array size is set when the program is compiled, so it is a compile-time decision. Perhaps you think an array of 20 elements is sufficient 80% of the time, but that occasionally the program will need to handle 200 elements. So to be safe, you use an array with 200 elements. This results in your program wasting memory most of the times it's used. OOP tries to make a program more flexible by delaying such decisions until run time. That way, once the program is running, you can tell it you'll need only 20 elements one time or that you need 205 elements another time. In short, you make the array size a run-time decision. To make this approach possible, the language has to allow you to create an array—or equivalent—while the program runs. The C++ method, as you'll soon see, involves using the keyword `new` to request the correct amount of memory and using pointers to keep track of where the newly allocated memory is found.

The new strategy for handling stored data switches things around by treating the *location* as the named quantity and the *value* as a derived quantity. A special type of variable—the pointer—holds the address of a value. Thus the name of the pointer represents the location. Applying the * operator, called the *indirect value* or the *dereferencing* operator, yields the value at the location. (Yes, this is the same * symbol used for multiplication; C++ uses the context to determine whether you mean multiplication or dereferencing.) Suppose, for example, that *manly* is a pointer. Then *manly* represents an address, and **manly* represents the value at that address. The combination

**manly* becomes equivalent to an ordinary type `int` variable. Listing 4-10 demonstrates these points. It also shows how to declare a pointer.

LISTING 4-10 pointer.cpp

```
// pointer.cpp -- our first pointer variable
#include <iostream.h>
int main(void)
{
    int updates = 6;          // declare a variable
    int * p_updates;          // declare pointer to an int

    p_updates = &updates;     // assign address of int to pointer

// NOTE: Some systems may require you to type cast the addresses
//       in cout statements to unsigned

    // express values two ways
    cout << "Values: updates = " << updates;
    cout << ", *p_updates = " << *p_updates << "\n";

    // express address two ways
    cout << "Addresses: &updates = " << &updates;
    cout << ", p_updates = " << p_updates << "\n";

    // use pointer to change value
    *p_updates = *p_updates + 1;
    cout << "Now updates = " << updates << "\n";
    return 0;
}
```

Here is the output:

```
Values: updates = 6, *p_updates = 6
Addresses: &updates = 0x85b0fff4, p_updates = 0x85b0fff4
Now updates = 7
```

As you can see, the `int` variable *updates* and the pointer variable *p_updates* are just two sides of the same coin. The *updates* variable represents the value as primary and uses the `&` operator to get the address, while the *p_updates* variable represents the address as primary and uses the *operator to get the value. See Figure 4-8. And because *p_updates* points to *updates*, **p_updates* and *updates* are completely equivalent. So we can use **p_updates* exactly as we would use a type `int` variable. As the program shows, we can even assign values to **p_updates*. Doing so changes the value of the pointed-to value, *updates*.

```
int jumbo = 23;
int * pe = &jumbo;
```

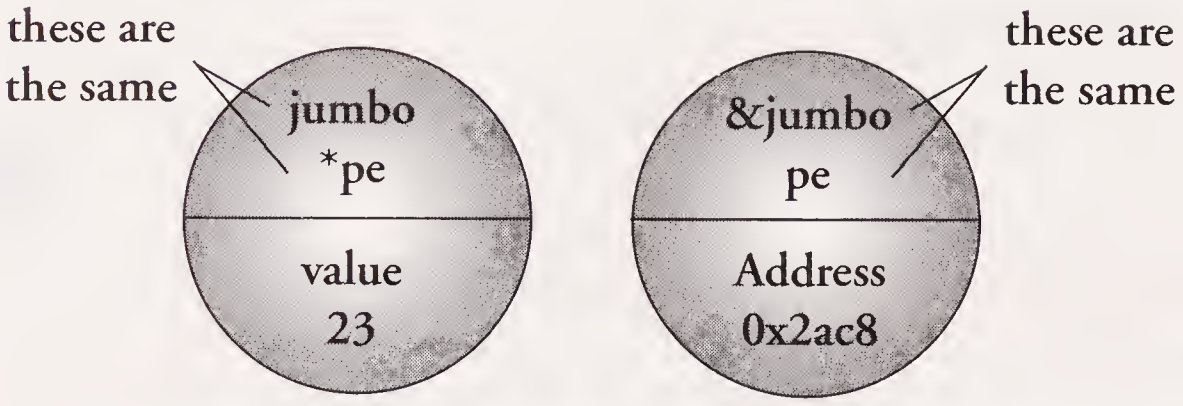


Figure 4-8 *Two sides of a coin*

DECLARING AND INITIALIZING POINTERS

Let's examine the process of declaring pointers. A computer needs to keep track of the type of value a pointer refers to. For instance, the address of a char looks the same as the address of a `double`, but `char` and `double` use different numbers of bytes and different internal formats for storing values. Therefore, a pointer declaration must specify what type of data the pointer points to. For example, our program has this declaration:

```
int * p_updates;
```

This states that the combination * *p_updates* is type `int`. (The space between the * operator and the variable name is optional.) Because the * operator is used by applying it to a pointer, the *p_updates* variable itself must be a pointer. We say that *p_updates* points to type `int`. We also say that the type for *p_updates* is pointer-to-`int` or, more concisely, `int*`. To repeat: *p_updates* is a pointer (an address), and **p_updates* is an `int` and not a pointer. Also see Figure 4-9.



In C++, the combination `int*` is a derived *type name*, the type pointer-to-*int*.

You use the same syntax to declare pointers to other types:

```
double * tax_ptr;       // tax_ptr points to type double
char * str;             // str points to type char
```

Because we declare *tax_ptr* as a pointer-to-`double`, the compiler knows that **tax_ptr* is a type `double` value. That is, it knows that **tax_ptr* represents a number stored in floating-point format and occupying (on most systems) eight bytes. A pointer variable is never simply a pointer. It is always a pointer to some type. So *tax_ptr* is type pointer-to-`double` (or type `double*`) and *str* is type pointer-to-`char` (or *char* `*`). Although both are pointers, they are pointers of two different types. Thus, like arrays, pointers are derived from other types.

Note that while *tax_ptr* and *str* point to data types of two different sizes, the two variables *tax_ptr* and *str* themselves typically are the same size. That is, the address of a `char` is the same size as the address of a `double`, much as 1015 might be the street address for a department store while 1022 could be the street address of a small cottage. The size or value of an address doesn't really tell you anything about the size or kind of variable or building found at that address. Usually, addresses require two or four bytes, depending on the computer system. (Some systems may have larger addresses, and a system conceivably could use different address sizes for different data types, but that would be unusual.)

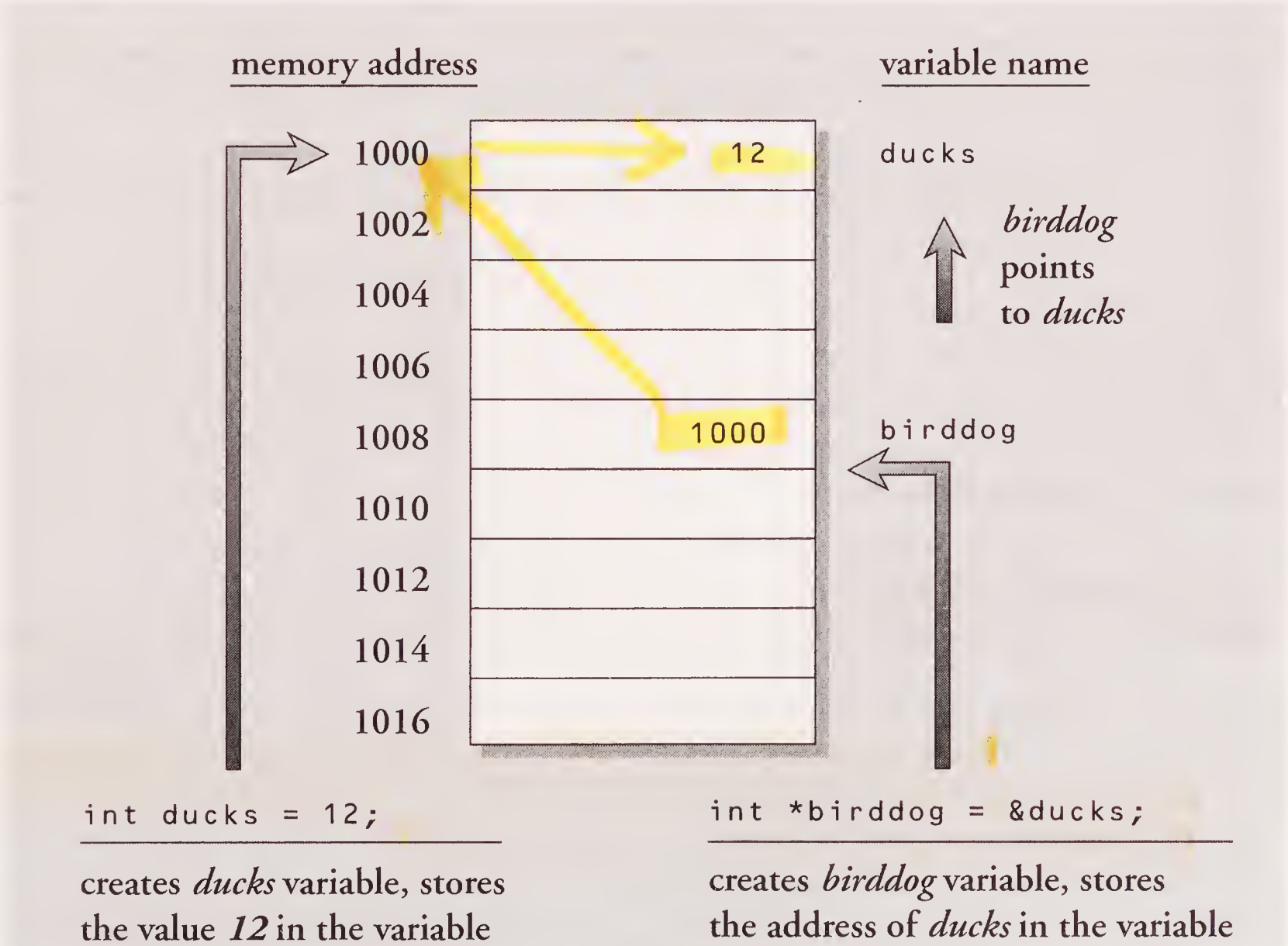
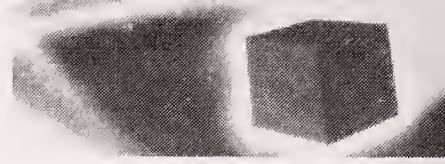
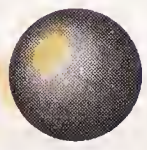


Figure 4-9 *Pointers store addresses*



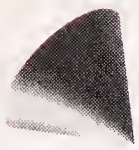


You can use a declaration statement to initialize a pointer. In that case, the pointer, not the pointed-to value, is initialized. That is, the statements

```
int higgens = 5;
int * pi = &higgens;
```

set *pi* and not **pi* to the value *&higgens*.

Listing 4-11 demonstrates initializing a pointer to an address.



LISTING 4-11 init_ptr.cpp

```
// init_ptr.cpp -- initialize a pointer
#include <iostream.h>
int main(void)
{
    int higgens = 5;
    int * pi = &higgens;

// NOTE: Some users may need to type cast the addresses
//       in cout statements (&higgens and pi) to unsigned
    cout << "Value of higgens = " << higgens
        << "; Address of higgens = " << &higgens << "\n";
    cout << "Value of *pi = " << *pi
            << "; Value of pi = " << pi << "\n";
    return 0;
}
```

Here is the output:

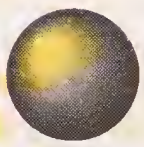
```
Value of higgens = 5; Address of higgens = 0x8fa0fff4
Value of *pi = 5; Value of pi = 0x8fa0fff4
```

You can see that the program initializes *pi*, not **pi*, to the address of *higgens*.

Danger awaits those who use pointers incautiously. One extremely important point is that when you create a pointer in C++, the computer allocates memory to hold an *address*, but it does not allocate memory to hold the *data* that will be pointed to. Creating space for the data involves a separate step. Omitting that step, as in the following, is an invitation to disaster:

```
long * fellow;          // create a pointer-to-long
* fellow = 223323;      // place a value in never-never land
```

Sure, *fellow* is a pointer. But *where* does it point? The code failed to assign an address to *fellow*. So where is the value *223323* placed? We can't say. Because *fellow* wasn't initialized, it could have any value. Whatever that value is, the program interprets it to be the address at which to store *223323*. If *fellow* happens to have the value 1200, then the computer attempts to place the data at address 1200, even if that happens to be an address in the middle of your program code. Chances are that wherever *fellow* points, you don't want to put the number *223323* there. This kind of error can produce some of the most insidious and hard-to-trace bugs around.



Pointer Golden Rule: ALWAYS set a pointer to a definite and appropriate address before applying the dereferencing operator (*) to it.

POINTERS AND NUMBERS

Pointers are not integer types, even though computers typically handle addresses as integers. Conceptually, however, pointers are distinct types from integers. Integers are numbers you can add, subtract, divide, and so on. But a pointer describes a location, and it doesn't make sense to, say, multiply two locations times each other. So, in terms of the operations you can perform with them, pointers and integers are different from each other. Consequently, you can't simply assign an integer to a pointer:

```
int * pi;
pi = 0xB8000000;      // type mismatch
```

Here the left-hand side is a pointer to `int`, so it can be assigned an address. But the right-hand side is just an integer. You may know that 0xB8000000 is the combined segment-offset address of video memory on your system, but nothing in the statement tells the program that this number is an address. C will let you make assignments like this. But C++ enforces type agreement more stringently, and the compiler will give you an error message saying you have a type mismatch. So if you want to use a numeric value as an address, you should use a type cast to convert the number to the appropriate address type:

```
int * pi;
pi =  (int *) 0xB8000000; // types now match
```

Now both sides of the assignment statement represent addresses of integers, so the assignment is valid. Note that being the address of a type `int` value doesn't mean that *pi* itself is type `int`. For instance, in the large memory model on an IBM PC, type `int` is a two-byte value, while the addresses are four-byte values.

Pointers have some other interesting properties that we'll discuss as they become relevant. Meanwhile, let's look at how pointers can be used to manage run-time allocation of memory space.

ALLOCATING MEMORY WITH NEW

Now that you have some feel for how pointers work, let's see how they can implement that important OOP technique of allocating memory as a program runs. So far, we've initialized pointers to the addresses of variables; this is *named* memory allocated during compile time, and the pointers merely provided an alias for memory you could access directly by name anyway. The true worth of pointers comes into play when you allocate *unnamed* memory during run time to hold values. In this case, pointers become the *only* access to that memory. In C, you could allocate memory with the library function `malloc()`. You can still do so in C++, but C++ also has a better way, the `new` operator.

Let's try out this new technique by creating unnamed, run-time storage for a type `int` value and accessing the value with a pointer. The key is the C++ `new` operator. You tell `new` what data type you want memory for; `new` finds a block of the correct size and returns the address of the block. Assign this address to a pointer, and you're in business. Here's a sample of the technique:

```
int * pn = new int;
```

The `new int` part tells the program we want some new storage suitable for holding an `int`. The `new` operator uses the type to figure out how many bytes are needed. Then it finds the memory and returns the address. We assign the address to *pn*, which is declared to be of type pointer-to-`int`. So now *pn* is the address and **pn* is the value stored there. Compare this with assigning the address of a variable to a pointer:

```
int higgens;
int *pi = &higgens;
```

In both cases (*pn* and *pi*) we assign the address of an `int` to a pointer. In the second case, we can also access the `int` by name: *higgens*. In the first case, our only access is via the pointer. That raises a question: Since the memory to which *pn* points lacks a name, what do you call it? We say that *pn* points to a *data object*. This is not "object" in the sense of "object-oriented programming"; it's just "object" in the sense of "thing." The term "data object" is more general than the term "variable," for it means any block of memory allocated for a data item. Thus a variable is also a data object, but the memory the *pn* points to is not a variable. The pointer method for handling data objects may seem more awkward at first, but it offers greater control over how your program manages memory.

The general form for obtaining and assigning memory for a single data object, which can be a structure as well as a fundamental type, is this:

```
type * pointer_name = new type;
```

You use the data type twice, once to specify the kind of memory requested and once to declare a suitable pointer. Of course, if you've already declared a pointer of the correct type, you can use it instead of declaring a new one. Listing 4-12 illustrates using `new` with two different types.

LISTING 4-12 new.cpp

```
// new.cpp -- using the new operator
#include <iostream.h>
int main(void)
{
    int * pi = new int;           // allocate space for an int
    *pi = 1001;                   // store a value there

// NOTE: Some users may need to type cast the addresses
//       in cout statements (pi and pd) to unsigned
    cout << "int ";
    cout << "value = " << *pi << ": location = " << pi << "\n";

    double * pd = new double;     // allocate space for a double
    *pd = 10000001.0;             // store a double there

    cout << "double ";
    cout << "value = " << *pd << ": location = " << pd << "\n";
    cout << "size of pi = " << sizeof pi;
    cout << ": size of *pi = " << sizeof *pi << "\n";
    cout << "size of pd = " << sizeof pd;
    cout << ": size of *pd = " << sizeof *pd << "\n";
    return 0;
}
```

Here is the output:

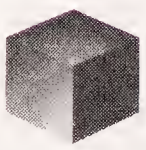
```
int value = 1001: location = 0x857011ec
double value = 10000001: location = 0x857011f4
size of pi = 2: size of *pi = 2
size of pd = 2: size of *pd = 8
```

Program Notes

The program uses `new` to allocate memory for a type `int` data object and for a type `double` data object. This occurs while the program is running. The pointers *pi* and *pd* point to these two data objects. Without them, you cannot access those memory locations. With them, you can use **pi* and **pd* just as you would use variables. Assigning values to **pi* and **pd* assigns values to the new data objects. Similarly, printing **pi* and **pd* displays those values.

The program also demonstrates one of the reasons you have to declare the type a pointer points to. An address in itself reveals only the beginning address of the object stored, not its type nor the number of bytes used. Look at the addresses of the two values. They are just numbers with no type or size information. Also, note that the size of a pointer-to-`int` is the same as the size of a pointer-to-`double`. Both are just addresses. But because `new.cpp` declared the pointer types, the program knows that **pd* is a `double` value of eight bytes, while **pi* is an `int` value of two bytes. So when `new.cpp` prints the value of **pd*, `cout` can tell how many bytes to read and how to interpret them.

The sharp-eyed among you may be wondering how the program manages to store a number like 0x857011ec, which appears to be four bytes long, in a two-byte variable. The system on which we ran the program (IBM PC-compatible) divides memory into segments of 64K (a K is 1024 bytes), and the program uses a two-byte pointer to hold the offset of a memory location from the beginning of the segment used for data. But `cout` was implemented to print both the two-byte segment value (0x8570) and the two-byte offset (0x11ec) as a single four-byte value (0x857011ec).



Out of Memory?

It's possible that the computer may not have sufficient memory available to satisfy a new request. When that is the case, `new` returns the value *0*. In C++, a pointer with the value *0* is called the *null pointer*. C++ guarantees that the null pointer never points to valid data, so it often is used to indicate failure for operators or functions that otherwise return useable pointers. Once you learn about `if` statements (Chapter 6), you can check to see if `new` returns the null pointer and thus protect your program from attempting to exceed its bounds.

Using new to Create Dynamic Arrays

If all a program needs is a single value, you may as well declare a simple variable, for that is simpler, if less impressive, than using `new` and a pointer to manage a single small data

object. More typically, you use `new` with larger chunks of data, such as arrays, strings, and structures. This is where `new` gets useful. Suppose, for example, you're writing a program that may or may not need an array, depending on information given to the program while it is running. If you create an array by declaring it, the space is allocated when the program is compiled. Whether or not the program finally uses the array, the array is there, using up memory. This is called *static binding*, meaning the array is built into the program at compilation time. But with `new`, you can create an array during run time if you need it and skip creating the array if you don't need it. Or you can select an array size after the program is running. This is called *dynamic binding*, meaning the array is created while the program is running, and the array is called a *dynamic array*. With static binding, you must specify the array size when you write the program. With dynamic binding, the program can decide upon an array size while the program runs.

For now, we'll look at two basic matters concerning dynamic arrays: how to use C++'s `new` operator to create an array, and how to use a pointer to access array elements.

Creating a Dynamic Array with new

It's easy to create a dynamic array in C++; you tell `new` the type of array element you desire and how many elements you want. The syntax requires that you follow the type name with the number of elements in brackets. For instance, if you need an array of 10 `int`s, do this:

```
int * psome = new int [10];  // get a block of 10 ints
```

The *new* operator returns the *address* of the first element of the block. In this example, that value is assigned to the pointer *psome*. Note that *psome* is a pointer to a single `int`, the first element of the block. It's your responsibility to keep track of how many elements are in the block. That is, the compiler doesn't keep track of the fact that *psome* points to the first of *ten* integers, so you have to write your program so the program keeps track of the number of elements.

The general form for allocating and assigning memory for an array is this:

```
type_name * pointer_name = new type_name [num_elements];
```

This returns a block of memory large enough to hold *num_elements* elements of type *type_name*, with *pointer_name* pointing to the first element. As you're about to see, you can use *pointer_name* in many of the same ways you can use an array name.

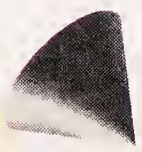
Using a Dynamic Array

Once you create a dynamic array, how do you use it? First, think about the problem conceptually. The statement

```
int * psome = new int [10] ; // get a block of 10 ints
```

creates a pointer *psome* that points to the first element of a block of ten `int` values. Think of it as a finger pointing to that element. Suppose an `int` occupies two bytes. Then, by moving your finger two bytes in the correct direction, you can point to the second element. Altogether, there are ten elements, and that tells the range over which you can move your finger. Thus, the `new` statement supplies you with all the information you need to identify every element in the block.

Now think about the problem practically. How do you access one of these elements? The first element is no problem. Because *psome* points to the first element of the array, **psome* is the value of the first element. That leaves nine more elements to access. The simplest way may surprise you if you haven't worked with C: just use the pointer as if it were an array name. That is, you can use *psome[0]* instead of **psome* for the first element, *psome[1]* for the second element, and so on. So using a pointer to access a dynamic array turns out to be very simple, even if it may not be immediately obvious why the method works. The reason you can do this is that C and C++ handle arrays internally by using pointers anyway. This near equivalence of arrays and pointers is one of the beauties of C and C++. We'll elaborate on this equivalence in a moment. First, Listing 4-13 shows how you can use `new` to create a dynamic array and then use array notation to access the elements. It also points out a fundamental difference between a pointer and a true array name.



LISTING 4-13 arraynew.cpp

```
// arraynew.cpp -- using the new operator for arrays
#include <iostream.h>
int main(void)
{
    double * p3 = new double [3];     // space for 3 doubles
    p3[0] = 0.2;                      // treat p3 like an array name
    p3[1] = 0.5;
    p3[2] = 0.8;
    cout << "p3[1] is " << p3[1] << ".\n";
    p3 = p3 + 1;                      // increment the pointer
    cout << "Now p3[0] is " << p3[0] << " and ";
    cout << "p3[1] is " << p3[1] << ".\n";
    return 0;
}
```

Here is the output:

```
p3[1] is 0.5.
```

```
Now p3[0] is 0.5 and p3[1] is 0.8.
```

As you can see, `arraynew.cpp` uses the pointer *p3* as if it were the name of an array, with *p3[0]* being the first element, and so on. The fundamental difference between an array name and a pointer shows in the following line:

```
p3 = p3 + 1;    // okay for pointers, wrong for array names
```

You can't change the value of an array name. But a pointer is a variable, hence you can change its value. Note the effect of adding *1* to *p3*. The expression *p3[0]* now refers to the former second element of the array. Thus adding *1* to *p3* caused it to point to the second element instead of the first. The actual addresses of consecutive `int`s typically differ by two or four bytes, so the fact that adding *1* to *p3* gives the address of the next element suggests that there is something special about pointer arithmetic. There is.

POINTERS, ARRAYS, AND POINTER ARITHMETIC

The near equivalence of pointers and array names stems from pointer arithmetic and how C++ handles arrays internally. First, let's check out the arithmetic. Adding 1 to an integer variable increases its value by 1. But adding 1 to a pointer variable increases its value by the number of bytes of the type it points to. So adding 1 to a pointer-to-`double` adds 8 to the numerical value on systems with eight-byte `double`, while adding 1 to a pointer-to-`short` adds 2 to the pointer value if `short` is two bytes. Listing 4-14 demonstrates this amazing point. It also shows a second important point: C++ interprets the array name as an address.

LISTING 4-14 addpntrs.cpp

```
// addpntrs.cpp -- pointer addition
#include <iostream.h>
int main(void)
{
    double wages[3] = {10000.0, 20000.0, 30000.0};
    short stacks[3] = {3, 2, 1};

// NOTE: Some users may need to type cast the addresses
//       in cout statements (ps and pw) to unsigned

// Here are two ways to get the address of an array
    double * pw = wages;         // name of an array = address
    short * ps = &stacks[0];     // use address operator
                                 // with array element
    cout << "pw = " << pw << ", *pw = " << *pw << "\n";
```

```
    pw = pw + 1;
    cout << "add 1 to the pw pointer:\n";
    cout << "pw = " << pw << ", *pw = " << *pw << "\n\n";

    cout << "ps = " << ps << ", *ps = " << *ps << "\n";
    ps = ps + 1;
    cout << "add 1 to the ps pointer:\n";
    cout << "ps = " << ps << ", *ps = " << *ps << "\n\n";

    cout << "access two elements with array notation\n";
    cout << stacks[0] << " " << stacks[1] << "\n";
    cout << "access two elements with pointer notation\n";
    cout << *stacks << " " << *(stacks + 1) << "\n";

    cout << sizeof wages << " = size of wages array\n";
    cout << sizeof pw << " = size of pw pointer\n";
    return 0;
}
```

Here is the output:

```
pw = 0x7c87ffd8, *pw = 10000
add 1 to the pw pointer:
pw = 0x7c87ffe0, *pw = 20000

ps = 0x7c87fff0, *ps = 3
add 1 to the ps pointer:
ps = 0x7c87fff2, *ps = 2

access two elements with array notation
3 2
access two elements with pointer notation
3 2
24 = size of wages array
2 = size of pw pointer
```

Program Notes

In most contexts, C++ interprets the name of an array as the address of its first element. Thus the statement

```
double * pw = wages;
```

makes *pw* a pointer to type `double`, then initializes *pw* to *wages*, which is the address of the first element of the *wages* array. For *wages*, as with any array, we have the following equality:

```
wages = &wages[0] = address of first element of array
```

Just to show that we aren't jiving you, we used the address operator explicitly in the expression *&stacks[0]* for initializing the *ps* pointer to the first element of the *stacks* array.

Next, the program inspects the values of *pw* and **pw*. The first is an address and the second is the value at that address. Because *pw* points to the first element, the value displayed for **pw* is that of the first element, *10000*. Then the program adds 1 to *pw*. As promised, this adds 8 (8 + 8 = 10 in hexadecimal) to the numeric address value, because `double` on our system is eight bytes. This makes *pw* equal to the address of the second element. Thus **pw* is now *20000*, the value of the second element. See Figure 4-10. (The address values in the figure are adjusted to make the figure clearer.)

After this, the program goes through similar steps for *ps*. This time, because *ps* points to type `short` and because `short` is two bytes, adding 1 to the pointer increases its value by 2. Again, the result is to make the pointer point to the next element of the array.

```
double wages[3] = {10000.0, 20000.0, 30000.0};
short stacks[3] = {3, 2, 1};
double * pw = wages;
short * ps = &stacks[0];
```

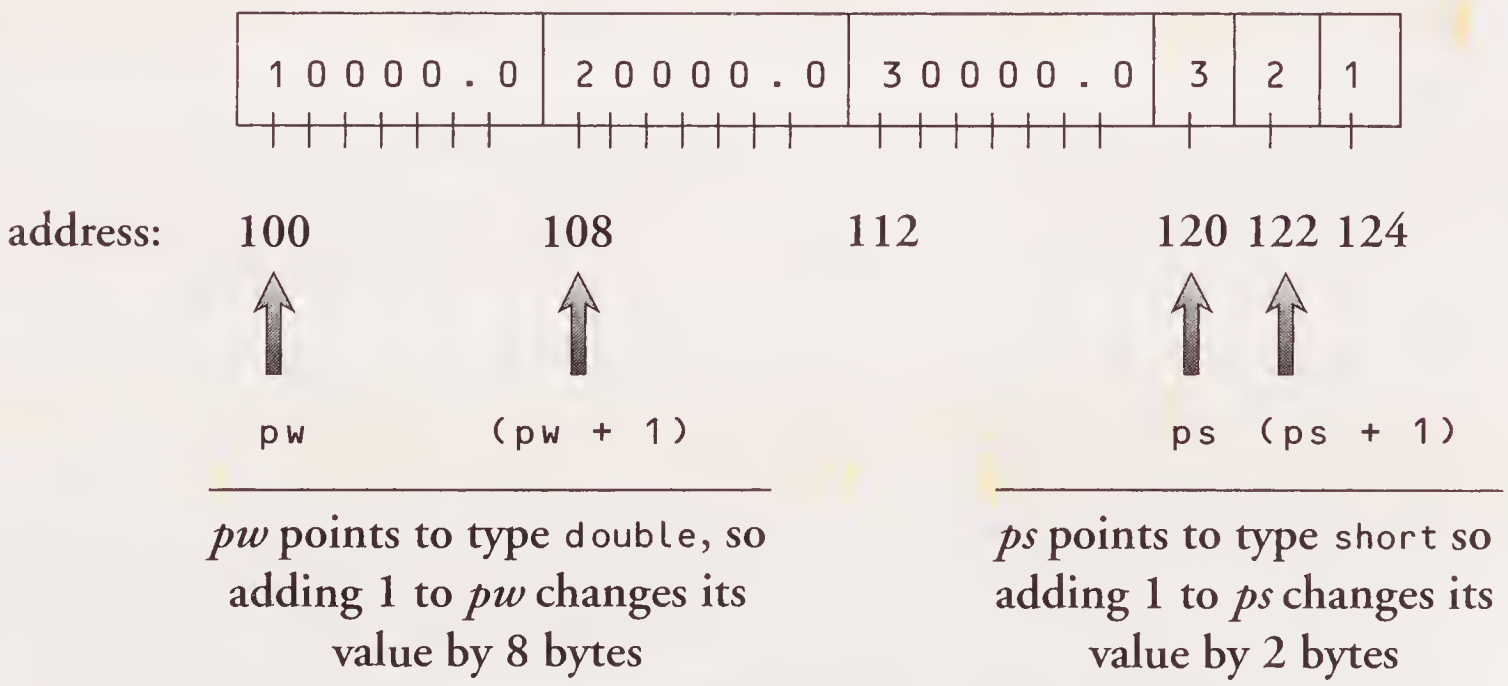
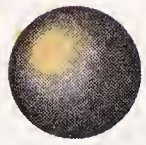


Figure 4-10 *Pointer addition*



Remember, adding 1 to a pointer variable increases its value by the number of bytes of the type to which it points.

Now consider the array expression *stacks[1]*. The C++ compiler treats this expression exactly as if you wrote it as **(stacks + 1)*. The second expression means calculate the address of the second element of the array, then find the value stored there. The end result is precisely what we mean by *stacks[1]*. (Operator precedence requires using the parentheses in the expression **(stacks +1)*. Without them, 1 would be added to **stacks* instead of to *stacks*.)

The program output demonstrates that **(stacks + 1)* and *stacks[1]* are the same. Similarly, **(stacks + 2)* is the same as *stacks[2]*. In general, wherever you use array notation, C++ makes the following conversion:

```
arrayname[i] becomes *(arrayname + i)
```

And if you use a pointer instead of an array name, C++ makes the same conversion:

```
pointername[i] becomes *(pointername + i)
```

Thus in many respects you can use pointer names and array names the same way. You can use array notation with either. You can apply the dereferencing operator (*) to either. In most expressions, each represents an address. One difference is that you can change the value of a pointer while an array name is a constant:

```
pointername = pointername + 1;          // valid
arrayname = arrayname + 1;              // not allowed
```

The second difference is that applying the `sizeof` operator to an array name yields the size of the array, but applying `sizeof` to a pointer yields the size of the pointer, even if the pointer points to the array. For instance, in Listing 4-14, both *pw* and *wages* refer to the same array. But applying the `sizeof` operator to them produced the following results:

```
24 = size of wages array          ← displaying sizeof wages
2 = size of pw pointer            ← displaying sizeof pw
```

This is one case in which C++ doesn't interpret the array name as an address.

In short, using `new` to create an array and using a pointer to access the different elements is a simple matter. Just treat the pointer as an array name. Understanding why this works, however, is an interesting challenge. If you really wish to understand arrays and pointers, you should review their mutual relationships carefully. In fact, you've been exposed to quite a bit of pointer knowledge lately, so let's summarize what's been revealed about pointers and arrays to date.

Summarizing Pointer Points

Declaring Pointers: To declare a pointer to a particular *type*, use this form:

```
typename * pointername;
```

Examples:

```
double * pn;      // pn points to a double value
char * pc;        // pc points to a char value
```

Here *pn* and *pc* are pointers and `double*` and `char*` are the C++ notations for the types pointer-to-`double` and pointer-to-`char`.

Assigning Values to Pointers: A pointer should be assigned a memory address. You can apply the `&` operator to a variable name to get an address of named memory, and the `new` operator returns the address of unnamed memory.

Examples:

```
double bubble = 3.2;
pn = &bubble;           // assign address of bubble to pn
pc = new char;          // assign address of newly allocated
                        // char memory to pc
```

Dereferencing Pointers: Dereferencing a pointer means referring to the pointed-to value. Apply the dereferencing, or indirect value, operator (*) to a pointer to dereference it. Thus if *pn* is a pointer to *bubble*, as in the last example, then **pn* is the pointed-to value, or *3.2* in this case.

Examples:

```
cout << *pn;      // print the value of bubble
*pc = 'S';        // place the value 'S' in the memory location
                  // whose address is pc
```

Distinguishing Between a Pointer and the Pointed-to Value: Remember, if *pi* is a pointer-to-`int`, that **pi* is not a pointer-to-`int`; instead, **pi* is the complete equivalent to a type `int` variable. It is *pi* that is the pointer.

Examples:

```
int * pi = new int;     // assigns an address to the pointer pi
*pi = 5;                // stores the value 5 at that address
```

Array Names: In most contexts, C++ treats the name of an array as being equivalent to the address of the first element of an array.

Example:

```
int tacos[10];          // tacos is the same as &tacos[0]
```

One exception is when you use the name of an array with the `sizeof` operator. In that case, `sizeof` returns the size of the entire array, in bytes.

Pointer Arithmetic: C++ allows you to add an integer to a pointer. The result equals the original address value plus a value equal to the number of bytes in the pointed-to object.

Examples:

```
float fog = 2.2;
float * pf = &fog;    // suppose pf and &fog are the address 3000
pf = pf + 1;          // now pf is 3004 if a float is four bytes

int tacos[10];        // suppose tacos, or &tacos[10], is 4000
                      // then tacos + 1 is 4002 if int is two bytes
```

Dynamic Binding and Static Binding for Arrays: Use an array declaration to create an array with static binding, that is, an array whose size is set during compilation time:

```
int tacos[10];        // static binding, size fixed at compile time
```

Use the `new` operator to create an array with dynamic binding (a dynamic array), that is, an array that is allocated and whose size can be set during run time:

```
int size;
cin >> size;
int * pz = new int [size];      // dynamic binding, size set at
                                // run time
```

Array Notation and Pointer Notation: Using bracket array notation is equivalent to dereferencing a pointer:

```
tacos[0] means *tacos means the value at address tacos
tacos[3] means *(tacos + 3) means the value at address tacos + 3
```

This is true both for array names and for pointer variables, so you can use either pointer notation or array notation with pointers and with array names.

Examples:

```
int * pi = new int [10];// pi points to block of 10 ints
*pi = 5;                // set zero element to 5
pi[0] = 6;              // reset zero element to 6
pi[10] = 44;            // set eleventh element to 44
int tacos[10];
*(tacos + 4) = 12;      // set tacos[4] to 12
```

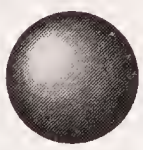
POINTERS AND STRINGS

The special relationship between arrays and pointers extends to strings. Consider the following code:

```
char flower[10] = "rose";
cout << flower << "s are red\n";
```

The name of an array is the address of its first element, so *flower* in the `cout` statement is the address of the `char` element containing the character *'r'*. The `cout` object assumes that the address of a `char` is the address of a string, so it prints the character at that address, then continues printing characters until it runs into the null character (*\0*). In short, if you give `cout` the address of a character, it prints everything from that character to the first null character following it.

The crucial element here is not that *flower* is an array name but that *flower* acts as the address of a `char`. This implies, for instance, that you can use a pointer-to-`char` as an argument to `cout`, also, since it, too, is the address of a `char`. Of course, that pointer should point to the beginning of a string. We'll check that out in a moment. But first, what about the final part of the above `cout` statement? If *flower* really is the address of the first character of a string, what is the expression *"s are red\n"*? To be consistent with `cout`'s handling of string output, this quoted string should also be an address. And it is, for in C++ a quoted string, like an array name, serves as the address of its first element. So the code above doesn't really send a whole string to `cout`, it just sends the string address. This means strings in an array, quoted strings, and strings described by pointers are all handled equivalently. Each is really passed along as an address. That's certainly less work than passing each and every character in a string.



With `cout` and with most C++ expressions, the name of an array of `char`, a pointer-to-`char`, and a quoted string constant are all interpreted as the address of the first character of a string.

Listing 4-15 illustrates using the different forms of strings. It uses two functions from the string library. The `strlen()` function, which we've used before, returns the length of a string. The `strcpy()` function copies a string from one location to another. Both have function prototypes in the `string.h` header file. The program also showcases some pointer misuses that you should try to avoid.

LISTING 4-15 pntstr.cpp

```
// pntstr.cpp -- using pointers to strings
#include <iostream.h>
#include <string.h>                 // declare strlen(), strcpy()
int main(void)
{
    char animal[20] = "bear";       // animal holds bear
    char * bird = "penguin";        // bird holds address of string
    char * ps;                      // uninitialized

    cout << animal << " and ";      // display bear
    cout << bird << "\n";           // display penguin
    cout << ps << "\n";             // blunder - display garbage

    cout << "Enter a kind of animal: ";
    cin >> animal;                  // ok if input < 20 chars
    cout << "Enter a kind of bird: ";
    cin >> bird;                    // not recommended, may work on
                                    // some systems, but behavior
                                    // is undefined
    // cin >> ps;   Too horrible a blunder to try; ps doesn't
    //              point to allocated space

    ps = animal;                    // set ps to point to string
    cout << ps << "s!\n";           // ok, same as using animal
    ps = bird;                      // reassign pointer
    cout << ps << "s!\n";           // ok, same as using bird

    ps = new char[strlen(animal) + 1];  // get new storage
    strcpy(ps, animal);             // copy string to new storage
    cout << "Now ps points to " << ps << "!\n";
    return 0;
}
```

Here is a sample run:

```
bear and penguin
                                    ← output from cout << ps << "\n";
Enter a kind of animal: sloth
Enter a kind of bird: parrot
sloths!
parrots!
Now ps points to sloth!
```

Program Notes

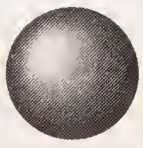
The program in Listing 4-15 creates one *char* array (*animal*) and two pointers-to-`char` variables (*bird* and *ps*). The program begins by initializing the *animal* array to the *"bear"* string, just as we've initialized arrays before. Then the program does something new. It initializes a pointer-to-`char` to a string:

```
char * bird = "penguin";    // bird holds address of string
```

Remember, *"penguin"* really represents the address of the string, so this statement assigns the address of *"penguin"* to the *bird* pointer. This means you can use the pointer *bird* just as you would have used the string *"penguin"*, as in *cout* << *"A concerned"* << *bird* << *"speaks\n"*. Finally, the pointer *ps* remains uninitialized, so it doesn't point to any string. (This, you recall, is usually a bad idea, and this example is no exception.)

Next, the program illustrates that we can use the array name *animal* and the pointer *bird* equivalently with `cout`. Both, after all, are the addresses of strings, and `cout` displays the two strings (*bear* and *penguin*) stored at those addresses. When we made the error of attempting to display *ps*, we got a blank line. Creating an uninitialized pointer is a bit like distributing a blank signed check; you lack control over how it will be used. We were a little lucky here, for *ps*, being uninitialized, may have accidentally pointed to some awkward location. As it was, it pointed to a location containing a zero, so nothing was displayed. Otherwise, you might get some garbage output.

For input, the situation is a bit different. It's safe to use the array *animal* for input as long as the input is short enough to fit into the array. Using a pointer to a string constant, however, can cause problems. In our case it worked, for the original string was big enough to hold the input. But C++ doesn't guarantee that string constants are stored uniquely. That is, if you use a string constant *"penguin"* elsewhere in the program, the compiler might set *bird* to point to the same string instead of using a separate string. So reading a value into one string could affect what you thought was an independent string elsewhere. Worse yet is trying to read information into the location pointed to by *ps*. Because *ps* is not initialized, we don't know where the information will wind up. It might even overwrite information already in memory. Fortunately, it's easy to avoid these problems—just use a sufficiently large `char` array to receive input. Don't use string constants, and don't use uninitialized pointers to receive input.



When reading a string into a program, you should always use the address of previously allocated memory. This address can be in the form of an array name or of a pointer that has been initialized using `new`.

The end of a program shows a common string-handling trick for conserving memory. It also lets us bring `new` back into the discussion.

```
ps = new char[strlen(animal) + 1];  // get new storage
strcpy(ps, animal);         // copy string to new storage
```

The string "*sloth*" doesn't completely fill the *animal* array, so we're wasting space. This bit of code uses `strlen()` to find the length of the string; it adds 1 to get the length including the null character. Then the program uses `new` to allocate just enough space to hold the string. Finally, it uses the `strcpy()` function to copy the string in *animal* to the newly allocated memory. The `strcpy()` function takes two arguments. The first is the destination address, and the second is the address of the string to be copied. It's up to you to make certain that the destination really is allocated and has sufficient space to hold the copy. We accomplished that here by using `strlen()` to find the correct size and using `new` to get free memory.

By the way, don't use the following code to copy a string:

```
ps = animal;    // this does NOT copy a string
```

All this does is reset *ps* to the address of the *animal* array. This makes both *ps* and *animal* refer to the same storage location; all you get are two names for the same thing. But using `strcpy()` copies a string from one location to a second location, giving you two separate, but identical, strings.

Of course, we haven't really saved space because *animal* is still there, but this technique is the starting point for more sophisticated memory management.

USING NEW TO CREATE DYNAMIC STRUCTURES

You've seen how it can be advantageous to create arrays during run time instead of compile time. The same holds true for structures. You need allocate space for only as many structures as a program needs during a particular run. Again, the `new` operator is the tool to use. With it, you can create dynamic structures. Again, "dynamic" means the memory is allocated during run time, not during compilation; the term doesn't mean that the structure is pulsing with barely contained energy. Incidentally, because classes are much like structures, you'll be able to use the techniques you learn for structures with classes, too.

Using `new` with structures has two parts: creating the structure and accessing its members. To create a structure, use the structure type with `new`. For example, to create an unnamed structure of the *inflatable* type and to assign its address to a suitable pointer, you can do the following:

```
inflatable * ps = new inflatable;
```

This assigns to *ps* the address of a chunk of free memory large enough to hold a structure of the *inflatable* type. The syntax, you'll note, is exactly the same as it was for C++'s built-in types.

The tricky part is accessing members. When you create a dynamic structure, you can't use the dot membership operator with the structure name because the structure has no name. All you have is its address. C++ provides an operator just for this situation: the arrow membership operator (->). This operator, formed by typing a hyphen, then a greater-than symbol, does for pointers to structures what the dot operator does for structure names. For instance, if *ps* points to a type *inflatable* structure, then *ps->price* is the *price* member of the pointed-to structure. See Figure 4-11.

Sometimes new users become confused about when to use the dot operator and when to use the arrow operator to specify a structure member. The rule is simple. If the structure identifier is the name of a structure, use the dot operator. If the identifier is a pointer to the structure, use the arrow operator.

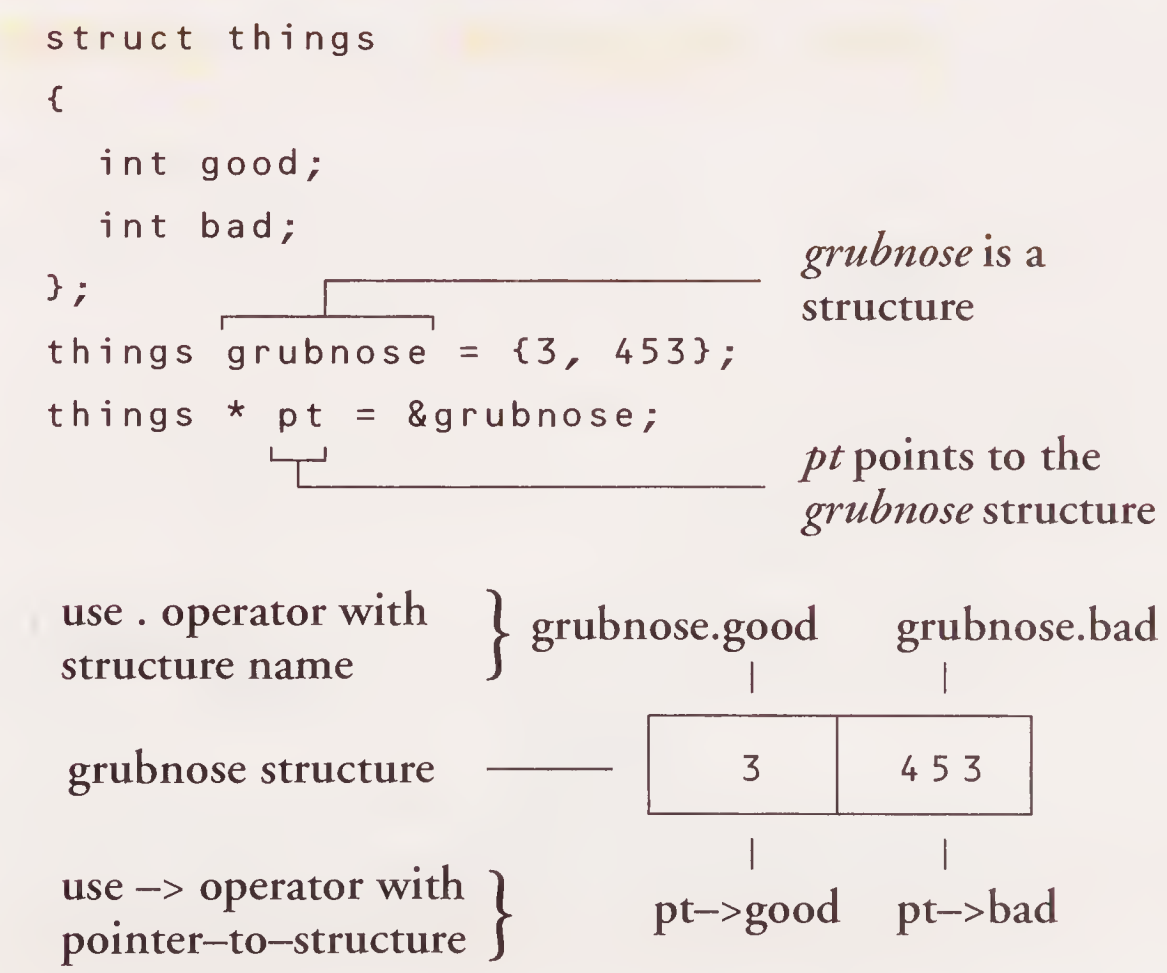


Figure 4-11 *Identifying structure members*

A second, uglier approach is to realize that if *ps* is a pointer to a structure, then **ps* represents the pointed-to value, the structure itself. Then, because **ps* is a structure, *(*ps).price* is the *price* member of the structure. C++'s operator precedence rules require that we use parentheses in this construction.

Listing 4-16 uses `new` to create an unnamed structure and demonstrates both pointer notations for accessing structure members.

LISTING 4-16 newstruc.cpp

```
// newstruc.cpp -- using new with a structure
#include <iostream.h>
struct inflatable                   // structure template
{
    char name[20];
    float volume;
    double price;
};
int main(void)
{
    inflatable * ps = new inflatable;// allot structure space
    cout << "Enter name of inflatable item: ";
    cin.get( (*ps).name, 20);   // method 1 for member access
    cout << "Enter volume in cubic feet: ";
    cin >> ps->volume;          // method 2 for member access
    cout << "Enter price: $";
    cin >> ps->price;
    cout << "Name: " << ps->name << "\n";
    cout << "Volume: " << ps->volume << " cubic feet\n";
    cout << "Price: $" << ps->price << "\n";
    return 0;
}
```

Here is a sample run:

```
Enter name of inflatable item: Fabulous Frodo
Enter volume in cubic feet: 1.40
Enter price: $17.99
Name: Fabulous Frodo
Volume: 1.4 cubic feet
Price: $17.99
```

FREEING MEMORY WITH DELETE

Using `new` to request memory when you need it is just the more glamorous half of the C++ memory-management package. The other half is the `delete` operator, which lets you return memory to the memory pool when you are finished with it. That is an important step for making the most effective use of memory. Memory that you return, or *free*, can then be reused by other parts of your program. You use `delete` by following it with a pointer to a block of memory originally allocated with `new`:

```
int * ps = new int;      // allocate memory with new
delete ps;               // free memory with delete
```

This removes the memory that *ps* points to; it doesn't remove the pointer *ps* itself. So you can reuse *ps* to, say, point to another `new` allocation.

You cannot use `delete` to free memory created by declaring variables:

```
int jugs = 5;            // ok
int * pi = & jugs;       // ok
delete pi;               // not allowed, pi not allocated by new
```



Use `delete` only to delete memory allocated with *new*.

Note that the critical test for using `delete` is that you use it with *memory* allocated by `new`. This doesn't mean you have to use the same *pointer* you used with `new`; instead, you have to use the same *address*:

```
int * ps = new int;      // allocate memory
int * pq = ps;           // set second pointer to same block
delete pq;               // delete with second pointer
```

Ordinarily, you won't create two pointers to the same block of memory, for that raises the possibility of mistakenly trying to delete the same block twice. But, as you'll soon see, using a second pointer does make sense when you work with a function that returns a pointer.

A NEW AND DELETE EXAMPLE

What new and marvelous things can you accomplish by using `new` and `delete`? Listing 4-17 shows some of the possibilities. The program defines a function that returns a pointer to an input string. This function reads the input into a large temporary array, then uses `new` to create a chunk of memory sized to fit to the input string. Then the function returns the pointer to the block. This approach could conserve a lot of memory

for programs that read in a large number of strings. Suppose your program had to read a thousand strings, that the largest string might be seventy-nine characters long, but that most of the strings would be much shorter. If you used *char* arrays to hold the strings, you'd need a thousand arrays of eighty characters each. That's 80,000 bytes, and much of that block of memory would wind up unused. Alternatively, you could create an array of a thousand pointers to `char`, then use `new` to allocate only the amount of memory needed for each string. That could save tens of thousands of bytes. Instead of having to use a large array for every string, you fit the memory to the input. Even better, you also could use `new` to find space to store only as many pointers as are needed. Well, that's a little too ambitious for right now. Even using an array of a thousand pointers is a little too ambitious for right now, but Listing 4-17 illustrates some of the techniques. Also, just to illustrate how `delete` works, the program uses it to free memory for reuse.

LISTING 4-17 delete.cpp

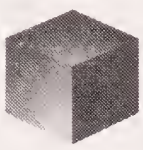
```
// delete.cpp -- using the delete operator
#include <iostream.h>
#include <string.h>
char * getname(void);    // function prototype
int main(void)
{
    char * name;          // create pointer but no storage

// NOTE:  Some users may need to type cast name to
//        unsigned instead of to (int *)

    name = getname();     // assign address of string to name
    cout << name << " at " << (int *) name << "\n";
    delete name;          // memory freed

    name = getname();     // reuse freed memory
    cout << name << " at " << (int *) name << "\n";
    return 0;
}
char * getname(void)      // return pointer to new string
{
    char temp[80];

    cout << "Enter last name: ";
    cin >> temp;
    char * pn = new char[strlen(temp) + 1];
    strcpy(pn, temp);     // copy string into smaller space
    return pn;            // temp lost when function ends
}
```



COMPATIBILITY NOTE

As mentioned earlier, Zortech C++'s implementation of `cout`, at the time of this writing, doesn't, in general, recognize pointers. Therefore, to print an address, you need to type cast the address to an integer type. However, Zortech's `cout` does recognize the pointer-to-`char` type and print the string at the indicated address, so no change is needed to print strings.

Here is a sample run:

```
Enter last name: Fredeldumpkin
Fredeldumpkin at 0x85b30dd6
Enter last name: Pook
Pook at 0x85b30dd6
```

Note that freeing memory resulted in the program using the same address for both names.

PROGRAM NOTES

First, consider the function `getname()`. It uses `cin` to place an input word into the *temp* array. Next, it uses `new` to allocate new memory to hold the word. Including the null character, the program needs `strlen(temp) + 1` characters to store the string, so that's the value given to `new`. Once the space is available, `getname()` uses the standard library function `strcpy()` to copy the string from *temp* to the new block. The function doesn't check to see if the string will fit or not, but `getname()` covers that by requesting the right number of bytes with `new`. Finally, the function returns *pn*, the address of the string copy.

In `main()`, the return value (the address) is assigned to the pointer *name*. This pointer is defined in `main()`, but it points to the block of memory allocated in the `getname()` function. The program then prints the string and the address of the string. Here we have used a small bit of trickery. The variable *name* is a pointer, and `cout` normally prints pointers as hexadecimal addresses. But `cout` processes a pointer-to-`char` differently from other pointers. Instead of printing the address value of a pointer-to-`char`, `cout` prints the string stored at the address. Therefore, to print the address itself, `main()` uses C++'s type casting facility to convert the *name* pointer to another type: pointer-to-`int`. That type `cout` prints as an address.

Next, after freeing the block pointed to by *name*, `main()` calls `getname()` a second time. C++ doesn't guarantee that newly freed memory is the first chosen the next time `new` is used, but in our sample run the same block of memory wound up being used for the next input.

To appreciate some of the more subtle aspects of this program, you should know a little more about how C++ handles memory. So let's preview some material that's covered more fully in Chapter 8.

AUTOMATIC STORAGE, STATIC STORAGE, THE FREE STORE

C++ has three ways of managing memory for data, depending on the method used to allocate memory: automatic storage, static storage, and the free store. Data objects allocated in these three ways differ from each other in how long they remain in existence. We'll take a quick look at each type.

Automatic Variables

Ordinary variables defined inside a function are called automatic variables. They come into existence automatically when the function containing them is invoked, and they expire when the function terminates. For example, the *temp* array in Listing 4-17 exists only while the `getname()` function is active. When program control returns to `main()`, the memory used for *temp* is freed automatically. So if `getname()` had returned the address of *temp*, the *name* pointer in `main()` would have been left pointing to a memory location that soon would be reused. That's one reason we had to use `new` in `getname()`.

Actually, automatic values are local to the block containing them. A block is a section of code enclosed between braces. So far, all our blocks have been entire functions. But, as you'll see in the next chapter, you can have blocks within a function. So if you define a variable inside one of those blocks, it exists only while the program is executing statements inside the block.

Static Storage

Static storage is storage that exists throughout the execution of an entire program. There are two ways to make a variable static. One is to define it externally, outside a function. The other is to use the keyword `static` when declaring a variable:

```
static double fee = 56.50;
```

Under K&R C, only static arrays and structures could be initialized, while C++ 2.0 and ANSI C allow you to initialize automatic arrays and structures, too. However, as some of you may have discovered, some C++ 2.0 implementations have not yet implemented initialization for automatic arrays and structures.

Chapter 8 takes up static storage in more detail. The main point here about automatic and static storage is that these methods rigidly define the lifetime of a variable. Either the variable exists for the entire duration of a program (the static variable) or else it exists only while a particular function or block is being executed (the automatic variable).

The Free Store

The `new` and `delete` operators, however, provide a more flexible approach. They manage a pool of memory, which C++ refers to as the *free store*. This pool is separate from the memory used for static and automatic variables. As Listing 4-17 shows, `new` and `delete` let you allocate memory in one function and free it in another. Thus the lifetime of the data is not tied arbitrarily to the life of the program or the life of a function. Using `new` and `delete` together gives you much more control over how a program uses memory than does using ordinary variables.

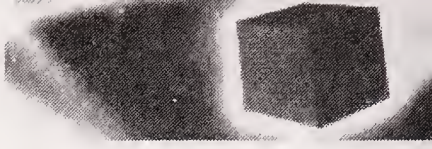
Pointers are among the most powerful of C++ tools. They also are the most dangerous, for they permit computer-unfriendly actions, such as using an uninitialized pointer to access memory or attempting to free the same memory block twice. Furthermore, until you get used, through practice, to pointer notation and pointer concepts, pointers can be confusing. This book will return to pointers several more times in the hopes that each exposure will make you more comfortable with them.

Summary

The array, the structure, and the pointer are three C++ derived types. The array can hold several values all of the same type in a single data object. By using an index, or subscript, you can access the individual elements in an array.

The structure can hold several values of different types in a single data object, and you can use the membership operator (`.`) to access individual members. The first step in using structures is creating a structure template defining what members the structure holds. The name, or tag, for this template then becomes a new type identifier. You can then declare structure variables of that type.

Pointers are variables designed to hold addresses. We say a pointer points to the address it holds. The pointer declaration always states what type of object a pointer points to. Applying the dereferencing operator (`*`) to a pointer yields the value at the location to which the pointer points.



A string is a series of characters terminated by a null character. A string can be represented by a quoted string constant, in which case the null character is implicitly understood. You can store a string in an array of `char`, and you can represent a string by a pointer-to-`char` that is initialized to point to the string. The `strlen()` function returns the length of a string, not counting the null character. The `strcpy()` function copies a string from one location to another. When using these functions, include the `string.h` header file.

The `new` operator lets you request memory for a data object while a program is running. The operator returns the address of the memory it obtains, and you can assign that address to a pointer. The only means to access that memory is to use the pointer. If the data object is a simple variable, you can use the dereferencing operator to indicate a value. If the data object is an array, you can use the pointer as if it were an array name to access the elements. If the data object is a structure you can use the pointer dereferencing operator (->) to access structure members.

Pointers and arrays are closely connected. If *ar* is an array name, then the expression *ar[i]* is interpreted as **(ar + i)*, with the array name being interpreted as the address of the first element of the array. Thus the array name plays the same role as a pointer. In turn, you can use a pointer name with array notation to access elements in an array allocated by `new`.

The `new` and `delete` operators let you control explicitly when data objects are allocated and when they are returned to the memory pool. Automatic variables, which are those declared within a function, and static variables, which are defined outside a function or with the keyword `static`, are less flexible. An automatic variable comes into being when the block containing it (typically a function definition) is entered and expires when the block is left. A static variable persists for the duration of a program.

Review Questions

1. How would you declare each of the following?
   a. *actors* is an array of 30 `char`
   b. *betsie* is an array of 100 `short`
   c. *chuck* is an array of 13 `float`
   d. *dipsea* is an array of 64 `long double`
2. Declare an array of 5 `int`s and initialize it to the first five odd integers.
   INT ODD = {1,3,5,7,9};
3. Write a statement that assigns the sum of the first and last elements of the array in question 2 to the variable *even*.
   INT EVEN = ODD[0] + ODD[4];

4. Write a statement that displays the value of the second element in the `float` array *ideas*. FLOAT IDEAS [5]; COUT << IDEA[1];

5. Declare an array of `char` and initialize it to the string *"cheeseburger"*. CHAR FOOD[] = "CHEESEBURGER";

6. Devise a structure template that describes a fish. The structure should include the kind, the weight in whole ounces, and the length in fractional inches. STRUCT FISH { CHAR KIND[20]; INT OZ; FLOAT INCH; };

7. Declare a variable of the type defined in question 6 and initialize it.

8. Suppose *ted* is a `double` variable. Declare a pointer that points to *ted*, and use the pointer to display *ted*'s value.

9. Suppose *treakle* is an array of 10 `float`s. Declare a pointer that points to the first element of *treakle*, and use the pointer to display the first and last elements of the array. FLOAT TREAKLE [10] {5.5, 6.2, 3}; 0 9

10. Write a code fragment that asks the user to enter a positive integer and then creates a dynamic array of that many `int`s.

11. Is the following valid code? If so, what does it print?

```
cout << (int *) "Home of the jolly bytes";
```

12. Write a code fragment that dynamically allocates a structure of the type described in question 6 and which then reads a value for the structure member that represents the kind of fish.

CONST NUM = 10;
FLOAT *PF = & TREAKLE [0];
COUT << *PF << "\n";
COUT << *PF(+9) "\n";

A LOOP

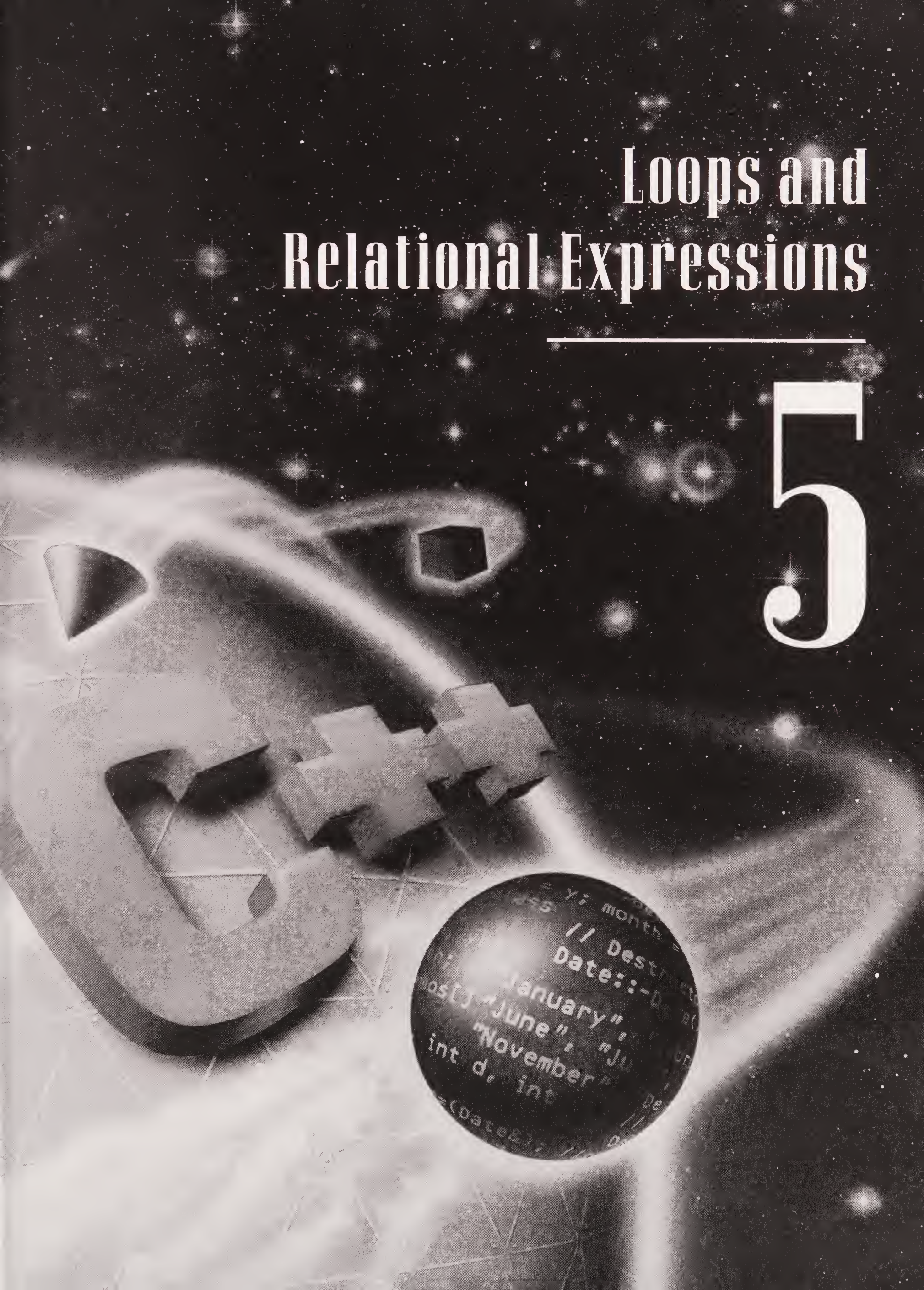
FOR (I = 0; I =< NUM; I ++;) {
FLOAT SUM = *PF(+1) + SUM;
}

FLOAT AVG = SUM/NUM

ASK FOR (CHAR) LETTER IN NAME
THEN LOOPS BY
ASKING FOR EACH LETTER IN A ROW
"ENTER YOUR FIRST LETT"
2ND
LAST
YOUR NAME...

Loops and Relational Expressions

5



Computers do more than store data. They analyze, consolidate, rearrange, extract, modify, extrapolate, synthesize, and otherwise manipulate data. Sometimes they even distort and trash data, but we'll try to steer clear of that kind of behavior. To perform their manipulative miracles, programs need tools for performing repetitive actions and for making decisions. C++, of course, provides such tools. Indeed, it uses the same `for` loops, `while` loops, `do while` loops, `if` statements, and `switch`es that regular C employs, so if you know C, you can zip through this and the next chapter. (But don't zip too fast—you don't want to miss how `cin` handles character input!) These various program control statements often use relational expressions and logical expressions to govern their behavior. This chapter discusses loops and relational expressions, and the next chapter follows up with branching statements and logical expressions.

Introducing the for Loop

Circumstances often call upon a program to perform repetitive tasks, such as adding together the elements of an array one by one or printing some paean to productivity twenty times. The C++ `for` loop makes such tasks easy to do. Let's look at a loop in Listing 5-1, see what it does, then discuss how it works.

LISTING 5-1 forloop.cpp

```
// forloop.cpp -- introducing the for loop
#include <iostream.h>
int main(void)
{
    //   initialize; test ; update
    for (int i = 0; i < 5; i++)
            cout << "C++ knows loops.\n";
    cout << "C++ knows when to stop.\n";
    return 0;
}
```

Here is the output:

```
C++ knows loops.
C++ knows loops.
C++ knows loops.
C++ knows loops.
C++ knows loops.
C++ knows when to stop.
```

This loop begins by defining an integer *i* and setting it to *0*:

```
int i = 0
```

This is the *loop initialization* part of the loop. Then, in the *loop test*, the program tests to see if *i* is less than *5*:

```
i < 5
```

If so, the program executes the following statement, which is termed the loop *body*:

```
cout << "C++ knows loops.\n";
```

Then the program uses the *loop update* part of the loop to increase *i* by *1*:

```
i++
```

We've used the ++ operator, called the *increment operator*. It increments the value of its operand by 1. (The increment operator is not restricted to `for` loops. For instance, you can use

```
i++;
```

instead of

```
i = i + 1;
```

as a statement in a program.) Incrementing *i* completes the first cycle of the loop.

Next, the loop begins a new cycle by comparing the new *i* value with *5*. Because the new value (*1*) also is less than *5*, the loop prints another line and then finishes by incrementing *i* again. That sets the stage for a fresh cycle of testing, executing a statement, and updating the value of *i*. The process continues until the loop updates *i* to *5*. Then the next test fails, and the program moves on to the next statement after the loop.

FOR LOOP PARTS

A `for` loop, then, provides a step-by-step recipe for performing repeated actions. Let's take a more detailed look at how it's set up. The usual parts of a `for` loop handle these steps:

- Setting a value initially
- Performing a test to see if the loop should continue
- Executing the loop actions
- Updating value(s) used for the test

The C++ loop design positions these elements so that you can spot them at a glance. The initialization, test, and update actions constitute a three-part control section enclosed in parentheses. Each part is an expression, and semicolons separate the expressions from each other. The statement following the control section is called the *body* of the loop, and it is executed as long as the *test-expression* remains true:

```
for (initialization; test-expression; update-expression)
    body
```

C++ syntax counts a complete `for` statement as a single statement, even though it may incorporate one or more statements in the *body* portion.

The loop evaluates *initialization* just once. Typically, programs use this expression to set a variable to a starting value, then use the variable to count loop cycles. If you're coming to C++ from C, you may be a little surprised that our example *declares* the variable *i* in the `for` loop control section. C doesn't allow you to use a declaration there, but C++ does. The C++ programming style is to declare variables near the location they're first used. This program first uses *i* inside the loop, so we declared *i* in the initialization part of the control statement. Of course, you could instead declare the variable before the loop, if you wanted to.

The *test-expression* determines whether the loop body gets executed. Typically, this expression is a *relational* expression, that is, one that compares two values. Our example, for instance, compares the value of *i* to *5*, checking to see if *i* is less than *5*. If the comparison is true, the program executes the loop body. Actually, C++ doesn't limit *test-expression* to true-false comparisons. You can use any expression. If the expression evaluates to zero, the loop terminates. If the expression evaluates to nonzero, the loop

continues. Listing 5-2 demonstrates this by using the expression *i* as the test condition. (In the update section, *i*--is similar to *i*++ except that it *decreases* the value of *i* by 1 each time it's used.)

LISTING 5-2 num_test.cpp

```
// num_test.cpp -- use numeric test in for loop
#include <iostream.h>
int main(void)
{
    cout << "Enter the starting countdown value: ";
    int limit;
    cin >> limit;
    for (int i = limit; i; i--)     // entry-condition test
          cout << "i = " << i << "\n";
    cout << "Done now that i = " << i << "\n";
    return 0;
}
```

Here is the output:

```
Enter the starting countdown value: 4

i = 4
i = 3
i = 2
i = 1
Done now that i = 0
```

Note that the loop terminates when *i* reaches *0*. Relational expressions, such as *i* < *5*, fit into this framework by evaluating to *1* if true and to *0* if false. Also note that the *i* variable is not confined to the loop; we're free to display its value after the loop completes.

The `for` loop is an *entry-condition* loop. This means the test expression is evaluated *before* each loop cycle. The loop never executes the loop body when the test expression is false. For instance, suppose we rerun the program in Listing 5-2 but give *0* as a starting value. Because the test condition fails the very first time it's evaluated, the loop body never gets executed:

```
Enter the starting countdown value: 0
Done now that i = 0
```

This look-before-you-loop attitude can help keep a program out of trouble. The *update-expression* is evaluated at the end of the loop, after the body has been executed.

Typically, it's used to increase or decrease the value of the variable keeping track of the number of loop cycles. However, it can be any valid C expression, as can the other control expressions. This makes the `for` loop capable of much more than simply counting from 0 to 5, the way our first loop did. You'll see some examples later.

The `for` loop body consists of a single statement, but you'll soon see how to stretch that rule. C++ doesn't let you use a declaration statement in the loop body itself because that would lead to multiple declarations of the same variable and also because

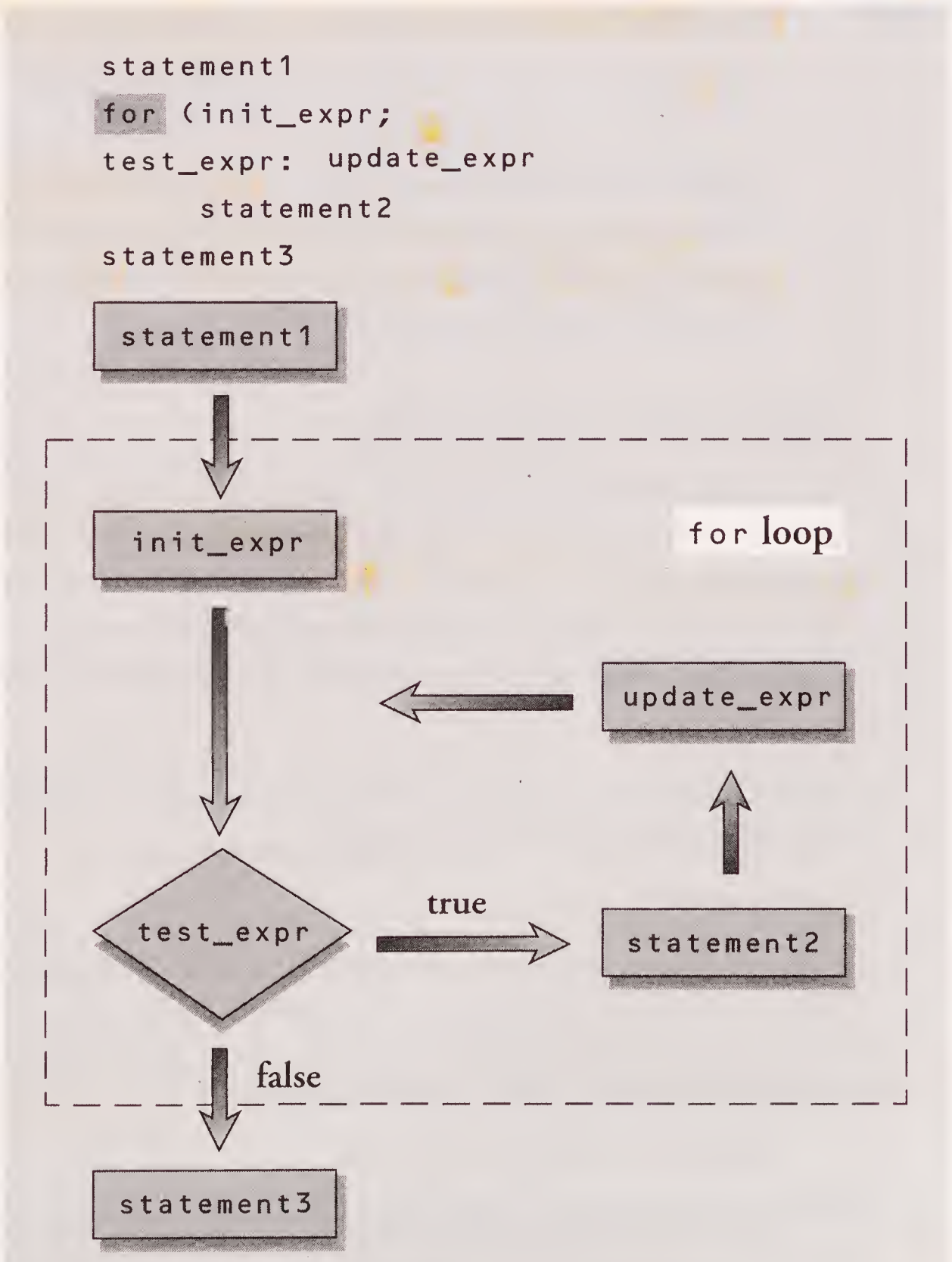
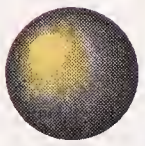


Figure 5-1 *The* `for` *loop*

it's conceivable that program execution could skip the loop entirely. Figure 5-1 summarizes the `for` loop design.

A `for` statement looks something like a function call because it uses a name followed by paired parentheses. However, `for`'s status as a C++ keyword prevents the compiler from thinking `for` is a function. It also prevents you from naming a function `for`.



Common C++ style is to place a space between `for` and the following parentheses and to omit space between a function name and the following parentheses:

```
for (int i = 6; i < 10; i++)
    smart_function(i);
```

Other control statements, such as `if` and `while`, are treated similarly to `for`. This serves to reenforce visually the distinction between a control statement and a function call. Also, common practice is to indent the body of a `for` statement to make it stand out visually.

Expressions and Statements

A `for` control section uses three expressions. Within its self-imposed limits of syntax, C++ is a very expressive language. Any value or any valid combination of values and operators constitute an expression. For example, *10* is an expression with the value *10* (no surprise), and *28 * 20* is an expression with the value *560*. In C++, every expression has a value. Often the value is obvious. For instance, the expression

```
22 + 27
```

is formed from two values and the addition operator, and it has the value *49*. Sometimes the value is less obvious. For example,

```
x = 20
```

is an expression because it's formed from two values and the assignment operator. C++ defines the value of an assignment expression to be the value of the left-hand member, so the expression above has the value *20*. The fact that assignment expressions have values permits statements such as the following:

```
maids = (cooks = 4) + 3;
```

The expression *cooks = 4* has the value *4*, so *maids* is assigned the value *7*. However, just because C++ permits this behavior doesn't mean you should encourage it. But the same rule that makes this peculiar statement possible also makes the following useful statement possible:

```
x = y = z = 0;
```

This is a fast way to set several variables to the same value. The precedence table (Appendix D) reveals that assignment associates right-to-left, so first *0* is assigned to *z*, then the value of *z = 0* is assigned to *y*, and so on.

Finally, as we mentioned, relational expressions evaluate to *1* if true and to *0* if false. The short program in Listing 5-3 illustrates some points about expression values. The << operator has higher precedence than the operators we're using in the expressions, so we've used parentheses to enforce the correct order.

LISTING 5-3 express.cpp

```
// express.cpp -- values of expressions
#include <iostream.h>
int main(void)
{
    int x;

    cout << "The expression x = 100 has the value ";
    cout << (x = 100) << "\n";
    cout << "Now x = " << x << "\n";
    cout << "The expression x < 3 has the value ";
    cout << (x < 3) << "\n";
    cout << "The expression x > 3 has the value ";
    cout << (x > 3) << "\n";
    return 0;
}
```

Here is the output:

```
The expression x = 100 has the value 100
Now x = 100
The expression x < 3 has the value 0
The expression x > 3 has the value 1
```

A C++ expression is a value or a combination of values and operators, and every C++ expression has a value.

To evaluate the expression *x = 100*, C++ must assign the value *100* to *x*. When the very act of evaluating an expression changes the value of data in memory, we say the evaluation has a *side effect*. Thus evaluating an assignment expression has the side effect of changing the assignee's value. You may think of assignment as the intended effect, but

from the standpoint of how C++ is constructed, evaluating the expression is the primary effect. Not all expressions have side effects. For instance, evaluating *x + 15* calculates a new value, but it doesn't change the value of *x*. But evaluating *++x + 15* does have a side effect, since it involves incrementing *x*.

From expression to statement is a short step; just add a semicolon. Thus

```
age = 100
```

is an expression, while

```
age = 100;
```

is a statement. Any expression can become a statement by adding a semicolon, but the result may not make programming sense. For instance, if *rodents* is a variable, then

```
rodents + 6;      // valid, but useless, statement
```

is a valid C++ statement. The compiler will pass it, but the statement doesn't accomplish anything useful. The program merely calculates the sum, does nothing with it, and goes on to the next statement.

Of course, what makes rules interesting are the exceptions. Note that C++ syntax causes the two semicolons in a `for` control section to act as expression separators, not as statement makers, so the three parts of the control section really are expressions, not statements, and the entire loop counts as a single statement.

Nonexpressions and Statements

Some concepts, such as knowing the structure of a `for` loop, are crucial to understanding C++. But there are also relatively minor aspects of syntax that can suddenly bedevil you just when you think you understand the language. We'll look at a couple of them now.

While it is true that adding a semicolon to any expression makes it a statement, the reverse is not true. That is, removing a semicolon from a statement does not necessarily convert it to an expression. Of the kinds of statements we've used so far, *return* statements, declaration statements, and `for` statements don't fit the statement = expression + semicolon mold. For instance, while

```
int toad;
```

is a statement, the fragment `int toad` is not an expression and does not have a value. This makes code like the following invalid:

```
eggs = int toad * 1000; // invalid, not an expression
cin >> int toad;        // can't combine declaration with cin
```

Similarly, you can't assign a `for` loop to a variable:

```
int fx = for (int i = 0; i< 4; i++)
    cout >> i;   // not possible
```

Here the `for` loop is not an expression, so it has no value, so it can't be assigned.

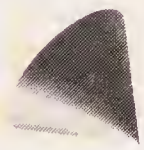
At this point you may have noticed an apparent inconsistency. If the semicolons in a `for` statement control section separate expressions from each other and if dropping the semicolon from a declaration statement doesn't produce an expression, then what do we call the *int i = 0* initialization in a `for` loop like this?

```
for (int i = 0; i< 4; i++)
    cout >> i;
```

As we said, *int i = 0* is not an expression. Nor is it a declaration statement, since the semicolon is not part of it. Because C didn't allow this form at all, C++ had to come up with a new term: the *declaration-statement expression.* What it boils down to is that C++ programmers want to be able to declare and initialize a variable in a `for` loop initialization, and they'll do whatever is necessary to C++ syntax and to the English language to make it possible.

BACK TO THE FOR LOOP

Let's be a bit more ambitious with loops. Listing 5-4 uses a loop to calculate and store the first 16 factorials. Factorials, which are handy for computing odds, are calculated the following way. Zero factorial, written as 0!, is defined to be 1. Then 1! is 1 * 0!, or 1. Next, 2! is 2 * 1!, or 2. Then 3! is 3 * 2!, or 6, and so on, with the factorial of each integer being the product of that integer with the preceding factorial. (One of the pianist Victor Borge's best-known monologues features phonetic punctuation, in which the exclamation mark is pronounced something like phffft pptz, with a moist accent. As of this writing, no reputable organization of mathematicians has denied that mathematicians privately use this pronunciation when discussing factorials.) The program uses one loop to calculate the values of successive fractorials, storing them in an array. Then it uses a second loop to display the results. Also, the program introduces using external declarations for values.



LISTING 5-4 formore.cpp

```
// formore.cpp -- more looping with for
#include <iostream.h>
const int Size = 16;        // example of external declaration
int main(void)
{
    double factorials[Size];
    factorials[1] = factorials[0] = 1.0;
```

```
    for (int i = 2; i < Size; i++)
            factorials[i] = i * factorials[i-1];
    for (i = 0; i < Size; i++)
    {
            cout << i << "! = ";
            cout << factorials[i] << "\n";
    }
    return 0;
}
```

Here is the output:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 3.99168e+07
12! = 4.790016e+08
13! = 6.227021e+09
14! = 8.717829e+10
15! = 1.307674e+12
```

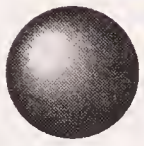
Factorials get big fast!

Program Notes

The program creates an array to hold the factorial values. Element 0 is 0!, element 1 is 1!, and so on. Since the first two factorials equal one, the program sets the first two elements of the *factorials* array to *1.0*. (Remember, the first element of an array has an index value of 0.) After that, the program uses a loop to set each factorial to the product of the index with the previous factorial. The loop illustrates that we can use the loop counter as a variable in the body of the loop.

Once the program declares *i* in the first loop, it can use *i* again in the second loop. The *i* variable is no different from any other variable; it just happens to be declared in a loop initialization. The *i* variable can be used anywhere subsequently in the `main()` function.

The program demonstrates how the `for` loop works hand in hand with arrays by providing a convenient means to access each array member in turn. Also, `formore.cpp` uses `const` to create a symbolic representation (*Size*) for the array size. Then it uses *Size* wherever the array size came into play, such as in the array definition and in the limits for the loops handling the array. Now, if you wish to extend the program to, say, 20 factorials, you just have to set *Size* to *20* in the program and recompile. By using a symbolic constant, you avoid having to manually change every occurrence of *16* to *20*.



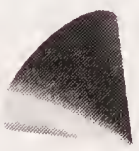
It's usually a good idea to define a `const` value to represent the number of elements in an array. Use the `const` value in the array declaration and in all other references to the array size, such as in a `for` loop.

The limit *i* < *Size* reflects the fact that subscripts for an array with *Size* elements run from *0* to *Size - 1*, so the array index should stop 1 short of *Size*. You could use the test *i* <= *Size - 1* instead, but it looks awkward in comparison.

One program sidelight is that we declared the `const int` variable *Size* outside the body of `main()`. As the end of Chapter 4 mentioned, this makes *Size external data*. The two consequences of declaring *Size* in this fashion are that *Size* exists for the duration of the program and that it can be used by all the functions in the program file. In this particular case, the program has just one function, so declaring *Size* externally has little practical effect. But multifunction programs often benefit from sharing external constants, so we'll practice using them now.

CHANGING THE STEP SIZE

So far our loops have increased the loop counter by 1 each cycle. We can change that by changing the update expression. The program in Listing 5-5, for example, increases the loop counter by a user-selected step size. Instead of using *i*++ as the update expression, it uses the expression *i* = *i* + *by*, where *by* is the user-selected step size.



LISTING 5-5 bigstep.cpp

```
// bigstep.cpp -- count as directed
#include <iostream.h>
int main(void)
{
    cout << "Enter an integer: ";
    int by;
    cin >> by;
    cout << "Counting by " << by << "s:\n";
```

```
    for (int i = 0; i < 100; i = i + by)
        cout << i << "\n";
    return 0;
}
```

Here is a sample run:

```
Enter an integer: 17
Counting by 17s:
0
17
34
51
68
85
```

Once *i* reaches the value *102*, the loop quits. The main point here is that the update expression can be any valid expression. For instance, if you wanted to square *i* and add 10 each cycle, you could use *i = i * i + 10*.

INSIDE STRINGS WITH THE FOR LOOP

The `for` loop provides a direct way to access each character in a string in turn. Listing 5-6, for example, lets you enter a string and then displays the string character by character in reverse order. The `strlen()` yields the number of characters in the string; the loop uses that value in its initializing expression to set *i* to the index of the last character in the string, not counting the null character. To count backwards, the program uses the *decrement* operator (--) to decrease the array subscript by *1* each loop. Also, Listing 5-6 uses the *greater than or equal to* relational operator (>=) to test whether the loop has reached the first element. We'll summarize all the relational operators soon.

LISTING 5-6 forstr1.cpp

```
// forstr1.cpp -- using for with a string
#include <iostream.h>
#include <string.h>
const int Size = 20;
int main(void)
{
    cout << "Enter a word: ";
    char word[Size];
    cin >> word;
```

```
    // display letters in reverse order
    for (int i = strlen(word) - 1; i >= 0; i--)
            cout << word[i];
    cout << "\n";
    return 0;
}
```

Here is a sample run:

```
Enter a word: animal
lamina
```

Yes, the program succeeds in printing *animal* backwards; choosing *animal* as a test word more clearly illustrates the effect of this program than choosing, say, *redder* or *stats*.

THE INCREMENT (++) AND DECREMENT (--) OPERATORS

C++ features several operators that are frequently used in loops, so let's take a little time to examine them now. We've already used two: the increment operator (++), which inspired the name C++, and the decrement operator (--). These perform two exceedingly common loop operations: increasing or decreasing a loop counter by 1. However, there's more to their story than you've seen to date. Each operator comes in two varieties. The *prefix* version comes before the operand, as in `++x`. The *postfix* version comes after the operand, as in `x++`. The two versions have the same effect upon the operand, but they differ in *when* they take place. It's like getting paid for mowing the lawn in advance or afterwards; both methods have the same final effect on your wallet, but they differ in when the money gets added. Listing 5-7 demonstrates this difference for the increment operator.

LISTING 5-7 plus_one.cpp

```
// plus_one.cpp -- the increment operator
#include <iostream.h>
int main(void)
{
    int a = 20;
    int b = 20;

    cout << "a   = " << a << ":   b = " << b << "\n";
    cout << "a++ = " << a++ << ": ++b = " << ++b << "\n";
    cout << "a   = " << a << ":   b = " << b << "\n";
```

```
    return 0;
}
```

Here is the output:

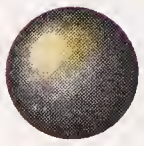
```
a    = 20:   b = 20
a++  = 20: ++b = 21
a    = 21:   b = 21
```

Roughly speaking, the notation *a*++ means "use the current value of *a* in evaluating an expression, then increment the value of *a*." Similarly, the notation ++*b* means "first increment the value of *b*, then use the new value in evaluating the expression." For example, we have the following relationships:

```
int x = 5;
int y = ++x;      // change x, then assign to y
                  // y is 6, x is 6

int z = 5;
int y = z++;      // assign to y, then change z
                  // y is 5, z is 6
```

The increment and decrement operators are a concise, convenient way to handle the common task of increasing or decreasing values by 1. You can use them with pointers as well as with basic variables. Recall that adding 1 to a pointer increases its value by the number of bytes in the type it points to. The same rule holds for incrementing and decrementing pointers.



Incrementing and decrementing pointers follow pointer arithmetic rules. Thus if *pt* points to the first member of an array, then ++*pt* changes *pt* so that it points to the second member.

The increment and decrement operators are nifty little operators, but don't get carried away and increment or decrement the same value more than once in the same statement. The problem is that the use-then-change rule and change-then-use rule can become ambiguous. That is, a statement such as

```
x++ = 2 * x++ * (3 - ++x);    // don't do it
```

can produce quite different results on different systems. C++ does not define correct behavior for this sort of statement.



COMBINATION ASSIGNMENT OPERATORS

Listing 5-5 uses the following expression to update a loop counter:

```
i = i + by
```

C++ has a combined addition and assignment operator that accomplishes the same result more concisely:

```
i += by
```

The += operator adds the values of its two operands together and assigns the result to the left-hand operand. This implies that the left-hand operand must be something you can assign to, such as a variable, an array element, a structure member, or data identified by dereferencing a pointer:

```
int k = 5;
k += 3;                    // ok, k set to 8
int *pa = new int[10];
pa[4] = 12;
pa[4] += 6;                // ok, pa[4] set to 18
*(pa + 4) += 7;            // ok, pa[4] set to 25
34 += 10;                  // quite wrong
```

Each arithmetic operator has a corresponding assignment operator, as summarized in Table 5-1. Each operator works analogously to +=. Thus the statement

```
k *= 10;
```

replaces the current value of *k* with a value ten times greater.

Table 5-1 Combined assignment operators

Operator	Effect (L=left operand, R=right operand)
+=	assigns L+R to L
-=	assigns L-R to L
*=	assigns L*R to L
/=	assigns L/R to L
%=	assigns L%R to L

COMPOUND STATEMENTS, OR BLOCKS

The format, or syntax, for writing a C++ *for* statement may seem restrictive to you because the body of the loop must be a single statement. That's awkward if you want the loop body to contain several statements. Fortunately, C++ provides a syntax loophole through which you may stuff as many statements as you like into a loop body. The trick is to use paired braces to construct a *compound statement*, or *block*. The block consists of the braces and the enclosed statements and, for the purposes of syntax, counts as a *single* statement. For instance, the program in Listing 5-8 uses braces to combine three separate statements into a single block. This enables the body of the loop to prompt the user, read input, and do a calculation. The program calculates the running sum of the numbers you enter, and this provides a natural occasion for using the `+=` operator.

LISTING 5-8 block.cpp

```
// block.cpp -- use a block statement
#include <iostream.h>
int main(void)
{
    cout << "The Amazing Accounto will sum and average ";
    cout << "five numbers for you.\n";
    cout << "Please enter five values:\n";
    double number;
    double sum = 0.0;
    for (int i = 1; i <= 5; i++)
    {                                   // block starts here
        cout << "Value " << i << ": ";
        cin >> number;
        sum += number;
    }                                   // block ends here
    cout << "Five exquisite choices indeed! ";
    cout << "They sum to " << sum << "\n";
    cout << "and average to " << sum / 5 << ".\n";
    cout << "The Amazing Accounto bids you adieu!\n";
    return 0;
}
```

Here is a sample run:

```
The Amazing Accounto will sum and average five numbers for you.
Please enter five values:
Value 1: 1942
```

```
Value 2: 1948
Value 3: 1957
Value 4: 1974
Value 5: 1980
Five exquisite choices indeed! They sum to 9801
and average to 1960.2.
The Amazing Accounto bids you adieu!
```

Suppose you left in the indentation but omitted the braces:

```
for (int i = 1; i <= 5; i++)
    cout << "Value " << i << ": ";      // loop ends here
    cin >> number;                      // after the loop
    sum += number;
cout << "Five exquisite choices indeed! ";
```

The compiler ignores indentation, so only the first statement would be in the loop. Thus the loop would print the five prompts and do nothing more. After the loop completed, the program would move to the following lines, reading and summing just one number.

Compound statements have another interesting property. If you define a new variable inside a block, the variable persists only as long as the program is executing statements within the block. When execution leaves the block, the variable is deallocated. That means the variable is known only within the block:

```
#include <iostream.h>
int main(void)
{
    int x = 20;
    {                                   // block starts
        int y = 100;
        cout << x << "\n";              // ok
        cout << y << "\n";              // ok
    }                                   // block ends
    cout << x << "\n";                  // ok
    cout << y << "\n";                  // invalid, won't compile
    return 0;
}
```

Note that a variable defined in an outer block is still defined in the inner block.

THE COMMA OPERATOR (OR MORE SYNTAX TRICKS)

The block, as you saw, lets you sneak two statements into a place where C++ syntax allows just one statement. The comma operator does the same for expressions, letting you sneak two expressions into a place where C++ syntax allows only one expression. For instance, suppose you have a loop in which one variable increases by 1 each cycle and a

second variable decreases by 1 each cycle. Doing both in the update part of a `for` loop control section would be convenient, but the loop syntax allows just one expression there. The solution is to use the comma operator to combine the two expressions into one:

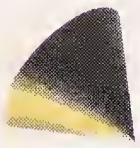
```
j++, i--   // two expressions count as one for syntax purposes
```

The comma is not always a comma operator. For instance, the comma in the declaration

```
int i, j;  // comma is a separator here, not an operator
```

serves to separate adjacent names in a list of variables.

Listing 5-9 uses the comma operator twice in a program that reverses the contents of a character array. Note that Listing 5-6 displayed the contents of an array in reverse order, but Listing 5-9 actually moves characters around in the array. The program also uses a block to group several statements into one.



LISTING 5-9 forstr2.cpp

```
// forstr2.cpp -- reversing an array
#include <iostream.h>
#include <string.h>
const int Size = 20;
int main(void)
{
    cout << "Enter a word: ";
    char word[Size];
    cin >> word;

    // physically modify array
    char temp;
    int i, j;
    for (j = 0, i = strlen(word) - 1; j < i; i--, j++)
    {                          // start block
        temp = word[i];
        word[i] = word[j];
        word[j] = temp;
    }                          // end block
    cout << word << "\n";
    return 0;
}
```

Here is a sample run:

```
Enter a word: parts
strap
```

Program Notes

Look at the `for` control section:

```
for (j = 0, i = strlen(word) - 1; j < i; i--, j++)
```

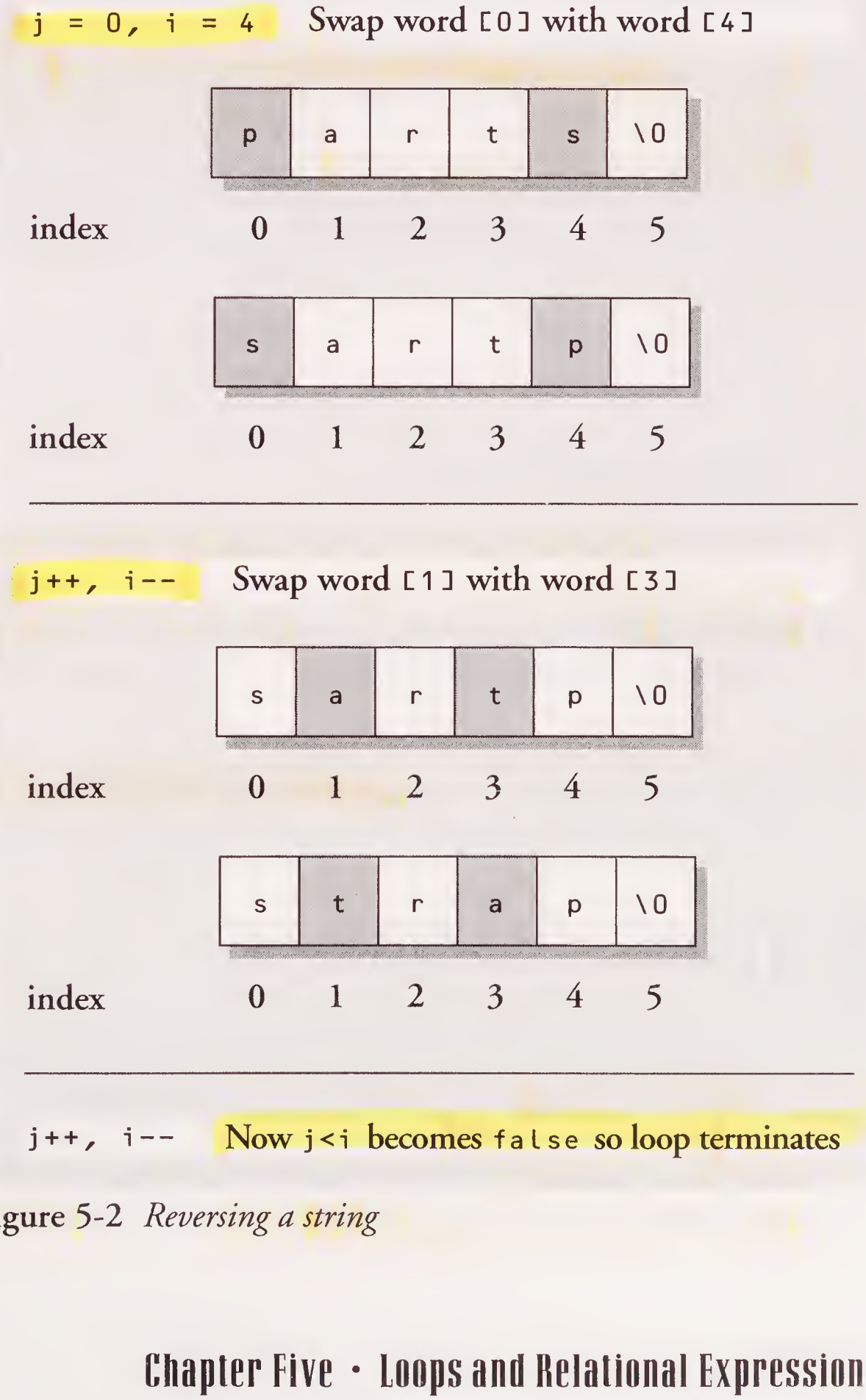


Figure 5-2 *Reversing a string*

First, it uses the comma operator to squeeze two initializations into one expression for the first part of the control section. Then it uses the comma operator again to combine two updates into a single expression for the last part of the control section.

Next, look at the body. The program uses braces to combine several statements into a single unit. In the body, the program reverses the word by switching the first element of the array with the last element. Then it increments *j* and decrements *i* so that they now refer to the next-to-the-first element and to the next-to-the-last element. This accomplished, the program swaps those elements. Note that the test condition *j*<*i* makes the loop stop when it reaches the center of the array. If it were to continue past this point, it would begin swapping the switched elements back to their original positions. See Figure 5-2.

Another thing to note is the location for declaring the variables *temp*, *i*, and *j*. We declared *i* and *j* before the loop because you can't combine two declarations with a comma operator. That's because declarations already use the comma for another purpose—separating items in a list. We could have used a single declaration-statement expression to create and initialize two variables, but it's a bit confusing visually:

```
int j = 0, i = strlen(word) - 1
```

In this case the comma is just a list separator, not the comma operator, so the expression declares and initializes both *j* and *i*. But it looks like it's just declaring *j*.

Comma Operator Tidbits

By far the most common use for the comma operator is to fit two or more expressions into a single *for* loop expression. But C++ does provide the operator with two additional properties. First, it guarantees that the first expression is evaluated before the second expression. So expressions such as the following are safe:

```
i = 20, j = 2 * i      // i set to 20, j set to 40
```

Second, C++ states that the value of a comma expression is the value of the second part. The value of the expression above, for instance, is *40*, since that is the value of *j = 2 * i*.

Relational Expressions

Computers are more than relentless number crunchers. They have the gift of being able to compare values, and this ability is the foundation of computer decision making. In C++, *relational operators* embody this ability. C++ provides six relational operators for comparing numbers. Because characters are represented by their ASCII code, you can use these operators with characters, too. But they don't work with strings. Each rela-

tional expression reduces to the value *1* if the comparison is true and to *0* if the comparison is false, so they are well suited for use in a loop test expression. Table 5-2 summarizes these operators.

Table 5-2 Relational operators

Operator	Meaning
<	is less than
<=	is less than or equal to
==	is equal to
>	is greater than
>=	is greater than or equal to
!=	is not equal to

These six operators exhaust the comparisons C++ lets you make for numbers. If you want to compare two values to see which is the more beautiful or the luckier, you'll have to look elsewhere.

Here are some sample tests:

```
for (x = 20; x > 5; x--) // continue while x is greater than 5
for (x = 1; y != x; x++) // continue while y is not equal to x
for (cin >> c; c == ' '; cin >> c)      // continue while c
                                        //is a space
```

The relational operators have a lower precedence than the arithmetic operators. That means the expression

```
x + 3 > y - 2      // expression 1
```

corresponds to

```
(x + 3) > (y - 2) // expression 2
```

and not the following:

```
x + (3 > y) - 2    // expression 3
```

Because the expression (3 > y) is either 1 or 0, expressions 2 and 3 both are valid. But most of us would want expression 1 to mean expression 2, and that is what C++ does.



THE MISTAKE YOU'LL PROBABLY MAKE

Don't confuse testing the is-equal-to operator (==) with the assignment operator (=). The expression

```
musicians == 4    // comparison
```

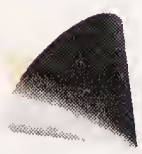
asks the musical question, Is *musicians* equal to *4*? The expression has the value *1* if the comparison is true and *0* if it is false. The expression

```
musicians = 4     // assignment
```

assigns the value *4* to *musicians*. The whole expression, in this case, has the value *4* because that's the value of the left-hand side.

The flexible design of the `for` loop creates an interesting opportunity for error. If you accidentally drop an equal sign(=) from the == operator and use an assignment expression instead of a relational expression for the test part of a `for` loop, you have still produced valid code. That's because you can use *any* valid C++ expression for a `for` loop test condition. Remember, the loop really checks to see if the expression is zero or non-zero, not if it is false or true. So an expression that assigns *4* to *musicians* has the value *4* and is treated as true. If you come from a language such as Pascal or BASIC that uses = to test for equality, you may be particularly prone to this slip.

Listing 5-10 shows a situation in which you could make this sort of error. The program attempts to examine an array of quiz scores and to stop when it reaches the first score that's not a *20*. We show a loop that correctly uses comparison and then one that mistakenly uses assignment in the test condition. The program also has another egregious design error that you'll see how to fix later. (You learn from your mistakes, and Listing 5-10 is happy to help in that respect.)



LISTING 5-10 equal.cpp

```
// equal.cpp -- equality vs assignment
#include <iostream.h>
int main(void)
{
    int quizscores[10] =
            { 20, 20, 20, 20, 20, 19, 20, 18, 20, 20};

// NOTE: some implementations may need to use
//  static int quizscores[10] to enable initialization

   cout << "Doing it right:\n";
   for (int i = 0; quizscores[i] == 20; i++)
```

```
    cout << "quiz " << i << " is a 20\n";

    cout << "Doing it dangerously wrong:\n";
    for (i = 0; quizscores[i] = 20; i++)
    cout << "quiz " << i << " is a 20\n";

    return 0;
}
```

Here is some sample output:

```
Doing it right:
quiz 0 is a 20
quiz 1 is a 20
quiz 2 is a 20
quiz 3 is a 20
quiz 4 is a 20
Doing it dangerously wrong:
quiz 0 is a 20
quiz 1 is a 20
quiz 2 is a 20
quiz 3 is a 20
quiz 4 is a 20
quiz 5 is a 20
quiz 6 is a 20
quiz 7 is a 20
quiz 8 is a 20
quiz 9 is a 20
quiz 10 is a 20
quiz 11 is a 20
quiz 12 is a 20
quiz 13 is a 20
...
```

The first loop correctly halts after displaying the first five quiz scores. But the second starts by displaying the whole array. Worse than that, it says every value is 20. Worse than that, it doesn't stop at the end of the array!

Where things go wrong, of course, is with the following test expression:

```
quizscores[i] = 20
```

First, simply because it assigns a nonzero value to the array element, the expression is always nonzero, hence always true. Second, because the expression assigns values to the array elements, it actually changes the data. Third, because the test expression remains true, the program continues changing data beyond the end of the array. It just keeps

putting more and more 20s into memory! This is not good. In our case, the computer eventually crashed.

Like C, C++ grants you more freedom than most languages in programming. This comes at the cost of requiring greater responsibility on your part. Nothing but your own good planning prevents a program from going beyond the bounds of a standard C++ array. However, with C++ classes, you can design a protected array type that prevents this sort of nonsense. Chapter 11 provides an example. In the meantime, you should build the protection into your programs when needed. For instance, our loop should have included a test that kept it from going past the last member. That's true even for the "good" loop. If all the scores had been *20*s, it, too, would have exceeded the array bounds. In short, the loop needed to test the value of the array *and* test the value of the array index. Chapter 6 will show you how to use logical operators to combine two such tests into a single condition.

COMPARING STRINGS

Suppose you want to see if a string in a character array is the word *mate*. If *word* is the array name, the following test may not do what you think:

```
word == "mate"
```

Remember that the name of an array is a synonym for its address. Similarly, a quoted string constant is a synonym for its address. Thus the above relational expression doesn't test to see if the *strings* are the same—it checks to see if they are stored at the same *address*. And the answer to that is no, even if the two strings have the same characters.

Because C++ handles strings as addresses, you'll get little satisfaction trying to use the relational operators to compare strings. Instead, you can go to the C++ string library and use the `strcmp()` function to compare strings. This function takes two string addresses as arguments. That means the arguments can be pointers, string constants, or character array names. If the two strings are identical, the function returns the value zero. If the first string precedes the second alphabetically, `strcmp()` returns a negative value, and if the first string follows the second alphabetically, `strcmp()` returns a positive value. Actually, "in the system collating sequence" would be more accurate than "alphabetically." This means that characters are compared according to the system code for characters. For instance, in ASCII code, all uppercase letters have smaller codes than the lowercase letters, so uppercase precedes lowercase in the collating sequence. Therefore the string *"Zoo"* precedes the string *"aviary"*. The fact that comparisons are based on code values also means that uppercase and lowercase letters differ, so the string *"FOO"* is different from the *"foo"* string.

In some languages, such as BASIC and standard Pascal, strings stored in differently

sized arrays are necessarily unequal to each other. But in C++ strings are defined by the terminating null character, not by the size of the containing array. That means that two strings can be identical even if contained in differently sized arrays:

```
char big[80] = "Daffy";          // 5 letters plus \0
char little[6] = "Daffy";        // 5 letters plus \0
```

By the way, although you can't use relation operators to compare strings, you can use them to compare characters because characters are really integer types. So

```
for (ch = 'a'; ch <= 'z'; ch++)
    cout << ch;
```

is valid code for displaying the characters of the alphabet.

Listing 5-11 uses `strcmp()` in the test condition of a `for` loop. The program displays a word, changes its first letter, displays the word again, and keeps going until `strcmp()` determines the word is the same as the string *"mate"*. Note that the listing includes the `string.h` file because it provides a function prototype for `strcmp()`.

LISTING 5-11 compstr.cpp

```
// compstr.cpp -- comparing strings
#include <iostream.h>
#include <string.h>    // prototype for strcmp()
int main(void)
{
    char word[5] = "?ate";

// NOTE: some implementations may need to use
//  static char word[5] to enable initialization

    for (char ch = 'a'; strcmp(word, "mate"); ch++)
    {
            cout << word << "\n";
            word[0] = ch;
    }
    cout << "After loop ends, word is " << word << "\n";
    return 0;
}
```

Here is the output:

```
?ate
aate
bate
cate
date
eate
fate
gate
hate
iate
jate
kate
late
After loop ends, word is mate
```

Program Notes

The program has some interesting points. One, of course, is the test. We want the loop to continue as long as *word* is not *mate*. That is, we want the test to continue as long as `strcmp()` says the two strings are not the same. The most obvious test for that is this:

```
strcmp(word, "mate") != 0    // strings are not the same
```

This statement has the value *1* (true) if the strings are unequal and the value 0 (false) if they are equal. But what about `strcmp(word, "mate")` by itself? It has a nonzero value (true) if the strings are unequal and the value *0* (false) if the strings are zero. In essence, the function returns true if the strings are different and false if they are the same. So you can use just the function instead of the whole relational expression. This produces the same behavior and involves less typing. Also, it's the way C and C++ programmers traditionally have used `strcmp()`.

Next, `compstr.cpp` uses the increment operator to march the variable *ch* through the alphabet:

```
ch++
```

You can use the increment and decrement operators with character variables because type `char` really is an integer type, so the operation actually changes the integer code stored in the variable. Also, note that using an array index makes it simple to change individual characters in a string:

```
word[0] = ch;
```

Finally, unlike most of the `for` loops to date, this loop isn't a counting loop. That is, it doesn't execute a block of statements a specified number of times. Instead, the loop watches for a particular circumstance (*word* being *"mate"*) to signal that it's time to stop. More typically, C++ programs use *while* loops for this second kind of test, so let's examine that form now.

THE WHILE LOOP

The `while` loop is a `for` loop stripped of the initialization and update parts; it has just a test condition and a body:

```
while (test-condition)
    body
```

First a program evaluates the *test-condition* expression. If the expression evaluates to a nonzero value (true), the program executes the statement(s) in the body. As with a `for` loop, the *body* consists of a single statement or of a block defined by paired braces. After finishing with the *body*, the program returns to the *test-condition* and reevaluates it. If the condition is nonzero, the program executes the *body* again. This cycle of testing and execution continues until the *test-condition* evaluates to *0* (false). See Figure 5-3. Clearly, if you want the loop to terminate eventually, something within the loop body must do something to affect the *test-condition* expression. For example, the loop could increment a variable used in the test condition or read a new value from keyboard input.

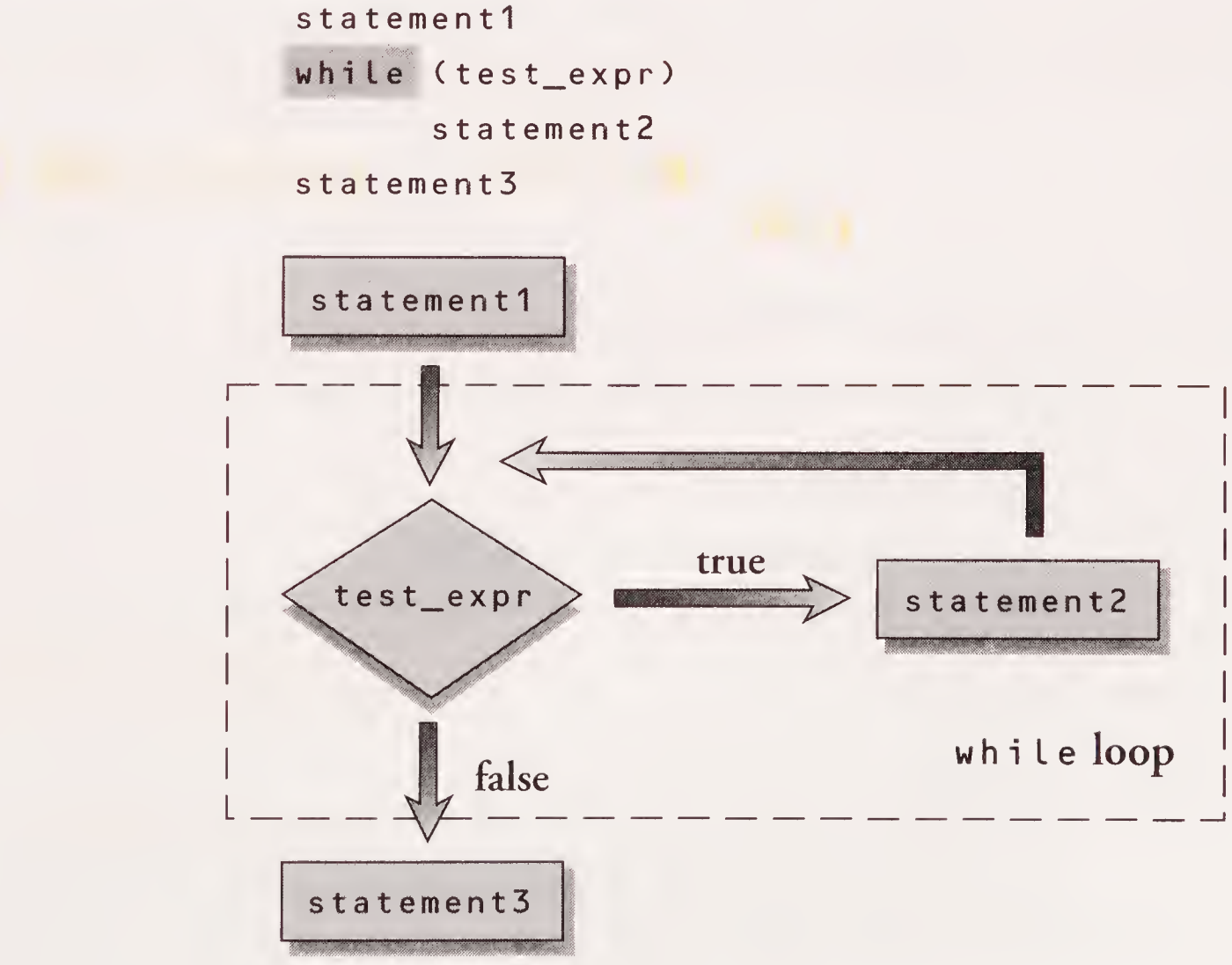


Figure 5-3 *The* `while` *loop*

Like the `for` loop, the `while` loop is an entry-condition loop. Thus if *test-condition* evaluates to *0* to begin with, the program never executes the body of the loop.

Listing 5-12 puts the `while` loop to work. The loop cycles through each character in a string and displays the character and its ASCII code. The loop quits when it reaches the null character. This technique of stepping through a string character by character until reaching the null characters is a standard C++ method for processing strings. Because a string contains its own termination marker, programs often don't need explicit information about how long a string is.

LISTING 5-12 while.cpp

```
// while.cpp -- introducing the while loop
#include <iostream.h>
const int Size = 20;
int main(void)
{
    char name[Size];

    cout << "Your first name, please: ";
    cin >> name;
    cout << "Here is your name, verticalized and ASCIIized:\n";
    int i = 0;                  // start at beginning of string
    while (name[i] != '\0')     // process to end of string
    {
            cout << name[i] << ": " << int(name[i]) << "\n";
            i++;                // don't forget this step
    }
    return 0;
}
```

Here is a sample run:

```
Your first name, please: Muffy
Here is your name, verticalized and ASCIIized:
M: 77
u: 117
f: 102
f: 102
y: 121
```

(No, verticalized and ASCIIized are not real words or even good would-be words. But they do add a technoid tone to the output.)

Program Notes

The `while` condition looks like this:

```
while (name[i] != '\0')
```

It tests whether a particular character in the array is the null character. For this test to eventually succeed, the loop body needs to change the value of *i*. And it does so by incrementing *i* at the end of the loop body. Omitting this step would keep the loop stuck on the same array element, printing the character and its code until you managed to kill the program. Such an *infinite* loop is one of the most common problems with loops. Often it is caused by forgetting to update some value within the loop body.

You can rewrite the `while` line this way:

```
while (name[i])
```

With this change, the program will work just as it did before. That's because when `name[i]` is an ordinary character, its value is the character code, which is nonzero, or true. But when `name[i]` is the null character, its character-code value is *0*, or false. This notation is more concise but less clear than what we used. Dumb compilers may produce faster code for the second version, but smart compilers will produce the same code for both.

To get the program to print the ASCII code for a character, we use a type cast to convert `name[i]` to an integer value. Then `cout` prints the value as an integer instead of interpreting it as a character code.

for vs. while

In C++ the `for` and `while` loops are essentially equivalent. For instance, the `for` loop

```
for (init-expression; test-expression; update-expression)
{
    statement(s)
}
```

could be rewritten this way:

```
init-expression;
while (test-expression)
{
    statement(s)
    update-expression;
}
```

Similarly, the `while` loop

```
while (test-expression)
    body
```

could be rewritten this way:

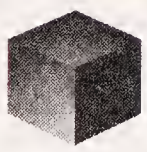
```
for ( ;test-expression;)
    body
```

The `for` loop requires three expressions, but they can be empty expressions. Only the two semicolons are mandatory. Incidentally, a missing test expression in a `for` loop is construed as being true, so the loop

```
for ( ; ; )
    body
```

runs forever.

Because the two loops are nearly equivalent, the one you use is a matter of style. (There is a slight difference if the body includes a `continue` statement, which is discussed in Chapter 6.) Typically, programmers use the `for` loop for counting loops because the `for` loop format lets one place all the relevant information—initial value, terminating value, and method of updating the counter—in one place. And the `while` loop most often is used when you don't know in advance precisely how many times the loop will execute.



BAD PUNCTUATION

Both the `for` loop and the `while` loop have bodies consisting of the *single* statement following the parenthesized expressions. As you've seen, that single statement can be a block, which can contain several statements. Keep in mind that braces, not indentation, define a block. Consider the following loop, for example:

```
i = 0;
while (name[i] != '\0')
    cout << name[i] << "\n";
    i++;
cout << "Done\n";
```

The indentation tells us the program author intended for the `i++;` statement to be part of the loop body. The absence of braces, however, tells the compiler that the body consists solely of the first `cout` statement. Thus the loop keeps printing the first character of the array indefinitely. The program never reaches the `i++;` statement because it is outside the loop.

The next example shows another potential pitfall:

```
i = 0;
while (name[i] != '\0');    // problem semicolon
{
    cout << name[i] << "\n";
    i++;
}
cout << "Done\n";
```

This time we got the braces right, but we also inserted an extra semicolon. Remember, a semicolon terminates a statement, so this statement terminates the `while` loop. In other words, the body of the loop is a *null statement*, that is, nothing followed by a semicolon. All the material in braces now comes *after* the loop. It never gets reached. Instead, the loop cycles doing nothing forever. So beware the straggling semicolon.

JUST A MOMENT

Sometimes it's useful to build a time delay into a program. For instance, you may have encountered programs that flash a message on screen, then go on to something else before you can read it. You're left with the fear that you've missed irretrievable information of vital importance. It would be so much nicer if the program paused five seconds before moving on. The `while` loop is handy for producing this effect. One of the earlier techniques was to make the computer count for a while to use up time:

```
long wait = 0;
while (wait++ < 10000)
    ;                          // counting silently
```

The problem with this approach is that you have to change the counting limit when you change computer processor speed. Several games written for the original IBM PC, for instance, became unmanageably fast when run on its faster successors. A better approach is to let the system clock do the timing for you.

The ANSI C and the C++ libraries have a function to help you do this. The function is called `clock()`, and it returns the system time elapsed since a program started execution. There are a couple of complications. First, `clock()` doesn't necessarily return the time in seconds. Second, the function's return type might be `long` on some systems, `unsigned long` on others, or perhaps some other type.

But the *time.h* header file provides solutions to these problems. First, it defines a symbolic constant, *TCK_CLK* by name, that equals the number of system time units per second. So dividing the system time by this value yields seconds. Or you can multiply seconds by *TCK_CLK* to get time in the system units. Second, `time.h` establishes `clock_t` as an alias for the `clock()` return type. (See the box about Type

Aliases.) This means you can declare a variable as type `clock_t` and the compiler will then convert that to `long` or `unsigned int` or whatever the proper type is for your system.



COMPATIBILITY NOTE

Some C++ implementations may have problems with `waiting.cpp` if the implementation's library component is not fully ANSI C compliant. That's because the `clock()` function is an ANSI addition to the traditional C library.

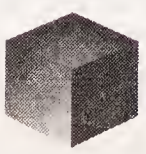
Listing 5-13 shows how to use `clock()` and the `time.h` header to create a time-delay loop.



LISTING 5-13 waiting.cpp

```
// waiting.cpp -- using clock() in a time-delay loop
#include <iostream.h>
#include <time.h> // describes clock() function, clock_t type
int main(void)
{
    cout << "Enter the delay time, in seconds: ";
    float secs; cin >> secs;
    clock_t delay = secs * CLK_TCK;     // convert to clock ticks
    cout << "starting\a\n";
    clock_t start = clock();
    while (clock() - start < delay )    // wait until time elapses
       ;
    cout << "done \a\n";
    return 0;
}
```

By calculating the delay time in system units instead of in seconds, the program avoids having to convert system time to seconds each loop cycle.



TYPE ALIASES

C++ has two ways to establish a new name as an alias for a type. One is to use the preprocessor:

```
#define BYTE char // preprocessor replaces BYTE with char
```

The preprocessor then replaces all occurrences of *BYTE* with `char` when you

compile a program, thus making *BYTE* an alias for `char`.

The second method is to use the C++ (and C) keyword `typedef` to create an alias. For instance, to make *byte* an alias for `char`, do this:

```
typedef char byte;   // makes byte an alias for char
```

Here's the general form:

```
typedef type alias-name;
```

The `typedef` approach is the more powerful, for it can be used with more complex types, such as pointers:

```
typedef char * byte_pointer; // pointer to char type
```

You could try something similar with `#define`, but that won't work if you declare a list of variables. For instance, consider the following:

```
#define FLOAT_POINTER float *
FLOAT_POINTER pa, pb;
```

Preprocessor substitution converts the declaration to this:

```
float * pa, pb;  // pa a pointer to float, pb just a float
```

The `typedef` approach doesn't have that problem.

Notice that *typedef* doesn't create a new type. It just creates a new name for an old type. So if you make `word` an alias for `int`, `cout` will treat a type `word` value as the `int` it really is.

THE DO WHILE LOOP

You've now seen the `for` loop and the `while` loop. The third C++ loop is the `do while`. It's different from the other two because it's an *exit-condition* loop. That means this devil-may-care loop first executes the body of the loop, and only then evaluates the test expression to see if it should continue looping. If the condition evaluates to *0*, the loop terminates; otherwise, a new cycle of execution and testing begins. Such a loop always executes at least once because its program flow has to pass through the body of the loop before reaching the test. Here's the syntax:

```
do
    body
while (test-expression);
```

The *body* portion can be a single statement or a brace-delimited statement block. Figure 5-4 summarizes the program flow for the `do while` loop.

Usually, an entry-condition loop is a better choice than an exit-condition loop because the entry-condition loop checks before looping. For instance, suppose Listing 5-12 had used `do while` instead of `while`. Then the loop would have printed the null character and its code before finding it had already reached the end of the string. But sometimes a `do while` test does make sense. For instance, if you're requesting user input, the program has to obtain the input before testing it. Listing 5-14 shows how to use `do while` in that situation.

```
statement1
do
     statement2
while (test_expr);
statement3
```

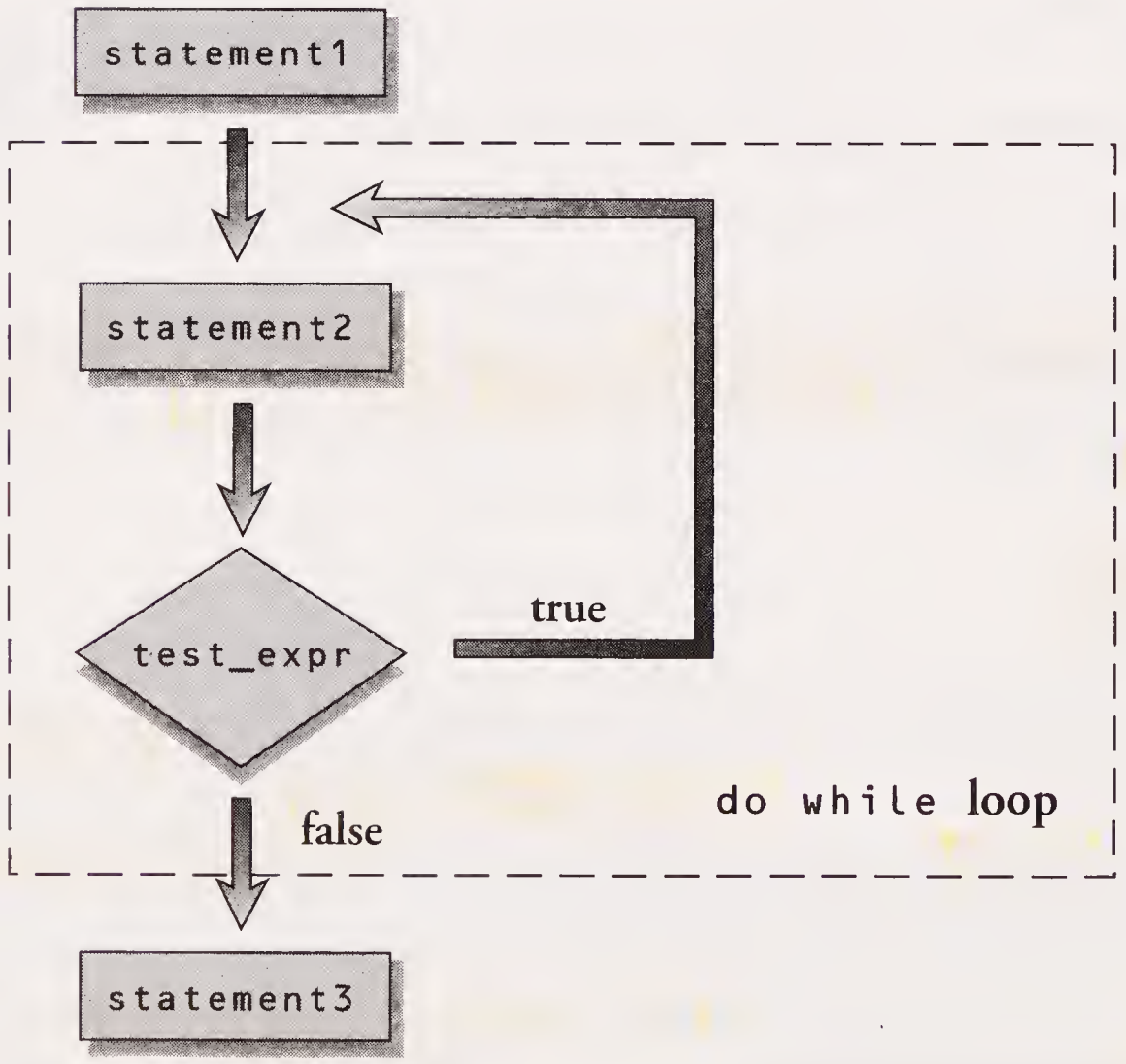
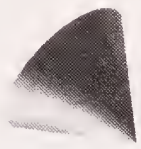


Figure 5-4 *The* `do while` *loop*



LISTING 5-14 dowhile.cpp

```
// dowhile.cpp -- exit-condition loop
#include <iostream.h>
int main(void)
{
    int n;

    cout << "Enter numbers in the range 1-10 to find ";
    cout << "my favorite number\n";
    do
    {
            cin >> n;              // execute body
    } while (n != 7);              // then test
    cout << "Yes, 7 is my favorite.\n" ;
    return 0;
}
```

Here's a sample run:

```
Enter numbers in the range 1-10 to find my favorite number
9
4
7
Yes, 7 is my favorite.
```

LOOPS AND TEXT INPUT

Now that you've seen how loops work, let's look at one of the most common and important tasks assigned to loops: reading text character by character from a file or from the keyboard. For instance, you may wish to write a program that counts the numbers of characters, lines, and words in the input. Traditionally, C++, like C, uses the `while` loop for this sort of task. We'll investigate now how that is done. If you already know C, don't skim through this part too fast. Although the C++ `while` loop is the same as C's, C++'s I/O facilities are different. This can give the C++ loop a somewhat different look. In fact, the `cin` object supports three distinct modes of single-character input, each with a different user interface. So let's look at how to use these choices with `while` loops.

USING UNADORNED CIN FOR INPUT

If a program is going to use a loop to read input from the keyboard, it has to have some way of knowing when to stop. How can it know when to stop? One way is choose some special character, sometimes called a sentinel character, to act as a stop sign. For instance, Listing 5-15 stops reading input when the program encounters a # character. The program counts the number of characters it reads and it *echoes* them. That is, it redisplays the characters that have been read. (Pressing a keyboard key doesn't automatically place a character on the screen; programs have to do that drudge work by echoing the input character.) When finished, it reports the total number of characters processed. Listing 5-15 shows the program.

LISTING 5-15 textin1.cpp

```
// textin1.cpp -- reading chars with a while loop
#include <iostream.h>
int main(void)
{
    char ch;
    int count = 0;          // use basic input
    cin >> ch;              // get a character
    while (ch != '#')       // test the character
    {
            cout << ch;     // echo the character
            count++;        // count the character
            cin >> ch;      // get the next character
    }
    cout << "\n" << count << " characters read\n";
    return 0;
}
```

Here's a sample run:

```
see ken run#really fast
seekenrun
9 characters read
```

Apparently Ken runs so fast, he obliterates space itself—or at least the space characters in the input.

PROGRAM NOTES

First, note the structure. The program reads the first input character *before* it reaches the loop. That way, the first character can be tested when the program reaches the loop

statement. This is important, for the first character may be #. Because `textin1.cpp` uses an entry-condition loop, the program correctly skips the entire loop in that case. And because the variable *count* was set to zero previously, *count* has the correct value.

Suppose the first character read is not a #. Then the program enters the loop, displays the character, increments the count, and reads the next character. This last step is vital. Without it, the loop repeatedly processes the first input character forever. With it, the program advances to the next character.

This all sounds reasonable. So why does the program omit the spaces on output? Blame `cin`. When reading type `char` values, just as when reading other basic types, `cin` skips over spaces and newlines. So the spaces in the input did not get echoed and they did not get counted.

To further complicate things, the input to `cin` is buffered. That means the characters you type don't get sent to the program until you press the Return key or Enter key. This is why we were able to type characters after the #. Once we pressed the Return key, the whole sequence of characters was sent to the program, but the program quit processing the input once it reached the # character.

CIN.GET() TO THE RESCUE

Usually, programs that read input character by character need to examine every character, including spaces, tabs, and newlines. The *istream* class (defined in `iostream.h`), to which `cin` belongs, includes member functions that meet this need. In particular, the member function call `cin.get(ch)` reads the next character, even if it is a space, from the input and assigns it to the variable *ch*. By replacing `cin>>ch` with this function call, we can fix Listing 5-15. Listing 5-16 shows the result.

LISTING 5-16 textin2.cpp

```
// textin2.cpp -- reading chars with a while loop
#include <iostream.h>
int main(void)
{
    char ch;
    int count = 0;

    cin.get(ch);                    // use the cin.get(ch) function
    while (ch != '#')
    {
            cout << ch;
            count++;
            cin.get(ch);            // use it again
```

```
    }
    cout << "\n" << count << " characters read\n";
    return 0;
}
```

Here is a sample run:

```
Did you use a #2 pencil?
Did you use a
14 characters read
```

Now the program echoes and counts *every* character, including the spaces. Input is still buffered, so it's still possible to type more input than what eventually reaches the program.

If you are familiar with C, this program may strike you as being terribly wrong! The `cin.get(ch)` call places a value in the *ch* variable, which means it is altering the value of the variable. In C, you must pass the *address* of a variable to a function if you want to change the value of that variable. But the call to `cin.get()` in Listing 5-16 passes *ch*, not *&ch*. In C, code like this won't work. In C++ it can, providing the function declares the argument as a *reference*. This is a derived type new to C++. The `iostream.h` header file declares the argument to `cin.get(ch)` as a reference type, so this function can alter the value of its argument. We'll get to the details in Chapter 8. Meanwhile, the C mavens among you can relax—ordinarily, argument passing in C++ works just as it does in C. But for `cin.get(ch)` it doesn't.

WHICH CIN.GET()?

Chapter 4 used this code:

```
char name[Size];
...

cout << "Enter your name:\n";
cin.get(name, Size);
```

At that time `cin.get(ch)` took two arguments: the array *name*, which is the address of the string (technically, type `char*`), and *Size*, which is an integer of type `int`. (Recall that the name of an array is the address of its first element, so that makes the name of a character array type `char*`.) But most recently, we've used `cin.get(ch)` this way:

```
char ch;
cin.get(ch);
```

This time `cin.get(ch)` has one argument, and it's type `char`.

Once again it is time for those of you familiar with C to get excited or confused. In C, if a function takes a pointer-to-`char` and an `int` as arguments, you can't successfully

use the same function with a single argument of a different type. But you can do so in C++ because the language supports an OOP feature called *function overloading*. Function overloading allows you to create different functions having the same name providing they have different argument lists. If, for instance, you use `cin.get(name, Size)` in C++, the compiler will find the version of `cin.get()` that uses a `char*` and an `int` as arguments. But if you use `cin.get(ch)`, the compiler fetches the version that uses a single type `char` argument. There is even another version of `cin.get()` that takes no arguments. Function overloading lets you use the same name for related functions that perform the same basic task in different manners or for different types. This is another topic awaiting you in Chapter 8. Meanwhile, you can get accustomed to function overloading by using the examples that come with the *istream* class. To distinguish between the different function versions, we'll include the argument list when referring to them. Thus `cin.get()` means the version that takes no arguments and `cin.get(ch)` means the version that takes one argument.

THE END-OF-FILE CONDITION

As Listing 5-16 showed, using a symbol such as # to signal the end of input is not satisfactory because such a symbol may be part of legitimate input. The same is true of other arbitrarily chosen symbols, such as @ or %. If the input comes from a file, you can employ a much more powerful technique—detecting the end-of-file. C++ input facilities cooperate with the operating system to detect when input reaches the end of a file and to report that information back to a program.

At first glance, reading information from files seems to have little to do with `cin` and keyboard input. But there are two connections. First, many operating systems, including UNIX and MS-DOS, support *redirection*, which lets you substitute a file for keyboard input. For instance, suppose in MS-DOS that you have an executable program called *gofish.exe* and a text file called *fishtale*. Then you can give this command line at the DOS prompt:

```
gofish <fishtale
```

This causes the program to take input from the *fishtale* file instead of from the keyboard. The < symbol is the redirection operator for both UNIX and DOS. Second, many operating systems let you simulate the end-of-file condition from the keyboard. In UNIX you do so by typing Control-D at the beginning of a line. In DOS, you type Control-Z, then the Return key, anywhere on the line.

If your program can test for the end of a file, you can use it with redirected files and you can use it for keyboard input in which you simulate end-of-file. That sounds useful, so let's see how it's done.

The key is using the return value for the `cin.get(ch)` function call. The `cin.get(ch)` function does two things. First, it assigns a value to *ch*; this feature we've

used. Second, and this is a new feature, the function returns a value to the calling program. That is, you can have a statement in the following form:

```
someobject = cin.get(ch);
```

Normally, the `cin.get(ch)` function returns an object of the *istream* class. The main point here is that the object is nonzero. But if the function determines that it has reached the end of the input file, it returns a value of zero instead and does not assign a value to *ch*. In other words, the return value is nonzero (true) if `cin.get(ch)` finds another character and zero (false) if the function finds the end-of-file. This means we can use the `cin.get(ch)` function as a test expression for a `while` loop. That way, the loop keeps reading characters until `cin.get(ch)` returns a zero value. This kind of loop is handy for reading character input, as shown in Listing 5-17.

LISTING 5-17 textin3.cpp

```
// textin3.cpp -- reading chars to end of file
#include <iostream.h>
int main(void)
{
    char ch;
    int count = 0;

    while (cin.get(ch))     // cin.get(ch) is 0 on EOF
    {
            cout << ch;
            count++;
    }
    cout << count << " characters read\n";
    return 0;
}
```

Here is sample output. Because we ran the program on a DOS system, we typed Control-Z, Return to simulate the end-of-file condition. UNIX users would type Control-D instead.

```
The green bird sings in the winter.Return
The green bird sings in the winter.
Yes, but the crow flies in the dawn.Return
Yes, but the crow flies in the dawn.
^ZReturn
73 characters read
```

By using redirection, you can use this program to display a text file and report how many characters it has. This time we'll have a program read, echo, and count a two-line file on a UNIX system (the $ is a UNIX prompt):

```
$ textin3 < stuff
I am a UNIX file. I am proud
to be a UNIX file.
49 characters read
$
```

YET ANOTHER CIN.GET()

The more nostalgic of the C users among you may yearn for C's character I/O functions, `getchar()` and `putchar()`. They are still available if you want them. Just use the `stdio.h` header file as you would in C. Or you can use member functions from the *istream* and *ostream* classes that work much the same way. We'll look at that approach now.

COMPATIBILITY NOTE

At the time of this writing, Zortech C++ doesn't support the `cin.get()` member function (no arguments) discussed here.

The `cin.get()` member function with no arguments returns the next character from the input. That is, you use it in this fashion:

```
ch = cin.get();
```

(Recall that `cin.get(ch)` returns an object, not the character read.) This function works much the same as C's `getchar()`, returning the character code as a type `int` value. Similarly, you can use the `cout.put()` function (see Chapter 3) to display the character:

```
cout.put(ch);
```

It works much like C's `putchar()`; its argument can be type `char` or type `int`.

To use `cin.get()` successfully, you need to know how it deals with the end-of-file condition. When the function reaches the end of a file, there are no more characters to be returned. Instead, `cin.get()` returns a special value represented by the symbolic

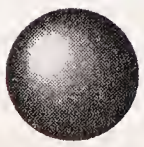
constant *EOF*. This constant is defined in the `iostream.h` header file. The *EOF* value must be different from any valid character value so that the program won't confuse *EOF* with a regular character. Typically, *EOF* is defined as the value *-1* because no character has an ASCII code of -1. But you don't need to know the actual value. Just use *EOF* in the program. For instance, the heart of Listing 5-15 looked like this:

```
cin >> ch;
while (ch != '#')
{
        cout << ch;
        count++;
        cin >> ch;
}
```

We can replace `cin` with `cin.get()`, `cout` with `cout.put()`, and *'#'* with *EOF*:

```
ch = cin.get();
while (ch != EOF)
{
        cout.put(ch);
        count++;
        ch = cin.get();
}
```

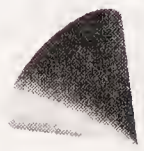
If *ch* is a character, the loop displays it. If *ch* is *EOF*, the loop terminates.



You should realize that *EOF* does not represent a character in the input. Rather, it's a signal that there are no more characters.

There's a subtle but important point about using `cin.get()` beyond the changes made so far. Because *EOF* represents a value outside of the valid character codes, it's possible that it may not be compatible with the `char` type. For instance, on some systems type `char` is unsigned, so a `char` variable could never have the usual *EOF* value of *-1*. For this reason, if you use `cin.get()` (no argument) and test for *EOF*, you need to assign the return value to type `int` instead of type `char`. Also, if you make *ch* type `int` instead of type `char`, you should use `cout.put(ch)` instead of `cout<<ch`. That's because `cout` displays type `int` as an integer, not as a character.

Listing 5-18 incorporates the `cin.get()` approach into a new version of Listing 5-15. It also condenses the code by combining character input with the `while` loop test.



LISTING 5-18 textin4.cpp

```
// textin4.cpp -- reading chars with cin.get()
#include <iostream.h>
```

```
int main(void)
{
    int ch;                                 // should be int, not char
    int count = 0;

    while ((ch = cin.get()) != EOF)         // test for end-of-file
    {
            cout.put(ch);                   // use if ch is type int
            count++;
    }
    cout << count << " characters read\n";
    return 0;
}
```

Here's a sample run:

```
The sullen mackerel sulks in the shadowy shallows.
The sullen mackerel sulks in the shadowy shallows.
Yes, but the blue bird of happiness harbors secrets.
Yes, but the blue bird of happiness harbors secrets.
^Z
104 characters read
```

Let's analyze the loop condition:

```
while ((ch = cin.get()) != EOF)
```

The parentheses enclosing the subexpression `ch=cin.get()` cause the program to evaluate that expression first. To do the evaluation, the program first has to call the `cin.get()` function. Next it assigns the function return value to *ch*. Because the value of an assignment statement is the value of the left operand, the whole subexpression reduces to the value of *ch*. If this value is *EOF*, the loop terminates; otherwise, it continues. The test condition needs all the parentheses. Suppose we leave them out:

```
while (ch = cin.get() != EOF)
```

The `!=` operator has higher precedence than `=`, so first the program compares `cin.get()`'s return value to *EOF*. A comparison produces a *0* (false) or *1* (true) result, and that's the value that gets assigned to *ch*.

Using `cin.get(ch)` (with an argument) for input, on the other hand, doesn't create any type problems. The `cin.get(ch)` function, recall, doesn't assign a special value to *ch* on end-of-file. In fact it doesn't assign anything to *ch* in that case. So *ch* is never called upon to hold a non-*char* value. Table 5-3 summarizes the differences between `cin.get(ch)` and `cin.get()`.

Table 5-3 cin.get(ch) versus cin.get()

PROPERTY	cin.get(ch)	ch=cin.get()
method for conveying input character	assign to argument *ch*	use function return value to assign to *ch*
function return value for character input	a class *istream* object	code for character as type `int` value
function return value at end-of-file	0	*EOF*

Nested Loops and Two-Dimensional Arrays

Earlier you saw how the `for` loop is a natural tool for processing arrays. Let's go a step further and look at how a `for` loop within a `for` loop (*nested* loops) serves to handle two-dimensional arrays.

First, let's examine what a two-dimensional array is. The arrays used so far are termed one-dimensional arrays because you can visualize each array as a single row of data. You can visualize a two-dimensional array as being more like a table, having both rows and columns of data. You could use a two-dimensional array, for instance, to represent quarterly sales figures for six separate districts, with one row of data for each district. Or you could use a two-dimensional array to represent the position of RoboDork on a computerized game board.

C++ doesn't provide a special two-dimensional array type. Instead, you create an array for which each element is itself an array. For example, suppose you want to store maximum temperature data for five cities over a four-year period. In that case you can declare an array as follows:

```
int maxtemps[4][5];
```

This declaration means that *maxtemps* is an array with four elements. Each of these elements is an array of five integers. See Figure 5-5. You can think of the *maxtemps* array as representing four rows of five temperature values each.

The expression `maxtemps[0]` is the first element of the *maxtemps* array, hence `maxtemps[0]` is itself an array of five `int`s. The first element of the `maxtemps[0]` array is `maxtemps[0][0]`, and this element is a single `int`. Thus you need to use two subscripts to access the `int` elements. You can think of the first subscript as representing the row and the second subscript as representing the column. See Figure 5-6.

maxtemps is an array of 4 elements

```
int maxtemps[4][5];
```

each element is an array of 5 ints

The *maxtemps* array

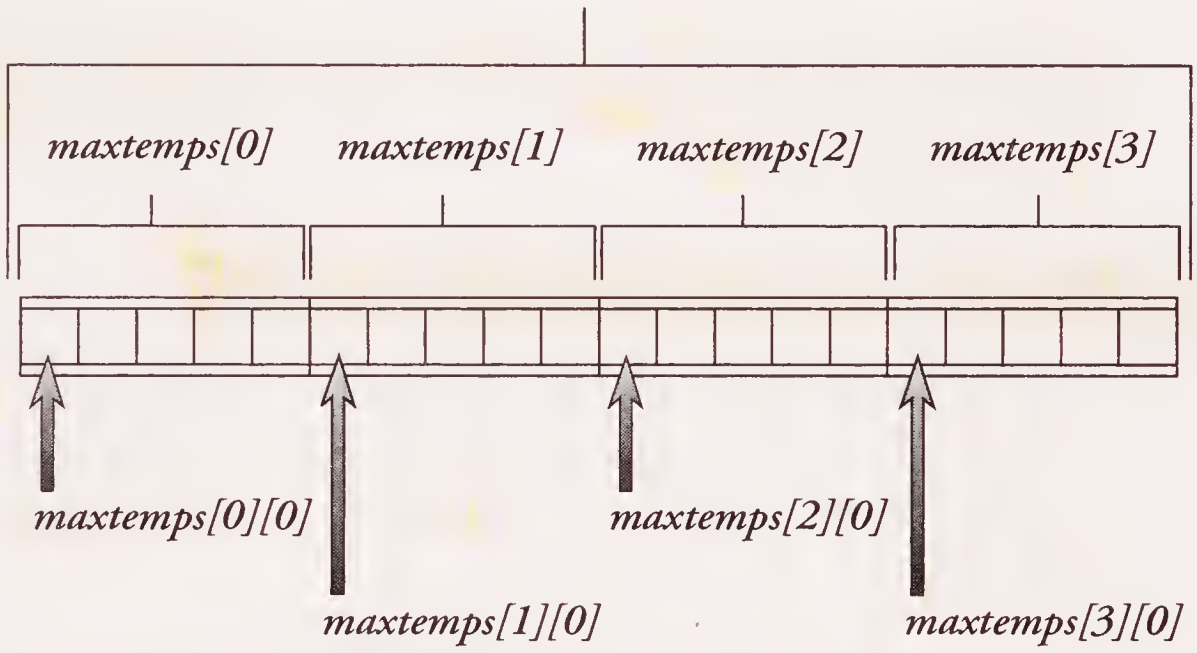


Figure 5-5 *An array of arrays*

```
int maxtemps[4][5];
```

The *maxtemps* array viewed as a table:

		0	1	2	3	4
maxtemps[0]	0	maxtemps[0][0]	maxtemps[0][1]	maxtemps[0][2]	maxtemps[0][3]	maxtemps[0][4]
maxtemps[1]	1	maxtemps[1][0]	maxtemps[1][1]	maxtemps[1][2]	maxtemps[1][3]	maxtemps[1][4]
maxtemps[2]	2	maxtemps[2][0]	maxtemps[2][1]	maxtemps[2][2]	maxtemps[2][3]	maxtemps[2][4]
maxtemps[3]	3	maxtemps[3][0]	maxtemps[3][1]	maxtemps[3][2]	maxtemps[3][3]	maxtemps[3][4]

Figure 5-6 *Accessing array elements with subscripts*

Suppose you want to print all the array contents. Then you can use one `for` loop to change rows and a second, nested, `for` loop to change columns:

```
for (int row = 0; row < 4; row++)
{
    for (int col = 0; col < 5; col++)
            cout << maxtemps[row][col] << "\t";
    cout << "\n";
}
```

For each value of *row*, the inner `for` loop cycles through all the *col* values. This example prints a tab character (`\t` in C++ notation) after each value and a newline character after each complete row.

INITIALIZING A TWO-DIMENSIONAL ARRAY

When you create a two-dimensional array, you have the option of initializing each element. The technique is based on that for initializing a one-dimensional array. That method, you remember, is to provide a comma-separated list of values enclosed in braces:

```
// initializing a one-dimensional array
int btus[5] = { 23, 26, 24, 31, 28};
```

For a two-dimensional array, each element is itself an array, so you can initialize each element using a form like that above. Thus the initialization consists of a comma-separated series of one-dimensional initializations all enclosed in a set of braces:

```
int maxtemps[4][5] =                    // 2-D array
{
    {94, 98, 87, 103, 101},             // values for maxtemps[0]
    {98, 99, 91, 107, 105},             // values for maxtemps[1]
    {93, 91, 90, 101, 104},             // values for maxtemps[2]
    {95, 100, 88, 105, 103}             // values for maxtemps[3]
};
```

The term `{94, 98, 87, 103, 101}` initializes the first row, represented by `maxtemps[0]`. As a matter of style, placing each row of data on its own line, if possible, makes the data easier to read.

Listing 5-19 incorporates an initialized two-dimensional array and a nested loop into a program. This time we reverse the order of the loops, placing the column loop (*city* index) on the outside and the row loop (*year* index) on the inside. Also, we use a common C++ practice of initializing an array of pointers to a set of string constants. That is, *cities* is declared as an array of pointers-to-`char`. That makes each element, such as *cities[0]*, a pointer-to-`char` that can be initialized to the address of a string. The pro-

gram initializes *cities[0]* to the address of the *"Gribble City"* string, and so on. Thus this array of pointers essentially is an array of strings.

LISTING 5-19 nested.cpp

```
// nested.cpp -- nested loops and 2-D array
#include <iostream.h>

const int Cities = 5;
const int Years = 4;
int main(void)
{
    char * cities[Cities] =                 // array of pointers
    {                                       // to 5 strings
            "Gribble City",
            "Gribbleton",
            "New Gribble",
            "San Gribble",
            "Gribble Vista"
    };

    int maxtemps[Years][Cities] =           // 2-D array
    {
            {94, 98, 87, 103, 101},         // values for maxtemps[0]
            {98, 99, 91, 107, 105},         // values for maxtemps[1]
            {93, 91, 90, 101, 104},         // values for maxtemps[2]
            {95, 100, 88, 105, 103}         // values for maxtemps[3]
    };

    cout << "Maximum temperatures for 1990 - 1993\n\n";

    for (int city = 0; city < Cities; city++)
    {
            cout << cities[city] << ":\t";
            for (int year = 0; year < Years; year++)
                    cout << maxtemps[year][city] << "\t";
            cout << "\n";
    }

    return 0;
}
```

Here is the program output:

```
Maximum temperatures for 1990 - 1993
```

```
Gribble City:    94     98     93     95
Gribbleton:      98     99     91     100
New Gribble:     87     91     90     88
San Gribble:     103    107    101    105
Gribble Vista:   101    105    104    103
```

Using tabs in the output spaced the data more regularly than using spaces would have. Chapter 12 presents more precise, but more complex, methods for formatting output.

SUMMARY

C++ offers three varieties of loops: the `for` loop, the `while` loop, and the `do while` loop. A loop cycles through the same set of instructions repetitively as long as the loop test condition evaluates to nonzero, and the loop terminates execution when the test condition evaluates to zero. The `for` loop and the `while` loop are entry-condition loops, meaning they examine the test condition before executing the statements in the body of the loop. The `do while` loop is an exit-condition loop, meaning it examines the test condition after executing the statements in the body of the loop.

The syntax for each loop calls for the loop body to consist of a single statement. However, that statement can be a compound statement, or block, formed by enclosing several statements within paired curly braces.

Relational expressions, which compare two values, are often used as loop test conditions. Relational expressions are formed by using one of the six relational operators: `<`, `<=`, `==`, `>=`, `>`, `!=`. Relational expressions evaluate to *1* if true and to *0* if false.

Many programs read text input or text files character by character. The *istream* class provides several ways to do this. If *ch* is a type `char` variable, the statement

```
cin >> ch;
```

reads the next input character into *ch*. However, it skips over spaces, newlines, and tabs. The member function call

```
cin.get(ch);
```

reads the next input character, regardless of its value, and places it in *ch*. The member function call `cin.get()` returns the next input character, including spaces, newlines, and tabs, so it can be used as follows:

```
ch = cin.get();
```

The `cin.get(char)` member function call reports encountering the end-of-file condition by returning the value *0*, while the `cin.get()` member function call reports end-of-file by returning the value *EOF*, which is defined in the `iostream.h` file.

A nested loop is a loop within a loop. Nested loops provide a natural way to process two-dimensional arrays.

Review Questions

1. What's the difference between an entry-condition loop and an exit-condition loop? Which kind is each of the C++ loops?

2. What would the following code fragment print if it were part of a valid program?

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

3. What would the following code fragment print if it were part of a valid program?

```
int j;
for (j = 0; j < 11; j += 3)
      cout << j;
cout << "\n" << j << "\n";
```

4. What would the following code fragment print if it were part of a valid program?

```
int j = 5;
while ( ++j < 9)
      cout << j++ << "\n";
```

5. What would the following code fragment print if it were part of a valid program?

```
int k = 8;
do
      cout <<" k = " << k << "\n";
while (k++ < 5);
```

6. Write a `for` loop that prints the values 1 2 4 8 16 32 64 by increasing the value of a counting variable by a factor of 2 each cycle.

7. How do you make a loop body include more than one statement?

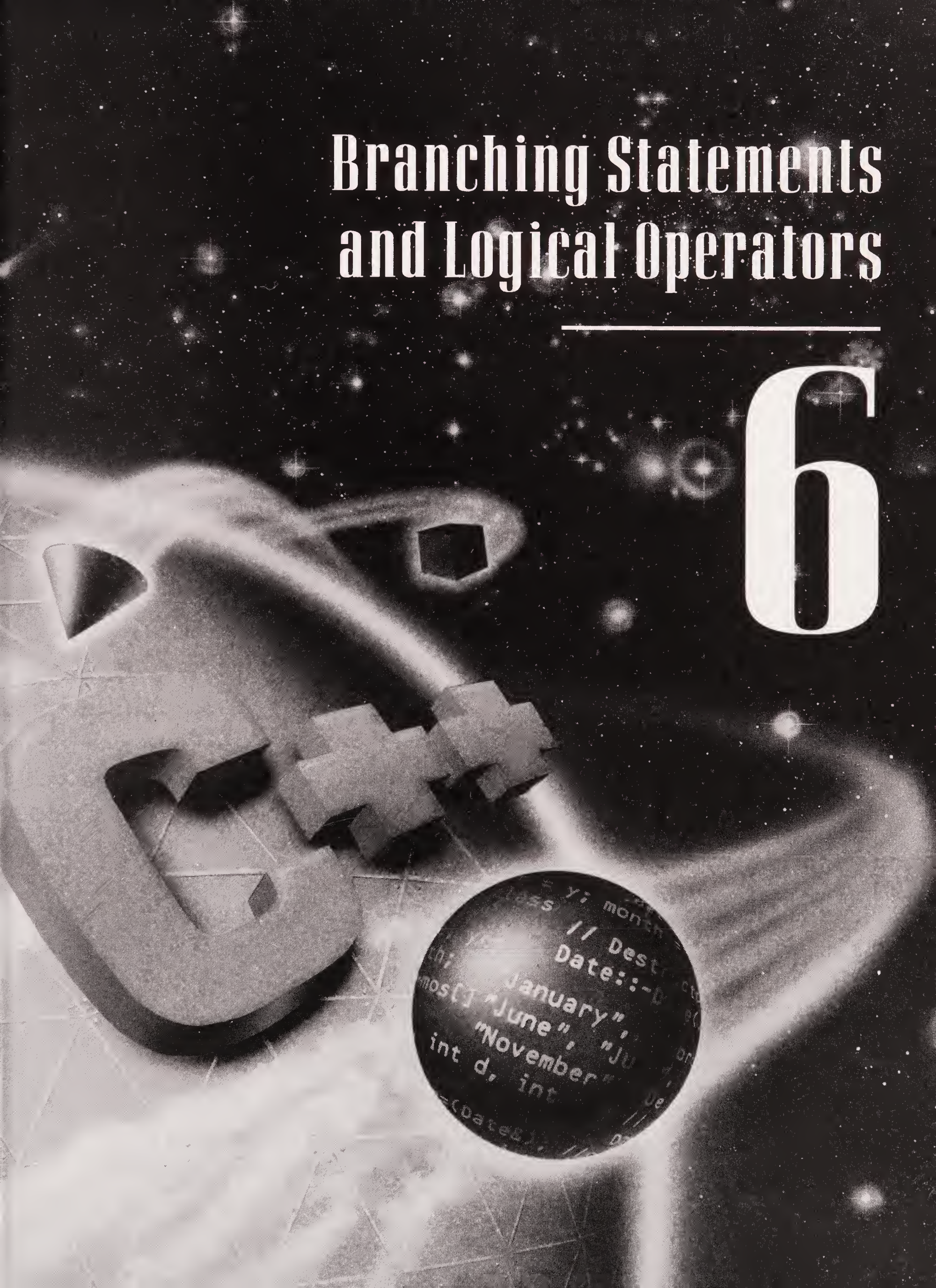
8. Is the following statement valid? If not, why not? If so, what does it do?

```
int x = 1,024;
```

9. How does `cin>>ch` differ from `cin.get(ch)` and `ch=cin.get()` in how it views input?

Branching Statements and Logical Operators

6



One of the keys to designing intelligent programs is to make them able to make decisions. In Chapter 5 you saw one kind of decision-making—looping—in which a program decides whether or not to continue looping. Now we'll investigate how C++ lets you use branching statements to decide among alternative actions. Which vampire-protection scheme (garlic or cross) should the program use? Which menu choice has the user selected? Did the user enter a zero? C++ provides the `if` and the `switch` statements for implementing decisions, and they are this chapter's main topics. We'll also look at the conditional operator, which provides another way to make a choice, and at the logical operators, which let you combine two tests into one.

The if Statement

When a C++ program must choose whether or not to take a particular action, you can use the `if` statement. The `if` comes in two forms: `if` and `if else`. We'll investigate the simple `if` first. It's modeled after ordinary English, as in "If you have a Captain Cookie card, you get a free cookie." The `if` statement directs a program to execute a statement or statement block if a test condition is true and to skip that statement or block if the condition is false. The syntax is similar to the `while` syntax:

```
if (test-condition)
    statement
```

A nonzero *test-condition* (true) causes the program to execute *statement*, which can be a single statement or a block. A zero *test-condition* (false) causes the program to skip *statement*. See Figure 6-1. The entire `if` construction counts as a single statement. By the way, the *statement* portion can't be a declaration statement; declarations have to be placed where they can't be skipped.

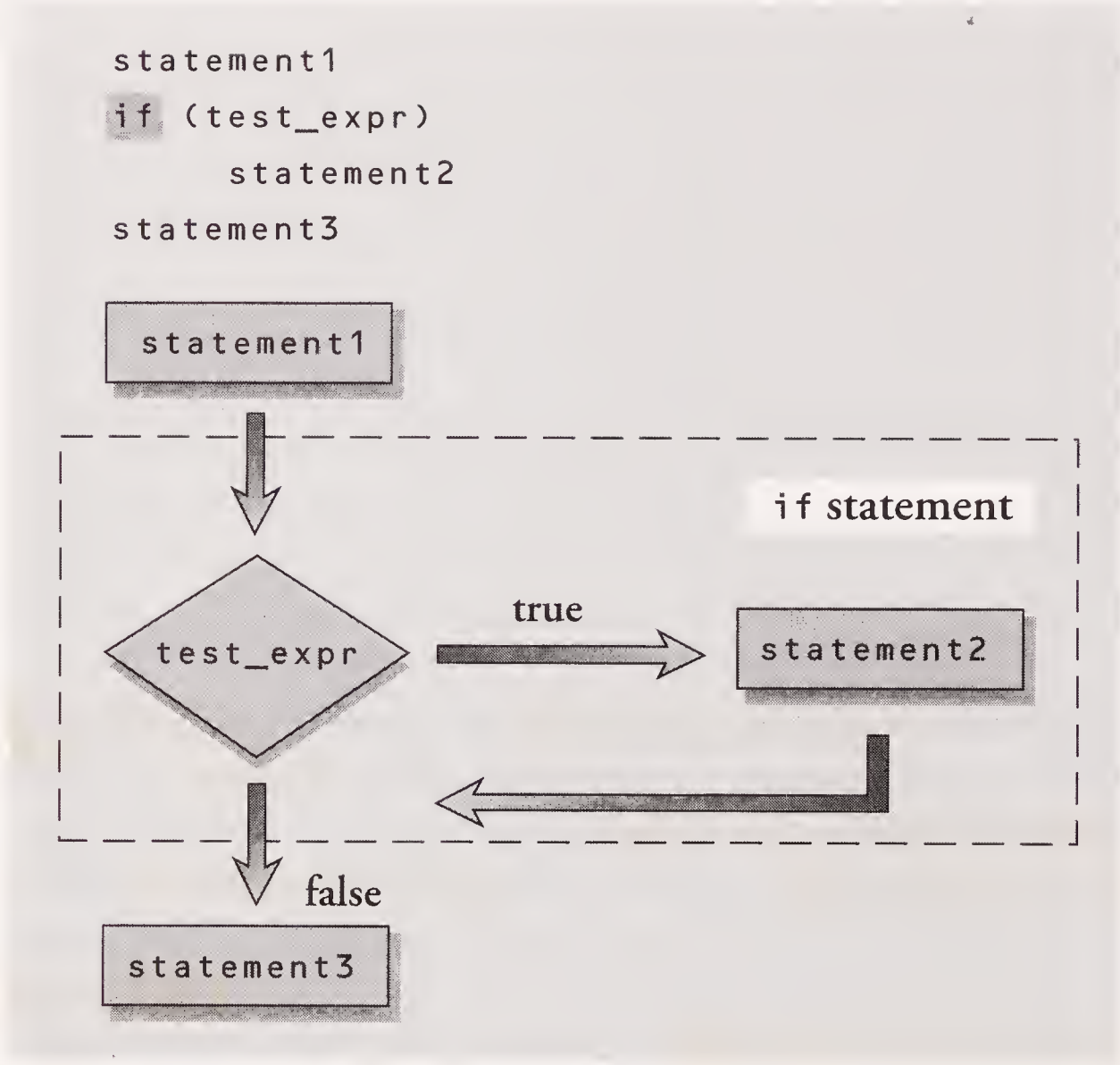
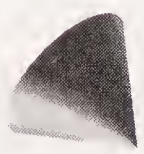


Figure 6-1 *The* `if` *statement*

Most often, *test-condition* will be a relational expression like those used to control loops. Suppose, for instance, you want a program that counts the spaces in the input as well as the total number of characters. You can use `cin.get(ch)` in a `while` loop to read the characters, then use an `if` statement to identify and count the space characters. Listing 6-1 does just that. (Remember that `cin.get(ch)` normally returns `cin`, which is nonzero, causing the loop to continue. But it returns zero when the program detects end of file, thus terminating the loop.)



LISTING 6-1 if.cpp

```
// if.cpp -- using the if statement
#include <iostream.h>
int main(void)
{
    char ch;
```

```
    int spaces = 0;
    int total = 0;

    while (cin.get(ch))
    {
            if (ch == ' ')          // check if ch is a space
                spaces++;
            total++;                        // done every time
    }
    cout << spaces << " spaces, " << total;
    cout << " characters total\n";
    return 0;
}
```

Here's some sample output (again, we've used RETURN to indicate pressing the Return or Enter key and ^Z to simulate end-of-file):

```
The balloonist was an airhead.RETURN
^ZRETURN
4 spaces, 31 characters total
```

As the comments indicate, the `spaces++;` statement gets executed only when *ch* is a space. Because it is outside the `if` statement, the `total++;` statement is executed every time. Note that the total count includes the newline character generated by typing the Return or Enter key.

The if else Statement

The `if` statement lets a program decide whether a particular statement or block is executed. The `if else` statement lets a program decide which of two statements or blocks is executed. It's an invaluable statement for creating alternative courses of action. The C++ `if else` is modeled after simple English, as in "If you have a Captain Cookie card, you get a Cookie Plus Plus, else you just get a Cookie d'Ordinaire." The `if else` statement has this general form:

```
if (test-condition)
    statement1
else
    statement2
```

If *test-condition* is nonzero (true), the program executes *statement1* and skips over *statement2*. Otherwise, when *test-condition* is zero (false), the program skips *statement1* and executes *statement2* instead. So the code fragment

```
if (answer == 1492)
    cout << "That's right!\n";
else
    cout << "You'd better review Chapter 1 again.\n";
```

prints the first message if *answer* is *1492* and prints the second message otherwise. Each statement can be either a single statement or a statement block delimited by braces. See Figure 6-2. The entire `if else` construct counts syntactically as a single statement.

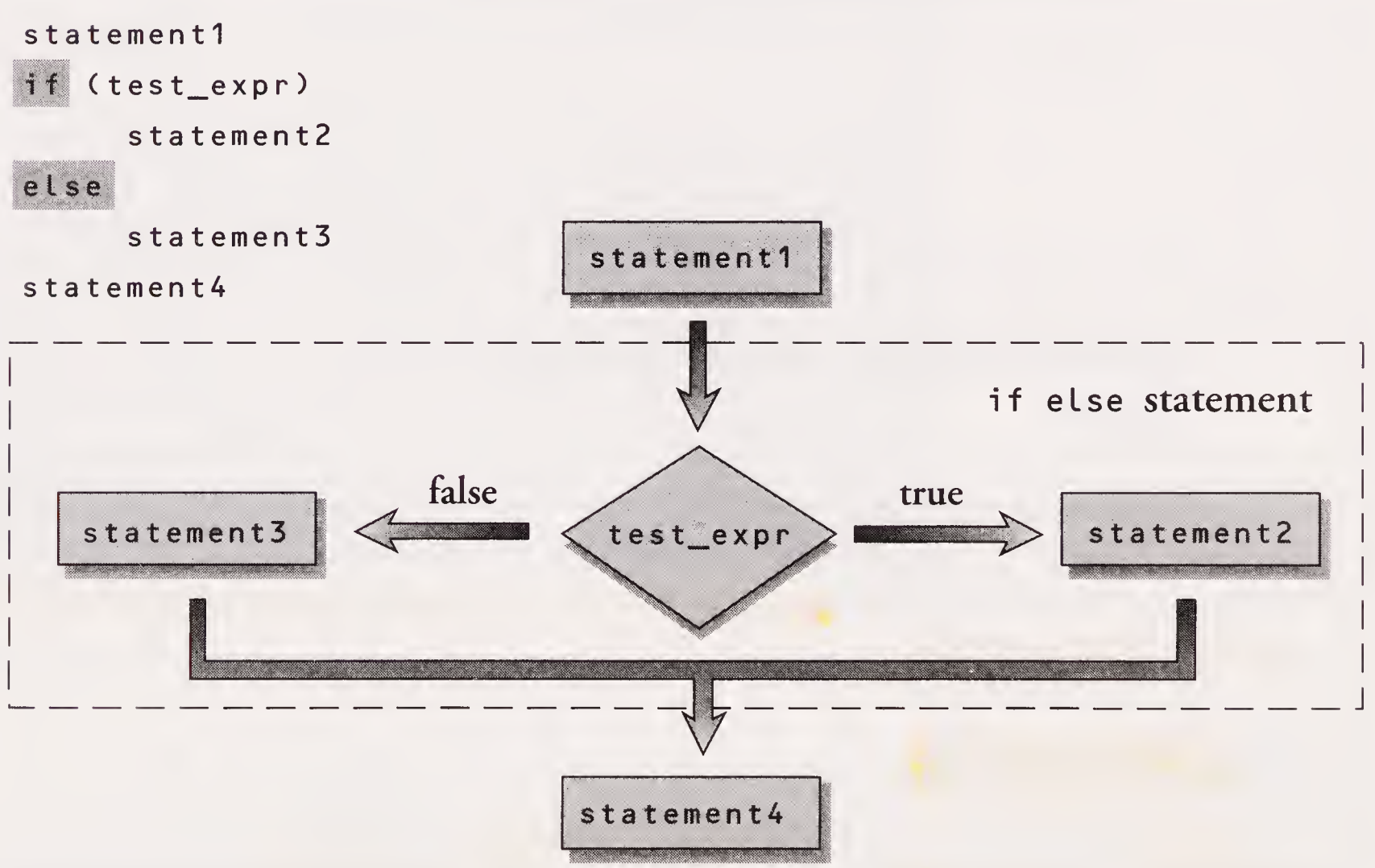
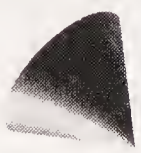


Figure 6-2 *The* `if else` *statement*

For instance, suppose you want to alter incoming text by scrambling the letters while keeping the newline character intact. That way, each line of input gets converted to an output line of equal length. This means you want the program to take one course of action for newline characters and a different course of action for all other characters. As Listing 6-2 shows, `if else` makes this task easy.



LISTING 6-2 ifelse.cpp

```
// ifelse.cpp -- using the if else statement
#include <iostream.h>
```

```
int main(void)
{
    char ch;

    cout << "Type, and I shall repeat.\n";
    while (cin.get(ch))
    {
            if (ch == '\n')
                  cout << ch;    // done if newline
            else
                  cout << ++ch;  // done otherwise
    }
    // try ch + 1 instead of ++ch for interesting effect
    cout << "Please excuse the slight confusion.\n";
    return 0;
}
```

Here's some sample output:

```
Type, and I shall repeat.
I am extremely pleased to
J!bn!fyusfnfmz!qmfbtfe!up
use such a powerful computer!
vtf!tvdi!b!qpxfsgvm!dpnqvufs"
^Z
Please excuse the slight confusion.
```

Note that one of the program comments suggests that changing `++ch` to `ch+1` has an interesting effect. Can you deduce what it will be? If not, try it out, then see if you can explain what's happening. (Hint: Think about how `cout` handles different types.)

FORMATTING YOUR IF ELSE STATEMENTS

Keep in mind that the two `if else` alternatives must be single statements. If you need more than one statement, use braces to collect them into a single block statement. Unlike some languages, such as BASIC or FORTRAN, C++ does not automatically consider everything between `if` and `else` a block, so you have to use braces to make the statements a block. The following code, for instance, produces a compiler error. The compiler sees it as a simple `if` statement ending with the `zorro++;` statement. Then comes a `cout` statement. So far, so good. But then comes what the compiler perceives as an unattached `else`. And that gets flagged as a syntax error.

```
if (ch == 'Z')
```

```
    zorro++;                    // if ends here
    cout << "Another Zorro candidate\n";
else                            // wrong
    dull++;
    cout << "Not a Zorro candidate\n";
```

Adding the braces converts the code to what we want:

```
if (ch == 'Z')
{                               // if true block
    zorro++;
    cout << "Another Zorro candidate\n";
}
else
{                               // if false block
    dull++;
    cout << "Not a Zorro candidate\n";
}
```

Because C++ is a free-form language, you can arrange the braces as you like as long as they enclose the statements. We've shown one popular format. Here's another:

```
if (ch == 'Z') {
    zorro++;
    cout << "Another Zorro candidate\n";
    }
else {
    dull++;
    cout << "Not a Zorro candidate\n";
    }
```

The first form emphasized the block structure for the statements while the second form ties the blocks more closely to the keywords `if` and `else`. Either style should serve you well unless you encounter a passionate advocate for some particular style.

THE IF ELSE IF ELSE CONSTRUCTION

Computer programs, like life, may present you with a choice from more than two selections. You can extend the C++ `if else` statement to meet that need. The `else`, you've seen, should be followed by a single statement, which can be a block. Because an `if else` statement itself is a single statement, it can follow an `else`:

```
if (ch == 'A')
    a_grade++;                  // alternative # 1
else
    if (ch == 'B')              // alternative # 2
        b_grade++;              // subalternative # 2a
```

```
    else
            soso++;                 // subalternative # 2b
```

If *ch* is not *'A'*, then the program goes to the `else`. There, a second `if else` subdivides that alternative into two more choices. C++'s free formatting lets you arrange these elements into an easier-to-read format:

```
if (ch == 'A')
    a_grade++;                      // alternative # 1
else if (ch == 'B')
    b_grade++;                      // alternative # 2
else
    soso++;                         // alternative # 3
```

This looks like a new control structure—an `if else if else` structure. But it's really one `if else` contained within a second. This revised format is much cleaner looking, and it lets you skim through the code picking out the different alternatives. This entire construction still counts as one statement.

Listing 6-3 uses this preferred formatting to construct a modest quiz program.

LISTING 6-3 ifelseif.cpp

```
// ifelseif.cpp -- using if else if else
#include <iostream.h>
const int Fave = 27;
int main(void)
{
    int n;

    cout << "Enter a number in the range 1-100 to find ";
    cout << "my favorite number: ";
    do
    {
            cin >> n;
            if (n < Fave)
                    cout << "Too low -- guess again: ";
            else if (n > Fave)
                    cout << "Too high -- guess again: ";
            else
                    cout << Fave << " is right!\n";
    } while (n != Fave);
    return 0;
}
```

Here's some sample output:

```
Enter a number in the range 1-100 to find my favorite number: 50
Too high -- guess again: 25
Too low -- guess again: 37
Too high -- guess again: 31
Too high -- guess again: 28
Too high -- guess again: 27
27 is right!
```

LOGICAL EXPRESSIONS

Many a time you will need to test for more than one condition. For instance, for a character to be a lowercase letter, its value must be greater than or equal to *'a'* and less than or equal to *'z'*. Or, if you ask a user to respond with a *'y'* or an *'n'*, you will want to accept uppercase (*'Y'* and *'N'*) as well as lowercase. To meet this kind of need, C++ provides three *logical* operators to combine or modify existing expressions. The operators are logical OR, written ||, logical AND, written `&&`, and logical NOT, written `!`. Let's examine them now.

THE LOGICAL OR OPERATOR: ||

In English, we use the word *or* to indicate when one or another, or both, of two conditions satisfy a requirement. For instance, you can go to the MegaMicro company picnic if you *or* your spouse work for MegaMicro, Inc. The C++ equivalent is the logical OR operator, written ||. This operator combines two expressions into one. If either, or both, of the original expressions is nonzero (true), the resulting expression has the value 1 (true). Here are some examples:

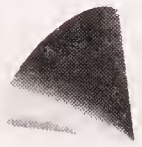
```
5 ==5 || 5 == 9   // true because first expression is true
5 > 3 || 5 > 10   // true because first expression is true
5 > 8 || 5 < 10   // true because second expression is true
5 < 8 || 5 > 2    // true because both expressions are true
5 > 8 || 5 < 2    // false because both expressions are false
```

Because the || has a lower precedence than the relational operators, we don't need to use parentheses in these expressions. Table 6-1 summarizes how the || operator works.

Listing 6-4 uses the || operator in an `if` statement to check for both uppercase and lowercase versions of a character. Also, it uses C++'s string concatenation feature (see Chapter 4) to spread a single string over three lines.

Table 6-1 The || operator

THE VALUE OF EXPR1 || EXPR2		
	expr1 == true	expr1== false
expr2 == true	1 (TRUE)	1 (TRUE)
expr2 == false	1 (TRUE)	0 (FALSE)



LISTING 6-4 or.cpp

```
// or.cpp -- use logical OR operator
#include <iostream.h>
int main(void)
{
    cout << "This program may reformat your hard disk\n"
                "and destroy all your data.\n"
                "Do you wish to continue? <y/n> ";
    char ch;
    cin >> ch;
    if (ch == 'y' || ch == 'Y')             // y or Y
            cout << "You were warned!\a\a\n";
    else if (ch == 'n' || ch == 'N')
            cout << "A wise choice ... bye\n";
    else
            cout << "That wasn't a y or an n, so I guess I'll "
                        "trash your disk anyway.\n";
    return 0;
}
```

Here is a sample run:

```
This program may reformat your hard disk
and destroy all your data.
Do you wish to continue? <y/n> N
A wise choice ... bye
```

The program reads just one character, so only the first character in the response matters. That means the user could have replied *NO!* instead of *N*. The program would just read the *N*. But if the program tried to read more input later, it would start at the *O*.

THE LOGICAL AND OPERATOR: &&

The logical AND operator, written `&&`, also combines two expressions into one. The resulting expression has the value *1* (true) only if both of the original expressions are true. Here are some examples:

```
5 == 5 && 4 == 4   // true because both expressions are true
5 == 3 && 4 == 4   // false because first expression is false
5 > 3 && 5 > 10    // false because second expression is false
5 > 8 && 5 < 10    // false because first expression is false
5 < 8 && 5 > 2     // true because both expressions are true
5 > 8 && 5 < 2     // false because both expressions are false
```

Because the `&&` has a lower precedence than the relational operators, we don't need to use parentheses in these expressions. Table 6-2 summarizes how the `&&` operator works.

Table 6-2 The && operator

THE VALUE OF EXPR1 && EXPR2		
	expr1 == true	expr1 == false
expr2 == true	1 (TRUE)	0 (FALSE)
expr2 == false	0 (FALSE)	0 (FALSE)

Listing 6-5 shows how to use `&&` to cope with a common situation: terminating a `while` loop for two different reasons. In the listing, a `while` loop reads values into an array. One test (*i* < *Size*) terminates the loop when the array is full. The second test (*temp* >= *0*) gives the user the option of quitting early by entering a negative number. The `&&` operator lets you combine the two tests into a single condition. The program also uses two `if` statements, an `if else` statement, and a `for` loop, so it demonstrates several topics from this and the preceding chapter.



LISTING 6-5 and.cpp

```
// and.cpp -- use logical AND operator
#include <iostream.h>
const int Size = 6;
int main(void)
{
    float iq[Size];
```

```
cout << "Enter the IQs of your in-laws. Program term"
        "inates when you make\n" << Size << " entries ";
cout << "or enter a negative value.\n";

int i = 0;
float temp;
cin >> temp;
while (i < Size && temp >= 0)        // 2 quitting criteria
{
        iq[i++] = temp;
        if (i < Size)                // room left in the array,
                cin >> temp;         // so get next value
}
if (i == 0)
        cout << "No data--bye\n";
else
{
        cout << "Enter your IQ: ";
        float you;
        cin >> you;
        int count = 0;
        for (int j = 0; j < i; j++)
                if (iq[j] > you)
                        count++;
            cout << count;
            cout << " of your in-laws are smarter than you.\n";
    }
    return 0;
}
```

Note that the program places input into a temporary variable *temp*. Only after verifying that the input is valid does the program assign the value to the array.

Here are a couple of sample runs. One terminates after six entries, and the second terminates after a negative value is entered:

```
Enter the IQs of your in-laws. Program terminates when you make
6 entries or enter a negative value.
120 140 115
85
123
100
Enter your IQ: 122
2 of your in-laws are smarter than you.

Enter the IQs of your in-laws. Program terminates when you make
6 entries or enter a negative value.
```

```
140
120
75
110
-3
Enter your IQ: 212.19
0 of your in-laws are smarter than you.
```

Program Notes

Let's look at the input part of the program:

```
cin >> temp;
while (i < Size && temp >= 0)  // 2 quitting criteria
{
    iq[i++] = temp;
    if (i < Size)                          // room left in the array,
            cin >> temp;                   // so read next value
}
```

The program begins by reading the first input value into a temporary variable called *temp*. Then the `while` test condition checks to see if there is still room left in the array (*i* < *Size*) and if the input value is nonnegative (*temp* >= *0*). If so, it copies the *temp* value to the array and increases the array index by 1. At this point, since array numbering starts at 0, *i* equals the total number of entries to date. That is, if *i* starts out at 0, then the first cycle through the loop assigns a value to *iq[0]*, then sets *i* to 1.

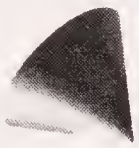
The loop terminates when the array is filled or when the user enters a negative number. Note that the loop reads another value into *temp* only if *i* is less than *Size*, that is, only if there is still room left in the array.

After getting data, the program uses an *if else* statement to comment if no data were entered (that is, if the first entry was a negative number) and to process the data if any is present.

Setting Up Ranges With &&

The `&&` operator also lets you set up a series of `if else if else` statements with each choice corresponding to a particular range of values. Listing 6-6 illustrates the approach. It also shows a useful technique for handling a series of messages. Just as a pointer-to-`char` variable can identify a string by pointing to its beginning, an array of pointers-to-`char` can identify a series of strings. Just assign the address of each string to a different array element. Listing 6-6 uses the *qualify* array to hold the addresses of four strings. For instance, *qualify[1]* holds the address of the string *"mud tug-of-war\n"*. The program can

then use *qualify[1]* like any other pointer to a string, for example, with `cout` or with `strlen()` or `strcmp()`. Using the *const* qualifier protects these strings from accidental alterations.

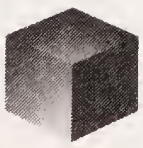


LISTING 6-6 more_and.cpp

```
// more_and.cpp -- use logical AND operator
#include <iostream.h>
const char * qualify[4] =       // an array of pointers
{                               // to strings
    "10,000-meter race.\n",
    "mud tug-of-war.\n",
    "masters canoe jousting.\n",
    "pie-throwing festival.\n"
};
int main(void)
{
    int age;
    cout << "Enter your age in years: ";
    cin >> age;
    int index;

    if (age > 17 && age < 35)
            index = 0;
    else if (age >= 35 && age < 50)
            index = 1;
    else if (age >= 50 && age < 65)
            index = 2;
    else
            index = 3;

    cout << "You qualify for the " << qualify[index];
    return 0;
}
```



COMPATIBILITY NOTE

You may recall that some C++ implementations require that you use the keyword `static` in an array declaration in order to make it possible to initialize that array. That restriction, as Chapter 8 will discuss, applies to arrays declared inside a function body. When an array is declared outside a function body, as is *qualify* in Listing 6-6, it's termed an *external* array and can be initialized even in pre-ANSI C implementations.

Here is a sample run:

```
Enter your age in years: 84
You qualify for the pie-throwing festival.
```

The entered age didn't match any of the test ranges, so the program set *index* to *3* and then printed the corresponding string.

Program Notes

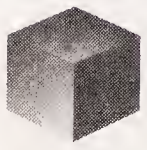
The expression *age* `> 17 && age<35` tests for ages *between* the two values, that is, ages in the range 18–34. The expression `age>=35 && age<50` uses the `>=` operator to include 35 in its range, which is 35–49. If we had used `age>35 && age<50` , the value 35 would have been missed by all the tests. When you use range tests, you should check that the ranges don't have holes between them. Also, be sure to set up each range correctly; see the discussion on Range Tests.

Look at the array declaration:

```
const char * qualify[4] =      // an array of pointers
{                              // to strings
    "10,000-meter race.\n",
    "mud tug-of-war.\n",
    "masters canoe jousting.\n",
    "pie-throwing festival.\n"
};
```

It states that *qualify* is an array of four elements, with each element being type `char*`, that is, pointer-to-`char`. So each element can hold the address of a character. Each quoted string, in turn, represents the address of its first character. Thus the program initializes four pointers to four addresses. Also, the `const` qualifier says these pointers point to characters that are constant. That means `more_and.cpp` has an array of string constants that are not to be modified.

The `if else` statement serves to select an array index, which, in turn, identifies a particular string.



Range Tests

Note that each part of a range test should use the AND operator to join two complete relational expressions:

```
if (age > 17 && age < 35)   // OK
```

Don't borrow from mathematics and use the following notation:

```
if (17 < age < 35)          // Don't do this
```

If you make this error, the compiler won't catch it, for it is still valid C++ syntax. The < operator associates from left to right, so the previous expression means the following:

```
if ( (17 < age) < 35)
```

But `17<age` is either true, or *1*, or else false, or *0*. In either case, the expression `17<age` is less than *35*, so the entire test is always true!

THE LOGICAL NOT OPERATOR: !

The ! operator negates, or reverses the truth value of, the expression following it. That is, if *expression* is true, then *!expression* is false, and vice versa. More precisely, if *expression* is nonzero, then !*expression* is zero. And if *expression* is zero, then *!expression* is *1*. Incidentally, many people call the exclamation point *bang*, making *!x* bang-exe and *!!x* bang-bang-exe.

Usually you can express a relationship more clearly without using this operator:

```
if (!(x > 5))                    // if (x <= 5) is clearer
```

But the ! operator can be useful with functions that return true-false values. For instance, `strcmp(s1,s2)` returns a nonzero (true) value if the two strings *s1* and *s2* are different from each other and a zero value (false) if they are the same. This implies that `!strcmp(s1,s2)` is true if the two strings are equal.

Listing 6-7 uses this technique (applying the ! operator to a true-false function) to screen numeric input for suitability to be assigned to type `int`. The function `is_int()`, which we'll discuss further in a moment, returns true if its argument is within the range of values assignable to type `int`. The program then uses the test `while(!is_int(num))` to reject values that don't fit in the range.

LISTING 6-7 not.cpp

```
// not.cpp -- using the not operator
#include <iostream.h>
#include <limits.h>
int is_int(double);
int main(void)
{
    double num;

    cout << "Yo, dude! Enter an integer value: ";
    cin >> num;
    while (!is_int(num))   // continue while num is not int-able
    {
```

```
            cout << "Out of range -- please try again: ";
            cin >> num;
    }
    int val = num;
    cout << "You've entered the integer " << val << "\n";
    return 0;
}

int is_int(double x)
{
    if (x <= INT_MAX && x >= INT_MIN)  // use limits.h values
            return 1;
    else
            return 0;
}
```

Here is a sample run on a system with a two-byte *int*:

```
Yo, dude! Enter an integer value: 50000
Out of range -- please try again: -42000
Out of range -- please try again: 32767
You've entered the integer 32767
```

Program Notes

If you enter a too-large value to a program reading a type `int`, most implementations simply truncate the value to fit without informing you that data was lost. This program avoids that by first reading the potential `int` as a `double`. The `double` type has more than enough precision to hold a typical `int` value, and its range is much greater.

The `is_int()` function uses the two symbolic constants (*INT_MAX* and *INT_MIN*) defined in the `limits.h` file (discussed in Chapter 3) to determine whether its argument is within the proper limits. If so, the program returns a value of *1*; otherwise, it returns *0*.

The `main()` program uses a `while` loop to reject invalid input until the user gets it right. You could make the program friendlier by displaying the `int` limits when the input is out of range. Once the input has been validated, the program assigns it to an `int` variable.

Logical Operator Facts

As we mentioned, the C++ logical OR and logical AND operators have a lower precedence than relational operators. That means an expression such as

```
x > 5 && x < 10
```

is read this way:

```
(x > 5) && (x < 10)
```

The ! operator, on the other hand, has a higher precedence than any of the relational or arithmetic operators. Therefore, to negate an expression, you should enclose the expression in parentheses:

```
!(x > 5)     // is it false that x is greater than 5
!x > 5       // is !x greater than 5
```

The second expression, incidentally, is always false, for *!x* can only have the values *1* (true) or *0* (false).

The logical AND operator has a higher precedence than the logical OR operator. Thus the expression

```
age > 30 && age < 45 || weight > 300
```

means the following:

```
(age > 30 && age < 45) || weight > 300
```

That is, one condition is that *age* be in the range 31 to 44, and the second condition is that *weight* be greater than 300. The entire expression is true if one or the other or both of these conditions are true.

You can, of course, use parentheses to tell the program the interpretation you want. For instance, suppose you want to use `&&` to combine the condition that *age* be greater than 50 or *weight* be greater than 300 with the condition that donation be greater than 1000. You have to enclose the OR part within parentheses:

```
(age > 50 || weight > 300) && donation > 1000
```

Otherwise, the compiler will combine the *weight* condition with the *donation* condition instead of with the *age* condition.

C++ guarantees that when a program evaluates a logical expression, it evaluates it from left to right and stops evaluation as soon as it knows what the answer is. Suppose, for instance, you have this condition:

```
x != 0  && 1.0 / x > 100.0
```

If the first condition is false, then the whole expression must be false. That's because for this expression to be true, each individual condition must be true. Knowing the first condition is false, the program doesn't bother evaluating the second condition. That's fortunate in this example, for evaluating the second condition would have resulted in dividing by 0.

THE CTYPE.H LIBRARY OF CHARACTER FUNCTIONS

C++ has inherited from C a handy package of character-related functions, prototyped in the `ctype.h` header file, which simplify such tasks as determining whether a character is an uppercase letter or a digit or punctuation, and the like. For example, the `isalpha(ch)` function returns a true (nonzero) value if *ch* is a letter and a false value (zero) otherwise. Similarly, the `ispunct(ch)` returns a true value only if *ch* is a punctuation character, such as a comma or period. Using these functions is more convenient than using the AND and OR operators. For instance, here's how you might use AND and OR to test if a character *ch* is an alphabetic character:

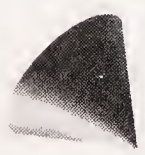
```
if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z')
```

Compare that to using `isalpha()`:

```
if (isalpha(ch))
```

Not only is `isalpha()` easier to use, it is more general. The AND, OR form assumes that characters codes for A through Z are in sequence, with no other characters having codes in that range. This assumption is true for the ASCII code, but it need not be true in general.

Listing 6-8 demonstrates some functions from this family. In particular, it uses `isalpha()`, which tests for alphabetic characters, `isdigits()`, which tests for digit characters, such as '3', `isspace()`, which tests for white space characters, such as newlines, spaces, and tabs, and `ispunct()`, which tests for punctuation characters. The program also reviews the `if else if else` structure and using a `while` loop with `cin.get()`.



LISTING 6-8 ctypes.cpp

```
// ctypes.cpp--use ctype.h library
#include <iostream.h>
#include <ctype.h>      / prototypes for character functions
int main(void)
{
    cout << "Enter text for analysis, and simulate EOF"
            " to terminate input.\n";
            char ch;
    int whitespace = 0;
    int digits = 0;
    int chars = 0;
    int punct = 0;
    int others = 0;
```

```
    while(cin.get(ch))
    {
            if(isalpha(ch))      // is it an alphabetic character?
                    chars++;
            else if(isspace(ch)) // is it a whitespace character?
                    whitespace++;
            else if(isdigit(ch)) // is it a digit?
                    digits++;
            else if(ispunct(ch)) // is it punctuation?
                    punct++;
            else
                    others++;
    }
    cout << chars << " letters, "
            << whitespace << " whitespace, "
            << digits << " digits, "
            << punct << " punctuations, "
            << others << " others.\n";
    return 0;
}
```

Here is a sample run; note that the white space count includes newlines:

```
Enter text for analysis, and simulate EOF to terminate input.
Freddy "Fuzzball" Fribble had 143 career
home runs and 1237 career strikeouts.
^Z
57 letters, 12 whitespace, 7 digits, 3 punctuations, 0 others.
```

Table 6-3 summarizes the functions available in the `ctype.h` package. The list corresponds to the ANSI C definition, but some systems may lack some of these functions or have additional ones.

THE ?: OPERATOR

C++ has an operator that can often be used instead of the `if` statement. This operator is called the conditional operator, written `?:`, and, for you trivia buffs, it is the only C++ operator that requires three operands. The general form looks like this:

```
expression1 ? expression2 : expression3
```

If *expression1* is true, then the value of the whole conditional expression is the value of *expression2*. Otherwise, the value of the whole expression is the value of *expression3*.

Here are two examples showing how the operator works:

```
5 > 3 ? 10 : 12  // 5 > 3 is true, so expression value is 10
3 == 9? 25 : 18  // 3 == 9 is false, so expression value is 18
```

We can paraphrase the first example this way: If 5 is greater than 3, the expression

Table 6-3 The ctype.h character functions

Function Name	Return Value
`isalnum()`	true if argument is alphanumeric, i.e., a letter or a digit
`isalpha()`	true if argument is alphabetic
`iscntrl()`	true if argument is a control character
`isdigit()`	true if argument is a decimal digit (0–9)
`isgraph()`	true if argument is any printing character, other than a space
`islower()`	true if argument is a lowercase letter
`isprint()`	true if argument is any printing character, including a space
`ispunct()`	true if argument is a punctuation character
`isspace()`	true if argument is a standard white-space character, i.e., a space, formfeed, newline, carriage return, horizontal tab, or vertical tab
`isupper()`	true if argument is an uppercase letter
`isxdigit()`	true if argument is a hexadecimal digit character, i.e., 0–9, a–f, or A–F
`tolower()`	if the argument is an uppercase character, `tolower()` returns the lowercase version of that character; otherwise, it returns the argument unaltered
`toupper()`	if the argument is a lowercase character, `toupper()` returns the uppercase version of that character; otherwise, it returns the argument unaltered

evaluates to 10; otherwise, it evaluates to 12. In real programming situations, of course, the expressions would involve variables.

Listing 6-9 uses the conditional operator to determine the larger of two values.

LISTING 6-9 condit.cpp

```
// condit.cpp -- using the conditional operator
#include <iostream.h>
int main(void)
{
    int a, b;
    cout << "Enter two numbers: ";
    cin >> a >> b;
    cout << "The larger of " << a << " and " << b;
    int c = a > b ? a : b;    // c = a if a > b, else c = b
    cout << " is " << c << "\n";
    return 0;
}
```

Here is a sample run:

```
Enter two numbers: 25 027          ← Note that 027 is octal
The larger of 25 and 23 is 25
```

The sample input should remind you that `cin`, like C++ in general, interprets integers beginning with the digit *0* as being written in octal. Thus *027* is interpreted to be the base ten value *23*.

The key part of the program is this statement:

```
int c = a > b ? a : b;
```

It produces the same result as the following statements:

```
int c;
if (a > b)
    c = a;
else
    c = b;
```

Compared to the `if else` sequence, the conditional operator is more concise but, at first, less obvious. One difference between the two approaches is that the conditional operator produces an expression and hence a single value that can be assigned or be incorporated into a larger expression, as we did in assigning the value of the conditional expression to the variable *c*. The conditional operator's concise form, unusual syntax, and overall weird appearance makes it a great favorite among programmers who appreciate those qualities. One favorite trick for concealing the purpose of code is to

nest conditional expressions within one another, as the following mild example shows:

```
char x[2] [20] = {"Jason ","at your service\n"};
char * y = "Quillstone ";

for (int i = 0; i < 3; i++)
    cout << ((i < 2)? !i ? x [i] : y : x[1]);
```

This is merely an obscure (but, by no means, maximally obscure) way to print the three strings.

```
Jason Quillstone at your service
```

The Switch Statement

Suppose you create a screen menu that asks the user to select one of five choices, for instance, Cheap, Moderate, Expensive, Extravagant, and Excessive. You can extend an `if else if else` sequence to handle five alternatives, but the C++ `switch` statement handles selecting a choice from an extended list more easily. Here's the general form for a `switch` statement:

```
switch (integer-expression)
{
    case label1 : statement(s)
    case label2 : statement(s)
     ...
    default     : statements(s)
}
```

A C++ `switch` statement acts as a routing device, telling the computer which line of code to execute next. On reaching a `switch`, the program jumps to the line labeled with the value corresponding to the value of *integer-expression*. For instance, if *integer-expression* has the value *4*, the program will go to the line having a `case 4:` label. The value *integer-expression*, as the name suggests, must be an expression that reduces to an integer value. Also, each label must be an integer constant expression. Most often, labels are simple `int` or `char` constants such as *1* or *'q'*. If *integer-expression* doesn't match any of the labels, the program jumps to the line labeled `default`. The `default` label is optional. If you omit it and there is no match, the program jumps to the next statement following the `switch`. See Figure 6-3.

The `switch` statement is different from similar statements in languages such as Pascal in a very important way. Each C++ `case` label functions only as a line label, not as a boundary between choices. That is, once a program jumps to a particular line in a `switch`, it then sequentially executes *all* the statements following that line in the

`switch` *unless* you explicitly direct it otherwise. Execution does NOT automatically stop at the next `case`. To make execution stop at the end of a particular group of statements, you must use the `break` statement. This causes execution to jump to the statement following the `switch`.

Listing 6-10 shows how to use `switch` and `break` together to implement a simple menu for executives. The program uses a `showmenu()` function to display a set of choices. A `switch` statement then selects an action based on the user's response.

LISTING 6-10 switch.cpp

```
// switch.cpp -- use the switch statement
#include <iostream.h>
void showmenu(void);   // function prototypes
void report(void);
void comfort(void);
int main(void)
{
    showmenu();
    int choice;
    cin >> choice;
    while (choice != 5)
    {
        switch(choice)
        {
            case 1 : cout << "\a\n";
```

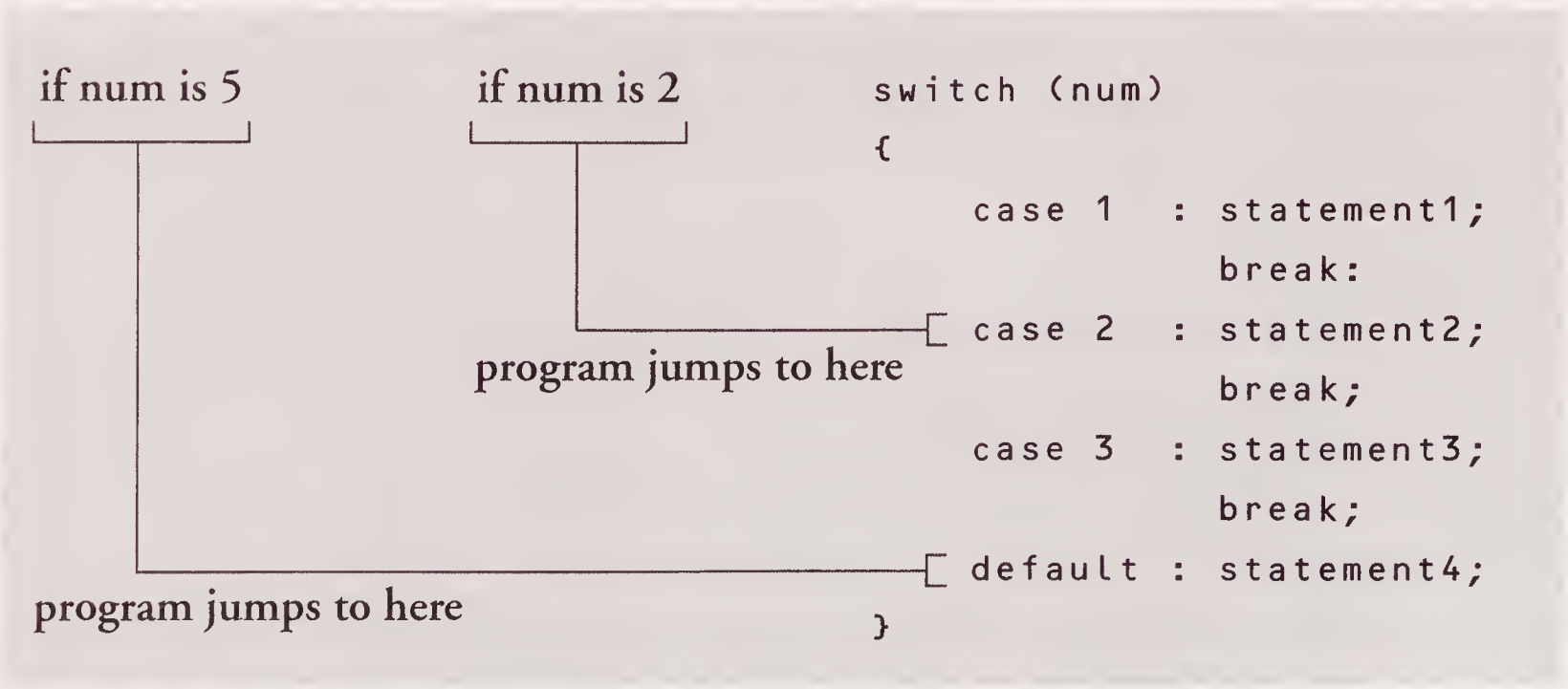


Figure 6-3 *The* `switch` *statement*

```
                        break;
                case 2 : report();
                        break;
                case 3 : cout << "The boss was in all day.\n";
                        break;
                case 4 : comfort();
                        break;
                default: cout << "That's not a choice.\n";
            }
            showmenu();
            cin >> choice;
    }
    cout << "Bye!\n";
    return 0;
}

void showmenu(void)
{
    cout << "Please enter 1, 2, 3, 4, or 5:\n"
                "1) alarm          2) report\n"
                "3) alibi          4) comfort\n"
                "5) quit\n";
}
void report(void)
{
    cout << "It's been an excellent week for business.\n"
            "Sales are up 120%. Expenses are down 35%.\n";
}
void comfort(void)
{
    cout << "Your employees think you are the finest CEO\n"
            "in the industry. The board of directors think\n"
            "you are the finest CEO in the industry.\n";
}
```

Here is a sample run of our executive menu program:

```
Please enter 1, 2, 3, 4, or 5:
1) alarm          2) report
3) alibi          4) comfort
5) quit
4
Your employees think you are the finest CEO
in the industry. The board of directors think
you are the finest CEO in the industry.
```

```
Please enter 1, 2, 3, 4, or 5:
1) alarm          2) report
3) alibi          4) comfort
5) quit
2
It's been an excellent week for business.
Sales are up 120%. Expenses are down 35%.
Please enter 1, 2, 3, 4, or 5:
1) alarm          2) report
3) alibi          4) comfort
5) quit
6
That's not a choice.
Please enter 1, 2, 3, 4, or 5:
1) alarm          2) report
3) alibi          4) comfort
5) quit
5
Bye!
```

The `while` loop terminates when the user enters a *5*. Entering *1* through *4* activates the corresponding choice from the `switch` list, and entering *6* triggers the `default` statements.

As noted before, this program needs the `break` statements to confine execution to a particular portion of a `switch`. To see that this is so, you can remove the `breaks` from Listing 6-9 and see how it works afterwards. You'll find, for example, that entering *2* causes the program to execute all the statements associated with case labels *2*, *3*, *4*, and the `default`. C++ works this way because that sort of behavior can be useful. For one thing, it makes it simple to use multiple labels. For instance, suppose you rewrote Listing 6-9 using characters instead of integers as menu choices and `switch` labels. Then you could use both an uppercase and a lowercase label for the same statements:

```
char choice;
cin >> choice;
while (choice != 'Q' && choice != 'q')
{
        switch(choice)
        {
                case 'a':
                case 'A': cout << "\a\n";
                          break;
                case 'r':
                case 'R': report();
                          break;
```

```
                case 'l':
                case 'L': cout << "The boss was in all day.\n";
                          break;
                case 'c'
                case 'C': comfort();
                          break;
                default : cout << "That's not a choice.\n";
        }
        showmenu();
        cin >> choice;
}
```

Because there is no *break* immediately following `case 'a'`, program execution passes on to the next line, which is the statement following `case 'A'`.

SWITCH AND IF ELSE

Both the `switch` statement and the `if else` statement let a program select from a list of alternatives. The `if else` is the more versatile of the two. For instance, it can handle ranges, as in the following:

```
if (age > 17 && age < 35)
        index = 0;
else if (age >= 35 && age < 50)
        index = 1;
else if (age >= 50 && age < 65)
        index = 2;
else
        index = 3;
```

The `switch`, however, isn't designed to handle ranges. Each `switch case` label must be a single value. Also, that value must be an integer (which includes *char*), so a `switch` won't handle floating-point tests. And the `case` label value must be a constant. So if your alternatives involve ranges or floating-point tests or comparing two variables, use `if else`.

If, however, all the alternatives can be identified with integer constants, you can use a `switch` or an `if else` statement. Because that's precisely the situation that the `switch` statement is designed to process, the `switch` statement usually is the more efficient choice in terms of code size and execution speed. The usual rule of thumb is to use the `switch` statement if you have three or more alternatives.

The break and continue Statements

The `break` and `continue` statements enable a program to skip over parts of the code. The `break` statement can be used in a `switch` statement and in any of the loops. It causes program execution to pass to the next statement following the `switch` or the loop. The `continue` statement is used in loops and causes a program to skip the rest of the body of the loop. See Figure 6-4.

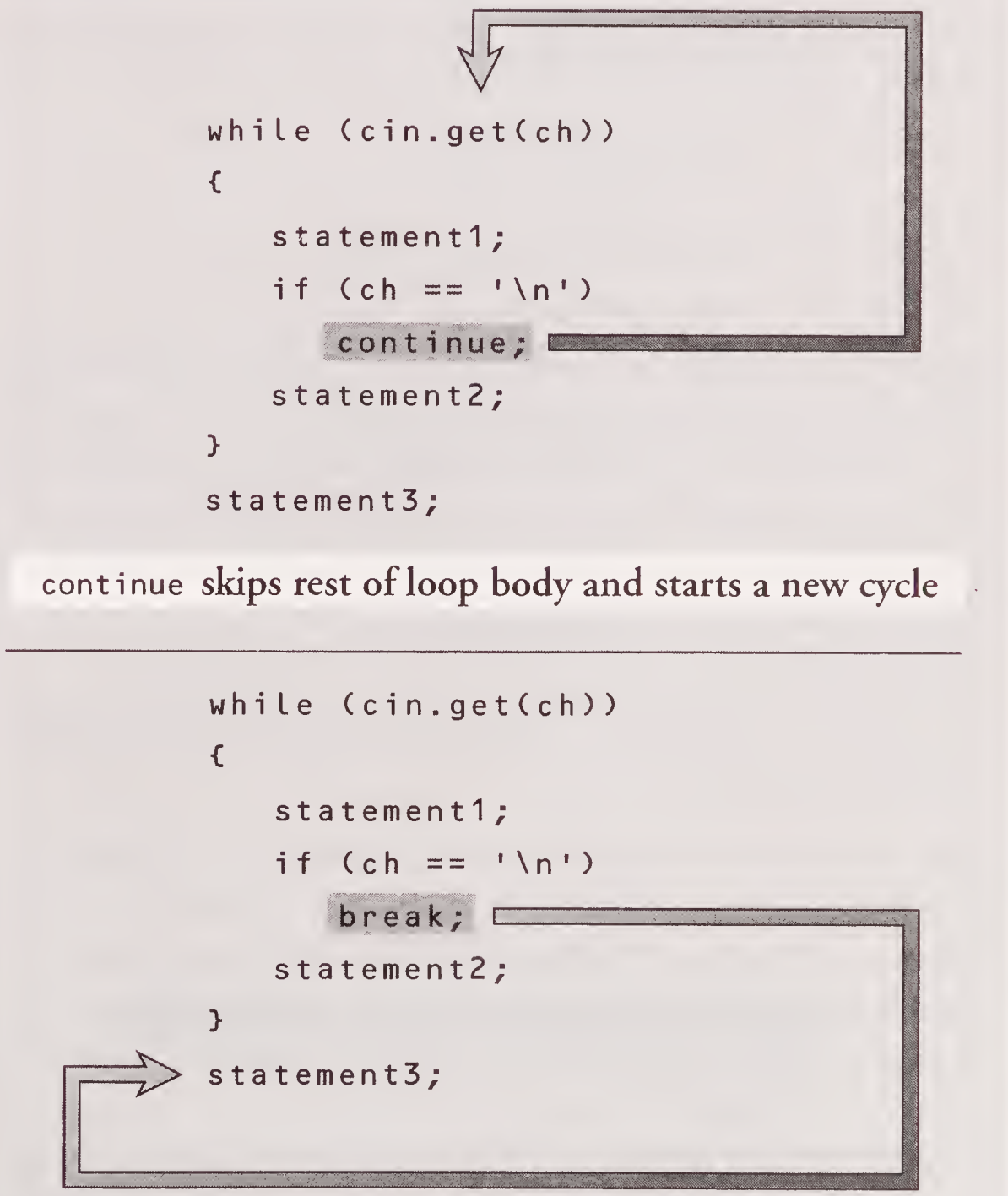


Figure 6-4 *The* `break` *and* `continue` *statements*

Listing 6-11 shows how the two statements work. The program lets you enter a line of text. The loop echoes each character and uses `break` to terminate the loop if the character is a period. This shows how you can use `break` to terminate a loop from within when some condition becomes true. Next the program counts characters, but not spaces. The loop uses `continue` to skip over the counting part of the loop when the character is a space.

LISTING 6-11 jump.cpp

```
// jump.cpp -- using continue and break
#include <iostream.h>
const int Size = 80;
int main(void)
{
    char line[Size];
    int spaces = 0;

    cout << "Enter a line of text:\n";
    cin.get(line, Size);
    for (int i = 0; line[i] != '\0'; i++)
    {
        cout << line[i];      // display character
        if (line[i] == '.')   // quit if it's a period
            break;
        if (line[i] != ' ')   // skip rest of loop
            continue;
        spaces++;
    }
    cout << "\n" << spaces << " spaces\n";
    return 0;
}
```

```
Enter a line of text:
Let's do lunch today. You can pay!
Let's do lunch today.
3 spaces
```

Program Notes

Note that while the `continue` statement causes the program to skip the rest of the loop body, it doesn't skip the loop update expression. In a `for` loop, the `continue` statement makes the program skip directly to the update expression, then to the test expression. For a `while` loop, however, `continue` makes the program go directly to the test expression. So any update expression in the loop body following the `continue` would be skipped. In some cases, that could be a problem.

This program didn't have to use `continue`. Instead, it could have used this code:

```
if (line[i] != ' ')
    spaces++;
```

However, the `continue` statement can make the program more readable when several statements follow the `continue`. That way, you don't need to make all those statements part of an `if` statement.

Summary

Programs and programming become more interesting when you introduce statements that guide the program through alternative actions. (Whether this also makes the programmer more interesting is a point we've not fully researched.) C++ provides the `if` statement, the `if else` statement, and the `switch` statements as means for managing choices. The C++ `if` statement lets a program execute a statement or statement block conditionally. That is, the program executes the statement or block if a particular condition is met. The C++ `if else` statement lets a program select from two choices which statement or statement block to execute. You can append additional `else if`s to the statement to present a series of choices. The C++ `switch` statement directs the program to a particular case in a list of choices.

C++ also provides operators to help in decision-making. In Chapter 5 we discussed the relational expressions, which compare two values. The `if` and `if else` statements typically use relational expressions as test conditions. By using C++'s logical operators (&&, ||, and !), you can combine or modify relational expressions, constructing more elaborate tests. The conditional operator (?:) provides a compact way to choose from two values.

The `ctype.h` library of character functions provides a convenient and powerful set of tools for analyzing character input.

With C++'s loops and decision-making statements, you have the tools for writing interesting, intelligent, and powerful programs. But we've only begun to investigate the real powers of C++. Next, we'll look at functions.

Review Questions

1. Consider the following two code fragments for counting spaces and newlines:

```
// Version 1
while (cin.get(ch))
{
    if (ch == ' ')
            spaces++;
    if (ch == '\n')
            newlines++;
}
// Version 2
while (cin.get(ch))
{
    if (ch == ' ')
            spaces++;
    else if (ch == '\n')
            newlines++;
}
```

What advantages, if any, does the second form have over the first?

2. In Listing 6-2, what is the effect of replacing `++ch` with `ch+1`?

3. Consider carefully the following program:

```
#include <iostream.h>
int main(void)
{
    char ch;
    int ct1, ct2;

    ct1 = ct2 = 0;
    while ((ch = cin.get()) != '$')
    {
            cout << ch;
            ct1++;
            if (ch = '$')
                    ct2++;
            cout << ch;
    }
    cout <<"ct1 = " << ct1 << ", ct2 = " << ct2 << "\n";
    return 0;
}
```

Suppose we provide the following input, where RETURN represents pressing the Return key:

```
Hi!RETURN
Send $10 or $20 now!RETURN
```

What is the output? (Assume input is buffered.)

4. Construct logical expressions to represent the following conditions:
   a. *weight* is greater than or equal to 115 but less than 125.
   b. *ch* is *'q'* or *'Q'*.
   c. *x* is even, but is not 26.
   d. *donation* is in the range 1000–2000 or *guest* is 1.
   e. *ch* is a lowercase letter or an uppercase letter (assume the lowercase letters are coded sequentially and that the uppercase letters are coded sequentially but that there is a gap in the code between uppercase and lowercase)

5. In English the statement "I will not not speak" means the same as "I will speak." In C++, is *!!x* the same as *x*?

6. Construct a conditional expression that is equal to the absolute value of a variable. That is, if a variable *x* is positive, the value of the expression is just *x*, but if *x* is negative, the value of the expression is *-x*, which would be positive.

7. Rewrite the following fragment using `switch`:

```
if (ch == 'A')
   a_grade++;
else if (ch == 'B')
   b_grade++;
else if (ch == 'C')
   c_grade++;
else if (ch == 'D')
   d_grade++;
else
   f_grade++;
```

8. In Listing 6-9, what advantage would there be in using character labels, such as *'a'* and *'c'*, instead of numbers for the menu choices and `switch` cases? (Hint: Think about what happens if the user types *q* in either case and what happens if the user types *5* in either case.)

9. Consider the following code fragment:

```
int line = 0;
char ch;
```

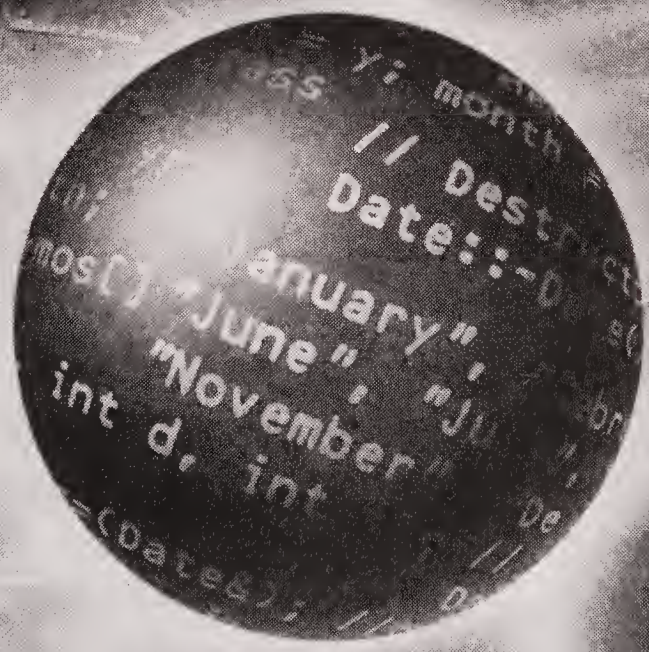
```
while (cin.get(ch))
{
   if (ch == 'Q')
    break;
   if (ch != '\n')
    continue;
   line++;
}
```

Rewrite this code without using `break` or `continue`.

10. Write a program that reads keyboard input to end-of-file and that echoes the input except for digits, converting each uppercase character to lowercase, and vice versa.

Functions—C++'s Programming Modules

7



Fun is where you find it. Look closely, and you'll find it in functions. C++ comes with a large library of useful functions, but real programming pleasure comes with writing your own. In this and the next chapter you'll examine how to define functions, how to convey information to them, and how to retrieve information from them. After reviewing how functions work, this chapter concentrates on how to use functions in conjunction with arrays, strings, and structures. If you've paid your C dues, you'll find much of this chapter familiar. But don't be lulled into a false sense of expertise. C++ has made several additions to what functions can do, and the next chapter deals primarily with them. Meanwhile, let's attend to the fundamentals.

Function Review

First, let's review what you've already seen about functions. To use a C++ function, you must do the following:

- Provide a function definition.
- Provide a function prototype.
- Call the function.

If you're using a library function, the function has already been defined and compiled for you. Also, you can use a standard library header file to provide the prototype. So all that's left to do is to call the function properly. The examples to date have done that several times. For example, the standard C++ library includes the `strlen()` function for finding the length of the string. The associated standard header file `string.h` contains the function prototype for `strlen()` and several other string-related functions. This advance work allows you to use the `strlen()` function in programs without further worries.

But when you create your own functions, you have to handle all three aspects—defining, prototyping, and calling—yourself. Listing 7-1 shows these steps in a short example.

LISTING 7-1 calling.cpp

```
// calling.cpp -- defining, prototyping, and calling a function
#include <iostream.h>

void simple(void);              // function prototype

int main(void)
{
    cout << "main() will call the simple() function:\n";
    simple();                   // function call
}

// function definition
void simple(void)
{
    cout << "I'm just a simple country function.\n";
}
```

Here's the output:

```
main() will call the simple() function:
I'm just a simple country function.
```

Let's take a more detailed look at these steps now.

DEFINING A FUNCTION

Functions can be grouped into two categories: functions that don't have return values and functions that do have return values. Functions without return values are termed type `void` functions and have the following general form:

```
void functionname(argumentlist)
{
    statement(s)
    return;         // optional
}
```

Here *argumentlist* specifies the types and number of arguments that are passed to the function. We'll investigate this list more fully later. The optional `return` statement marks the end of the function. Otherwise the function terminates at the closing brace. Type `void` functions correspond to Pascal procedures, FORTRAN subroutines, and modern BASIC subprogram procedures. Typically, you use a `void` function to perform some sort of action. For instance, a function to print *Cheers!* a given number (*n*) of times could look like this:

```
void cheers(int n)          // no return value
{
    for (int i = 0; i < n; i++)
            cout << "Cheers! ";
    cout << "\n";
}
```

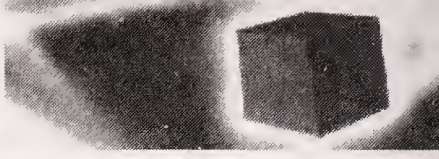
The `int n` argument list means that when you call this function, you should use a type `int` argument.

A function with a return value produces a value that it returns to the function that called it. In other words, if the function returns the square root of 9 (`sqrt(9)`), then this becomes replaced by the value *3*. Such a function is declared as having the same type as the value it returns. Here is the general form:

```
typename functionname(argumentlist)
{
    statement(s)
    return value;   // value is of type typename
}
```

Functions with return values require that you use a `return` statement so that the value is returned to the calling statement. The value itself can be a constant, a variable, or a more general expression. The only requirement is that the expression reduce to a value having the `typename` type. The function then returns the final value to the function that called it. C++ does place a restriction on what types you can use for a return value: the return value cannot be an array. That leaves everything else possible—integers, floating-point numbers, pointers, even structures and objects! (Interestingly, even though a C++ function can't return an array directly, it can return an array that's part of a structure or object.)

As a programmer, you don't need to know how a function returns a value, but knowing the method may make the concept clearer to you. (Also, it gives you something to talk about with your friends and family.) Typically, a function returns a value by copying the return value to a specified CPU register or memory location. Then the calling program examines that location. So both the returning function and the calling function have to agree on the type of data at that location. The function prototype tells



```
...
double cube(double x);   // function prototype
...
int main(void)
{
   ...
   double q = cube(1.2); // function call
   ...
}

double cube(double x)    // function definition
{
   return x * x * x;
}
```

cube() calculates return value and places it here; function header tells cube() to use a type double value

1.728

return value location

main() looks here for the return value and assigns it to *q*; cube() prototype tells main() to expect type double

Figure 7-1 *Typical return value mechanism*

the calling program what to expect, and the function definition tells the called program what to return. See Figure 7-1. Providing the same information in the prototype as in the definition may seem like extra work, but it does make good sense. Certainly, if you wanted a courier to pick up something for you from your desk at the office, you enhance the odds of the task being done right if you provide a description of what you want both to the courier and to someone at the office.

A function terminates after executing a `return` statement. If a function has more than one `return` statement, for example, as alternatives to different `if else` selections, the function terminates after executing the first `return` statement it reaches:

```
int bigger(int a, int b)
{
   if (a > b )
         return a;     // if a > b, function terminates here
```

```
    else
            return b;      // otherwise, function terminates here
}
```

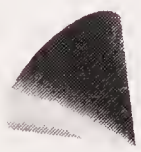
Functions with return values are much like functions in Pascal, FORTRAN, and BASIC. They return a value to the calling program, which can then assign that value to a variable, or display the value, or otherwise use it. Here's a simple example that returns the cube of a type `double` value:

```
double cube(double x)   // x times x times x
{
    return x * x * x;    // a type double value
}
```

For instance, the function call `cube(1.2)` would return the value *1.728*. Note that this `return` statement uses an expression. The function computes the value of the expression (*1.728* in this case) and returns the value.

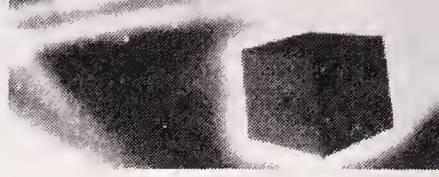
FUNCTION PROTOTYPING AND FUNCTION CALLS

By now you are familiar with making function calls, but you may be less comfortable with function prototyping because that's often been hidden in the include files. So let's use the two functions we just defined in a program (Listing 7-2), paying notice to the function prototypes.



LISTING 7-2 protos.cpp

```
// protos.cpp -- use prototypes and function calls
#include <iostream.h>
void cheers(int);         // prototype: no return value
double cube(double x);    // prototype: returns a double
int main(void)
{
    cheers(5);            // function call
    cout << "Give me a number: ";
    double side;
    cin >> side;
    double volume = cube(side); // function call
    cout << "A " << side <<"-foot cube has a volume of ";
    cout << volume << " cubic feet.\n";
    cheers(cube(2));      // prototype protection at work
    return 0;
}
```



```
void cheers(int n)
{
    for (int i = 0; i < n; i++)
            cout << "Cheers! ";
    cout << "\n";
}

double cube(double x)
{
    return x * x * x;
}
```

Here's a sample run:

```
Cheers! Cheers! Cheers! Cheers! Cheers!
Give me a number: 5
A 5-foot cube has a volume of 125 cubic feet.
Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers!
Cheers!
```

Note that `main()` calls the type `void` function `cheers()` by using the function name and arguments followed by a semicolon: `cheers(s);`. That's an example of a function call statement. But because `cube()` has a return value, `main()` can use it as part of an assignment statement:

```
double volume = cube(side);
```

But we said you should concentrate on the prototypes. What should you know about prototypes? First, you should understand why C++ requires prototypes. Then, because C++ requires prototypes, you should know the proper syntax. Finally, you should appreciate what the prototype does for you. Let's look at these points in turn, using Listing 7-2 as a basis for discussion.

Why Prototypes?

The prototype describes the function interface to the compiler. That is, it tells the compiler what type of return value, if any, the function has, and it tells the compiler the number and the type of function arguments. Consider, for example, how a prototype affects this function call from Listing 7-2:

```
double volume = cube(side);
```

First, the prototype tells the compiler that `cube` should have one type `double` argument. If the program fails to do so, prototyping allows the compiler to catch the error. Second, when the `cube()` function finishes its calculation, it places its return value at some

specified location—perhaps in a CPU register, perhaps in memory. Then the calling function, `main()` in this case, retrieves the value from that location. Because the prototype states that `cube()` is type `double`, the compiler knows how many bytes to retrieve and how to interpret them. Without that information, the compiler could only guess.

Still, you may wonder, why does the compiler need a prototype? Can't it just look further in the file and see how the functions are defined? One problem with that approach is that it is less efficient. The compiler would have to put compiling `main()` on hold while it searched the rest of the file. An even more serious problem is the fact that the function may not even be in the file. C++ allows you to spread a program over several files, which can be compiled independently, then combined later. If that's the case, then the compiler may not have access to the function code when it's compiling `main()`. The same is true if the function is part of a library. The only way to avoid using a function prototype is to place the function definition *above* its first use. That is not always possible. And the C++ programming style is to put `main()` first because it generally provides the structure for the whole program.

Prototype Syntax

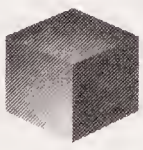
A function prototype is a statement, so it must have a terminating semicolon. The simplest way to get a prototype is to copy the function heading from the function definition and add a semicolon. That's what we did for `cube()`:

```
double cube(double x); // add ; to heading to get prototype
```

However, the function prototype does not require that you provide names for the variables; a list of types is enough. So we prototyped `cheers()` just using the argument type:

```
void cheers(int); // okay to drop variable names in prototype
```

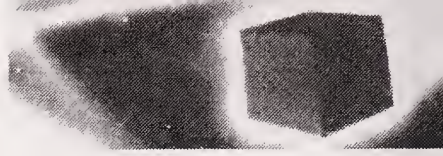
In general, you can either include or exclude variable names in the arguments lists for prototypes. The variable names in the prototype just act as place holders, so if you do use names, they don't have to match the names in the function definition.



C++ versus ANSI C Prototyping

ANSI C borrowed prototyping from C++, but the two languages do have some differences. The most important is that ANSI C, to preserve compatibility with classic C, made prototyping optional, while C++ makes prototyping mandatory. For instance, consider the following function declaration:

```
void say_hi();
```



In C++, leaving the parentheses empty is the same as using the keyword `void` within the parentheses. It means the function has no arguments. In ANSI C, leaving the parentheses empty means that you decline to state what the arguments are. That is, it means you're foregoing prototyping the argument list.

What Prototypes Do for You

You've seen that prototypes help the compiler. But what do they do for you? They greatly reduce the chances for program errors. In particular, prototypes ensure the following:

- The program correctly handles the function return value.
- The program checks that you use the correct number of function arguments.
- The program checks that you use the correct type of arguments. If not, it converts the arguments to the correct type, if possible.

We've already discussed getting the return value handled correctly. Let's look now at what happens when you use the wrong number of arguments. For instance, suppose you made the following call:

```
double z = cube();
```

Without function prototyping, the compiler would let this go by. When the function was called, it would look where `cube()` should have placed a number and use whatever value happened to be there. This, for instance, is how C worked before ANSI C borrowed prototyping from C++. Since prototyping is optional for ANSI C, this is still how some C programs work. But in C++ prototyping is not optional, so you are guaranteed protection from that sort of error.

Next, suppose you use the correct number but the wrong type of argument. In C, this could create weird errors. For instance, if a function expected a type `int` value (assume that's two bytes) and you passed a `double` (assume that's eight bytes), the function would look at just the first two bytes of the eight and try to interpret them as an `int` value. C++, however, automatically converts the value you pass to the type specified in the prototype. For instance, Listing 7-2 manages to get two type mismatches in one statement:

```
cheers(cube(2));
```

First, the program passes the `int` value of *2* to `cube()`, which expects type `double`. The compiler, noting that the `cube()` prototype specifies a type `double` argument, converts *2* to *2.0*, a type `double` value. Then `cube()` returns a type `double` value (*8.0*) to be used as an argument to `cheers()`. Again, the compiler checks the prototypes and notes

that `cheers()` requires an `int`. So it converts the return value to the integer *8*. In general, prototyping produces automatic type casts to the expected types.

Automatic type conversion doesn't head off all possible errors. For instance, if you pass a value of 8.33E27 to a function that expects an `int`, such a large value cannot be converted correctly to a mere `int`. Some compilers will warn you of possible data loss when there is an automatic conversion from a larger type to a smaller.

Also, prototyping results in type conversion only when it makes sense. It won't, for example, convert an integer to a structure or to a pointer.

Function Arguments and Passing by Value

It's time to take a closer look at function arguments. C++ normally passes arguments *by value*. That means the numerical value of the argument is passed to the function, where it is assigned to a new variable. For instance, Listing 7-2 has this function:

```
double volume = cube(side);
```

Here *side* is a variable that, in the sample run, had the value *5*. The function heading for `cube()`, recall, was this:

```
double cube(double x)
```

When this function is called, it creates a new type `double` variable called *x* and assigns the value *5* to it. This insulates data in `main()` from actions that take place in `cube()`, for `cube()` works with a copy of *side* instead of with the original data. You'll see an example of this protection soon. A variable that's used to receive passed values is called a *formal argument* or *parameter*. The value passed to the function is called the *actual argument*. Thus argument passing assigns the actual argument to the formal argument. See Figure 7-2.

Variables, including formal parameters, declared within a function are private to the function. When a function is called, the computer allocates the memory needed for these variables. When the function terminates, the computer frees the memory that was used for those variables. (Some C++ literature refers to this allocation and freeing of memory as creating and destroying variables. That does make it sound much more exciting.) Such variables are called *local* variables because they are localized to the function. As we mentioned, this helps preserve data integrity. It also means that if you declare a variable called *x* in `main()` and another variable called *x* in some other function, that these are two distinct, unrelated variables much as the Albany in California is distinct from the Albany in New York. See Figure 7-3.

MULTIPLE ARGUMENTS

Functions can have more than one argument. In the function call, just separate the arguments with commas:

```
n_chars('R', 25);
```

This passes two arguments to the function `n_char()`, soon to be defined.

Similarly, when you define the function, you use a comma-separated list of declarations in the function heading:

```
void n_chars(char c, int n)    // two arguments
```

This function heading states that the function `n_char()` takes one type `char` argument and one type `int` argument. The variables *c* and *n* are assigned the values passed to the function. If a function takes two arguments of the same type, you have to give the type of each argument separately. You can't combine declarations the way you can when declaring regular variables:

```
void fifi(float a, float b)    // declare each variable separately
void fifi(float a, b)          // NOT acceptable
```

As with other functions, just add a semicolon to get a prototype:

```
void n_chars(char c, int n);  // prototype, style 1
```

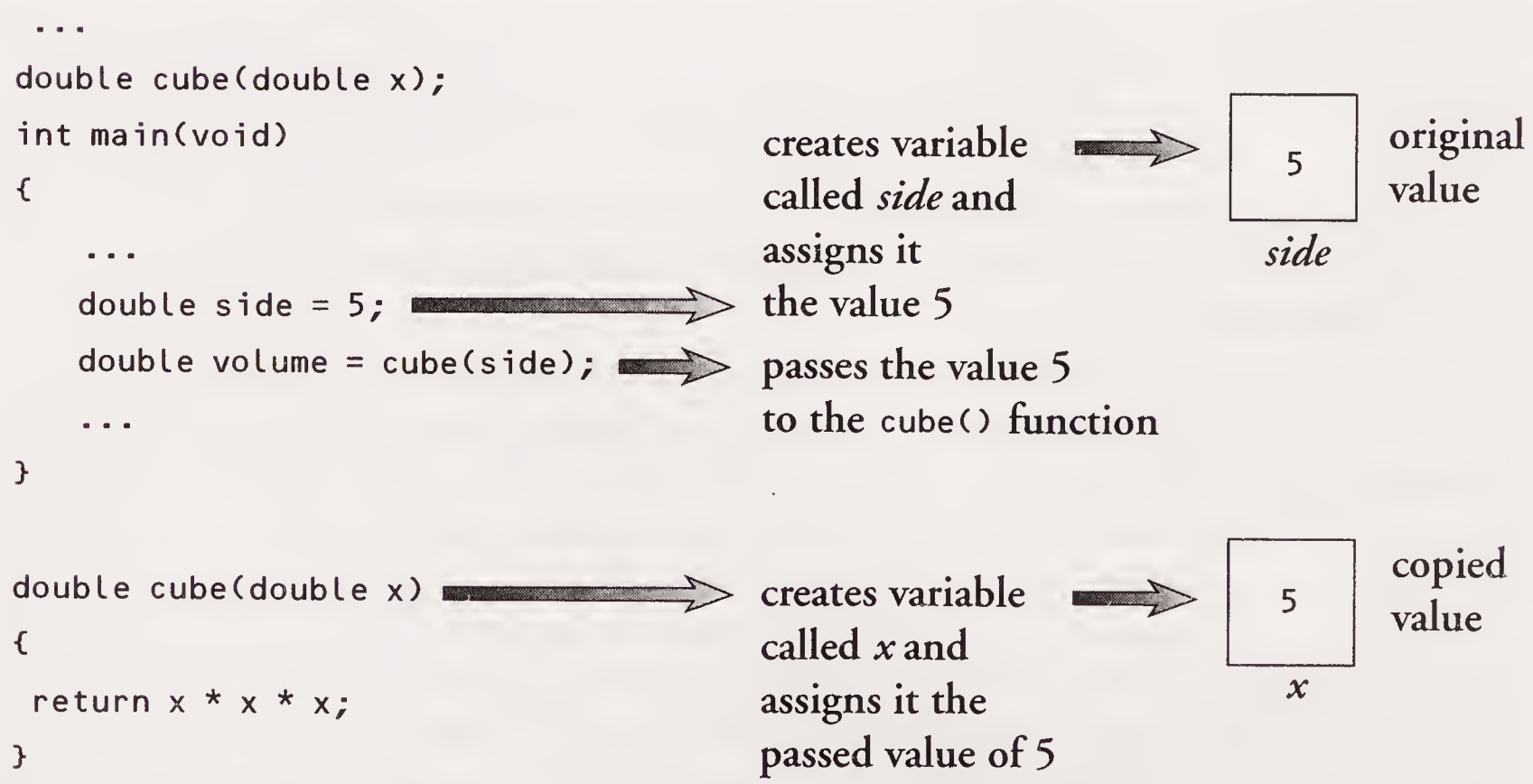


Figure 7-2 *Passing by value*

You don't have to use the same variable names in the prototype. Also, you can omit the variable names in the prototype:

```
void n_char s(char, int);    // prototype, style 2
```

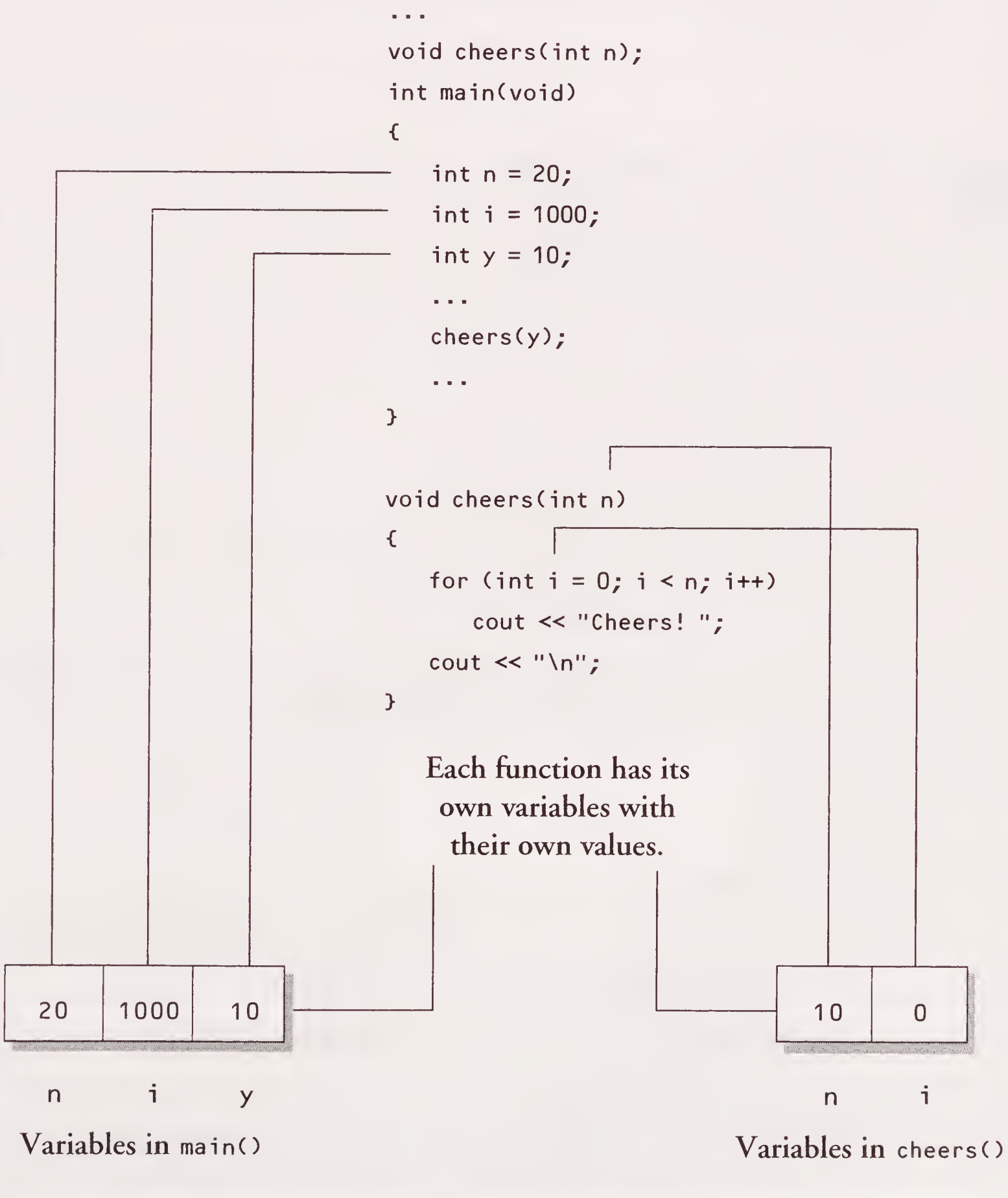
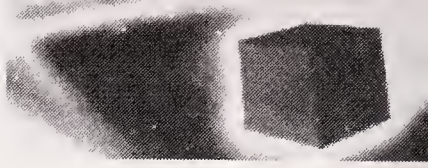


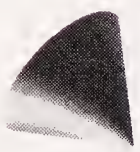
Figure 7-3 *Local variables*



However, providing variable names can make the prototype more understandable, particularly if two parameters are the same type. Then the names can remind you which argument is which:

```
double melon_density(double weight, double volume);
```

Listing 7-3 shows an example of a function with two arguments. It also illustrates how changing the value of a formal argument in a function has no effect on the data in the calling program.



LISTING 7-3 twoarg.cpp

```
// twoarg.cpp -- a function with 2 arguments
#include <iostream.h>
void n_chars(char, int);
int main(void)
{
    int times;
    char ch;
    cout << "Enter a character: ";
    cin >> ch;
    while (ch != 'q')              // q to quit
    {
            cout << "Enter an integer: ";
            cin >> times;
            n_chars(ch, times);  // function with two arguments
            cout << "\nEnter another character or press the"
                            " q-key to quit: ";
            cin >> ch;
    }
    cout << "The value of times is " << times << ".\n";
    cout << "Bye\n";
    return 0;
}

void n_chars(char c, int n)     // displays c n times
{
    while (n-- > 0)             // continue until n reaches 0
            cout << c;
}
```

Here is a sample run:

```
Enter a character: W
Enter an integer: 50
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
Enter another character or press the q-key to quit: a
Enter an integer: 20
aaaaaaaaaaaaaaaaaaaa
Enter another character or press the q-key to quit: q
The value of times is 20.
Bye
```

Program Notes

The `main()` function uses a `while` loop to keep loops fresh in your mind. Note that it uses `cin >> ch` to read a character instead of `cin.get(ch)` or `cin = cin.get()`. There's a good reason for doing so. The `cin.get()` pair of functions, you recall, read *all* input characters, including spaces and newlines, while `cin` skips spaces and newlines. When you respond to the program prompt, you have to press Return at the end of each line, thus generating a newline character. The `cin >> ch` approach conveniently skips over these newlines, but the `cin.get()` siblings would read the newline following each number entered as the next character to display. You can program around this nuisance, but it's simpler to use `cin` as we did.

The `n_char()` function takes two arguments: a character *c* and an integer *n*. It then uses a loop to display the character the number of times specified by the integer:

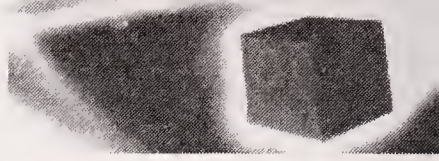
```
while (n-- > 0)          // continue until n reaches 0
    cout << c;
```

Notice that the program keeps count by decrementing the *n* variable, where *n* is the formal parameter from the argument list. This variable was assigned the value of the *times* variable in `main()`. The `while` loop then decreases *n* to zero, but, as the sample run demonstrates, changing the value of *n* has no effect on *times*.

Another Two-Argument Function

Let's create a more ambitious function, one that performs a nontrivial calculation. Also, the function will illustrate using local variables other than the function's formal arguments.

Many states in the United States now sponsor a lottery with some form of Lotto game. Lotto lets you pick a certain number of choices from a card. For instance, you might get to pick six numbers from a card having forty-nine numbers. Then the Lotto managers pick six numbers at random. If your choice exactly matches theirs, you win a



few million dollars or so. Our function will calculate the probability that you have a winning pick. (Yes, a function that successfully predicts the winning picks themselves would be more useful, but C++, although powerful, has yet to implement psychic facilities.)

First, we need a formula. Suppose you have to pick six values out of forty-nine. Then mathematics says you have one chance in R of winning, where R is given by the following formula:

$$R = \frac{49 \times 48 \times 47 \times 46 \times 45 \times 44}{6 \times 5 \times 4 \times 3 \times 2 \times 1}$$

For six choices the denominator is the product of the first six integers, or 6 factorial. The numerator also is the product of six consecutive numbers, this time starting with 49 and going down. More generally, if you pick *picks* values out of *numbers* numbers, the denominator is *picks* factorial and the numerator is the product of *picks* integers starting with the value *numbers* and working down. You can use a `for` loop to make that calculation:

```
long double result = 1.0;
for (n = numbers, p = picks; p > 0; n--, p--)
    result = result * n / p ;
```

Rather than multiplying all the numerator terms first, the loop begins by multiplying 1.0 by the first numerator term, then dividing by the first denominator term. Then, in the next cycle, the loop multiplies and divides by the second numerator and denominator terms. This keeps the running product smaller than if you did all the multiplication first. For instance, compare

```
(10 * 9) / (2 * 1)
```

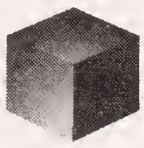
with the following:

```
(10 / 2) * (9 / 1)
```

The first evaluates to 90 / 2, then to 45, while the second evaluates to 5 * 9, then to 45. Both give the same answer, but the first method produces a larger intermediate value (90) than does the second. The more factors you have, the bigger the difference gets. For large numbers, this strategy can keep the calculation from overflowing the maximum possible floating-point value.

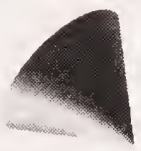
Listing 7-4 incorporates this formula into an `odds()` function. Because the number of picks and the total number of choices should be positive values, the program uses the `unsigned int` type (`unsigned`, for short) for those quantities. Multiplying several integers together can produce pretty large results, so `lotto.cpp` uses the `long`

`double` type for the function's return value. Also, terms such as 49 / 6 produce a truncation error for integer types.



COMPATIBILITY NOTE:

Some C++ implementations don't support type `long double`. If your implementation falls into that category, try ordinary `double` instead.



LISTING 7-4 lotto.cpp

```
// lotto.cpp -- odds against winning
#include <iostream.h>
// Note: some implementations require double instead of long double
long double odds(unsigned numbers, unsigned picks);
int main(void)
{
   double total, choices;
   cout << "Enter total number of game card choices and\n"
        "number of picks allowed:\n";
   while ((cin >> total >> choices) && choices <= total)
   {
      cout << "You have one chance in ";
      cout << odds(total, choices);     // compute the odds
      cout << " of winning.\n";
      cout << "Next two numbers (q to quit): ";
   }
   cout << "bye\n";
   return 0;
}

// the following function calculates the odds of picking picks
// numbers correctly from numbers choices
long double odds(unsigned numbers, unsigned picks)
{
   long double result = 1.0;  // here come some local variables
   long double n;
   unsigned p;

   for (n = numbers, p = picks; p > 0; n--, p--)
      result = result * n / p ;
   return result;
}
```

Here's a sample run. Notice that increasing the number of choices on the game card greatly increases the odds against winning.

```
Enter total number of game card choices and
number of picks allowed:
49 6
You have one chance in 13983816 of winning.
Next two numbers (q to quit): 53 6
You have one chance in 2.295748e+07 of winning.
Next two numbers (q to quit): 38 6
You have one chance in 2760681 of winning.
Next two numbers (q to quit): q
bye
```

Program Notes

The `odds()` function illustrates two kinds of local variables you can have in a function. First, there are the formal parameters (*numbers* and *picks*), which are declared in the function heading before the opening brace. Then come the other local variables (*result*, *n*, and *p*). They are declared in between the braces bounding the function definition. The main difference between the formal parameters and the other local variables is that the formal parameters get their values from the function that calls *odds()*, while the other variables get values from within the function.

Functions and Arrays

So far the example functions have been simple, using only the basic types for arguments and return values. But functions can be the key to handling more involved types, such as arrays and structures. Let's take a look now at how arrays and functions get along with each other.

Suppose you've been using an array to keep track of how many cookies each person has eaten at the family picnic. (Each array index corresponds to a person, and the value of the element corresponds to the number of cookies that person ate.) Now you want the total. That's easy to do; just use a loop to add all the array elements. But adding array elements is such a common task that it makes sense to design a function to do the job. Then you won't have to write a new loop every time you have to sum an array.

Let's consider what the function interface involves. Since the function calculates a sum, it should return the answer. If we keep our cookies intact, we can use a function

with a type `int` return value. So that the function knows what array to sum, we'll want to pass the array name as an argument. And to make the function general so that it is not restricted to an array of a particular size, we'll pass the size of the array. The only new ingredient here is that we have to declare that one of the formal arguments is an array name. Let's see what that and the rest of the function heading look like:

```
int sum_arr(int arr[], int n) // arr = array name, n = size
```

This looks plausible. The brackets seem to indicate that *arr* is an array, and the fact the brackets are empty seems to indicate we can use the function with an array of any size. But things are not always what they seem: *arr* is not really an array; it's a pointer! The good news is that we can write the rest of the function just as if *arr* were an array. First, we'll see that this approach works, then we'll look into why it works.

Listing 7-5 illustrates using a pointer as if it were an array name. The program initializes the array to some values and uses the `sum_arr()` function to calculate the sum. You'll note that the `sum_arr()` function uses *arr* as if it were an array name.

LISTING 7-5 arrfun1.cpp

```
// arrfun1.cpp -- functions with an array argument
#include <iostream.h>
const int Size = 8;
int sum_arr(int arr[], int n);          // prototype
int main(void)
{
    int cookies[Size] = {1,2,4,8,16,32,64,128};
// some systems require preceding int with static to
// enable array initialization

    int sum = sum_arr(cookies, Size);
    cout << "Total cookies eaten: " << sum <<  "\n";
    return 0;
}

// return the sum of an integer array
int sum_arr(int arr[], int n)
{
    int total = 0;

    for (int i = 0; i < n; i++)
            total = total + arr[i];
    return total;
}
```

Here is the program output:

```
Total cookies eaten: 255
```

As you can see, the program works. Now let's look at why it works.

ARRAYS AND POINTERS (AGAIN)

The key is that C++, like C, in most contexts treats the name of an array as if it were a pointer. Recall from Chapter 4 that C++ interprets an array name as the address of its first element:

```
cookies == &cookies[0]  // array name is address of first element
```

(There are two exceptions to this rule. First, the array declaration uses the array name to label the storage. Second, applying `sizeof` to an array name yields the size of the whole array, in bytes.)

Listing 7-5 makes the following function call:

```
int sum = sum_arr(cookies, Size);
```

Here *cookies* is the name of an array, hence, by C++ rules, the address of its first element. So the function passes an address. Because the array has type `int` elements, *cookies* must be type pointer-to-`int`, or `int*`. That suggests that the correct function heading should be this:

```
int sum_arr(int *arr, int n) // arr = array name, n = size
```

Here we've replaced `int arr[]` with `int*arr`. It turns out that both headings are correct, for in C++ the notations `int *arr` and `int arr[]` have the identical meaning when used in a function heading or function prototype. Both mean that *arr* is a pointer-to-`int`. However, the array notation version (`int arr[]`) symbolically reminds us that *arr* not only points to an `int`, it points to the first `int` in an array of `int`s. So we'll use the array notation when the pointer is to the first element of an array, and we'll use the pointer notation when the pointer is to an isolated value. Incidentally, the notations `int *arr` and `int arr[]` are *not* synonymous in any other context. For instance, you can't use the notation `int tip[]` to declare a pointer in the *body* of a function.

Given that the variable *arr* is really a pointer, the rest of the function makes sense. As you may recall from the discussion of dynamic arrays in Chapter 4, you can use the bracket array notation equally well with array names or with pointers to access elements of an array. So whether *arr* is a pointer or an array name, the expression *arr[3]* means the fourth element of the array. And it probably will do you no harm at this point to remind you of the following two identities:

```
arr[i] == *(ar + i)      // values in two notations
&arr[i] == ar + i        // addresses in two notations
```

Remember, adding 1 to a pointer, including an array name, actually adds a value equal to the size, in bytes, of the type the pointer points to. So pointer addition and array subscripting are two equivalent ways of counting elements from the beginning of an array.

IMPLICATIONS OF USING ARRAYS AS ARGUMENTS

Let's look at the implications of Listing 7-5. The function call `sum_arr(cookies, Size)` passes the address of the first element of the *cookies* array and the number of elements of the array to the `sum_arr()` function. The `sum_arr()` function assigns the *cookies* address to the pointer variable *arr* and assigns *Size* to the *int* variable *n*. This means Listing 7-5 doesn't really pass the array contents to the function. Instead, it tells the function *where* the array is (the address), *what* kind of elements it has (the type), and *how many* elements it has (the *n* variable). See Figure 7-4. Armed with this information, the function then uses the *original* array. Pass an ordinary variable, and the function works with a copy. But pass an array, and the function works with the original. Actually, this difference doesn't violate C++'s pass-by-value approach. The `sum_arr()` function still passes a value that's assigned to a new variable. But that value is a single address, not the contents of an array.

Is the correspondence between array names and pointers a good thing? Indeed, it is. The design decision to use array addresses as arguments saves the time and memory needed to copy an entire array. The overhead for using copies can be prohibitive if

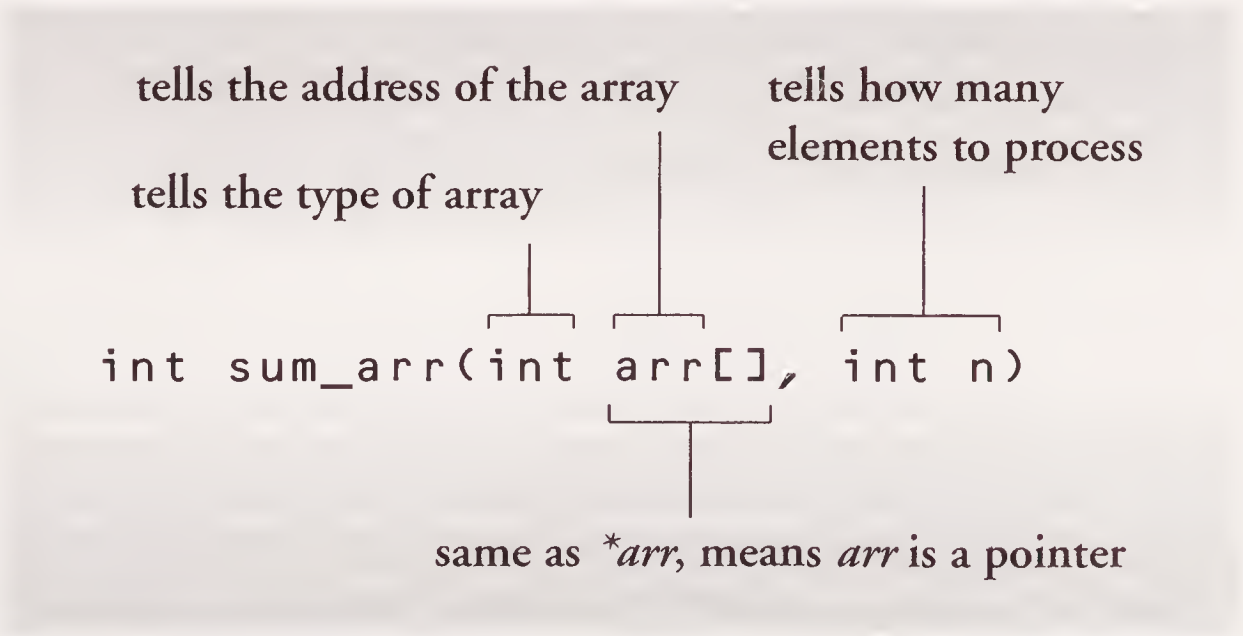


Figure 7-4 *Telling a function about an array*

you're working with large arrays. Not only would a program need more computer memory, but it would have to spend time copying large blocks of data. On the other hand, working with the original data raises the possibility of inadvertent data corruption. That's a real problem in classic C, but ANSI C and C++'s `const` modifier provides a remedy. We'll show an example soon. But first, let's alter Listing 7-5 to illustrate some points about how array functions operate. Listing 7-6 demonstrates that *cookies* and *arr* have the same value. It also shows how the pointer concept makes the `sum_arr` function more versatile than it may have appeared at first.

LISTING 7-6 arrfun2.cpp

```
// arrfun2.cpp -- functions with an array argument
#include <iostream.h>
const int Size = 8;
int sum_arr(int arr[], int n);
int main(void)
{
    int cookies[Size] = {1,2,4,8,16,32,64,128};
//  some systems require preceding int with static to
//  enable array initialization

    cout << cookies << " = array address, ";
//  some systems require a type cast: unsigned (cookies)

    cout << sizeof cookies << " = sizeof cookies\n";
    int sum = sum_arr(cookies, Size);
    cout << "Total cookies eaten: " << sum <<  "\n";
    sum = sum_arr(cookies, 3);          // a lie
    cout << "First three eaters ate " << sum << " cookies.\n";
    sum = sum_arr(cookies + 4, 4);      // another lie
    cout << "Last four eaters ate " << sum << " cookies.\n";
    return 0;
}

// return the sum of an integer array
int sum_arr(int arr[], int n)
{
    int total = 0;
    cout << arr << " = arr, ";
// some systems require a type cast: unsigned (arr)

    cout << sizeof arr << " = sizeof arr\n";
    for (int i = 0; i < n; i++)
            total = total + arr[i];
    return total;
}
```

Here's the output:

```
0x8f8cffe6 = array address, 16 = sizeof cookies
0x8f8cffe6 = arr, 2 = sizeof arr
Total cookies eaten: 255
0x8f8cffe6 = arr, 2 = sizeof arr
First three eaters ate 7 cookies.
0x8f8cffee = arr, 2 = sizeof arr
Last four eaters ate 240 cookies.
```

Program Notes

This example illustrates some very interesting points about array functions. First, note that *cookies* and *arr* both evaluate to the same address, exactly as claimed. But `sizeof cookies` is *16*, while `sizeof arr` is only *2*. That's because `sizeof cookies` is the size of the whole array, while `sizeof arr` is the size of the pointer variable. (Our system, remember, uses two-byte addresses.) By the way, that's why we have to pass the size of the array explicitly instead of using `sizeof arr` in `sum_arr()`.

Because the only way `sum_arr()` knows the number of elements in the array is through what we tell it with the second argument, we can lie to the function. For instance, the second time we use the function, we make this call:

```
sum = sum_arr(cookies, 3);
```

By telling the function that *cookies* has but three elements, we get the function to calculate the sum of the first three elements.

Why stop there? We also can lie about where the array starts:

```
sum = sum_arr(cookies + 4, 4);
```

Because *cookies* acts as the address of the first element, *cookies + 4* acts as the address of the fifth element. So this statement sums the fifth, sixth, seventh, and eighth elements of the array. Note in the output how the third call to the function assigns a different address to *arr* than the first two calls did. And yes, you could use *&cookies[4]* instead of *cookies + 4* as the argument; both mean the same thing.

More Array Functions

When you choose to use an array to represent data, you've made a design decision. But design decisions should go beyond how data is stored; they should also involve how the data is used. Often, you'll find it profitable to write specific functions to handle specific data operations. (The profits here are increased program reliability, ease of modifica-

tion, and ease of debugging.) Also, when you start integrating storage properties with operations in your thinking about a program, you'll be taking an important step towards the OOP mind-set; that, too, may prove profitable in the future.

Let's examine a simple case. Suppose you want to use an array to keep track of the dollar values of your real estate. (If necessary, suppose you have real estate.) You'll have to decide on what type to use. Certainly, `double` is less restrictive in its range than `int` or `long`, and it provides enough significant digits to represent the values precisely. Next, you have to decide on the number of array elements. (With dynamic arrays created with `new`, you can put off that decision, but we'll keep things simple.) Let's say that you have no more than five properties, so you can use an array of five `doubles`.

Now consider the possible operations you might want to execute with the real estate array. Two very basic ones are reading values into the array and displaying the array contents. Let's add one more operation to the list: reassessing the value of the properties. For simplicity, assume that all your properties increase or decrease in value at the same rate. (Remember, this is a book on C++, not on real estate management.) Next, think about what each function should do, then write the code accordingly. We'll go through these steps next.

Filling the Array

Because a function with an array name argument accesses the original array, not a copy, you can use a function call to assign values to array elements. So one argument to the function will be the name of the array to be filled. In general, a program might manage more than one person's investments, hence more than one array, so you won't want to build the array size into the function. Instead, pass the array size as a second argument, as in the previous example. Also, it's possible that you may wish to quit reading data before filling the array, so you'll want to build that feature into the function. Because you may enter fewer than the maximum number of elements, it makes sense to have the function return the actual number of values entered. These considerations suggest the following function prototype:

```
int fill_array(double ar[], int limit);
```

The function takes an array name argument and an argument specifying the maximum number of items to be read, and the function returns the actual number of items read. For instance, if you use this function with an array of five elements, you would pass *5* as the second argument. And if you entered only three values, the function would return *3*.

You can use a loop to read successive values into the array, but how can you terminate the loop early? One way is use a special value to indicate the end of input. Since no property should have a negative value, you can use a negative number to indicate the end of input. Given this, you can code the function as follows:

```
int fill_array(double ar[], int limit)
{
    double temp;
    for (int i = 0; i < limit; i++)
    {
            cout << "Enter value #" << i + 1 << ": ";
            cin >> temp;
            if (temp < 0)     // signal to terminate
                    break;
            ar[i] = temp;
    }
    return i;
}
```

Note that the code includes a prompt to the user in the program. If the user enters a nonnegative value, the value is assigned to the array. Otherwise the loop terminates. If the user enters only valid values, the loop terminates after reading *limit* values. The last thing the loop does is increment *i*, so after the loop terminates, *i* is 1 greater than the last array index, hence it's equal to the number of filled elements. The function then returns that value.

Showing the Array and Protecting It with const

Building a function to display the array contents is simple. You pass the name of the array and the number of filled elements to the function, which then uses a loop to display each element. But there is another consideration—guaranteeing that the display function doesn't alter the original array. Unless the purpose of a function is to alter data, you should safeguard it from doing so. That protection comes automatically with ordinary arguments, since C++ passes them by value and the function works with a copy. But functions that use an array work with the original. After all, that's why the `fill_array()` function is able to do its job. To protect an array from being altered, you can use the keyword `const` (discussed in Chapter 3) when declaring the formal argument:

```
void show_array(const double ar[], int n);
```

The declaration states that the pointer *ar* points to constant data. This means that you can't use *ar* to change the data. That is, you can use a value such as *ar[0]*, but you can't change that value. Note that this doesn't mean that the original array need be constant; it just means that you can't use *ar* in the `show_array()` function to change the data. Suppose you accidentally violate this restriction by doing something like the following in the `show_array()` function:

```
ar[0] += 10;
```

Then the compiler will put a stop to your wrongful ways. Turbo C++, for an example, will give an error message like this (edited slightly):

```
Cannot modify a const object in function
    show_array(const double *,int)
```

The message reminds us that C++ interprets the declaration `const double ar[]` to mean `const double *ar`. Thus the declaration really says that *ar* points to a constant value. We'll discuss this in detail when we finish with the current example. Meanwhile, here is the code for the *show_array()* function:

```
void show_array(const double ar[], int n)
{
    for (int i = 0; i < n; i++)
    {
            cout << "Property #" << i + 1 << ": $";
            cout << ar[i] << "\n";
    }
}
```

Modifying the Array

The third operation for our array is multiplying each element by the same revaluation factor. You need to pass three arguments to the function: the factor, the array, and the number of elements. No return value is needed, so the function can look like this:

```
void reassess(double r, double ar[], int n)
{
    for (int i = 0; i < n; i++)
            ar[i] *= r;
}
```

Because this function is supposed to alter the array values, you don't use `const` to declare *ar*.

Putting the Pieces Together

Now that we've defined the data type in terms of how it's stored (an array) and how it's used (three functions), we can put together a program using the design. Because we've already built all the array-handling tools, we've greatly simplified programming `main()`. Most of the remaining programming work consists of having `main()` call the functions we've just developed. Listing 7-7 shows the result.

LISTING 7-7 arrfun3.cpp

```
// arrfun3.cpp -- array functions and const
#include <iostream.h>
const int Max = 5;

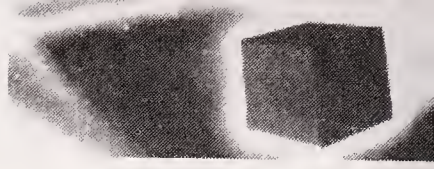
// function prototypes
int fill_array(double ar[], int limit);
void show_array(const double ar[], int n);  // don't change data
void reassess(double r, double ar[], int n);

int main(void)
{
    double properties[Max];

    int size = fill_array(properties, Max);
    show_array(properties, size);
    cout << "Enter reassessment rate: ";
    double rate;
    cin >> rate;
    reassess(rate, properties, size);
    show_array(properties, size);
    return 0;
}

int fill_array(double ar[], int limit)
{
    double temp;
    for (int i = 0; i < limit; i++)
    {
            cout << "Enter value #" << i + 1 << ": ";
            cin >> temp;
            if (temp < 0)
                    break;
            ar[i] = temp;
    }
    return i;
}

// the following function can use, but not alter,
// the array whose address is ar
void show_array(const double ar[], int n)
{
    for (int i = 0; i < n; i++)
    {
            cout << "Property #" << i + 1 << ": $";
            cout << ar[i] << "\n";
```



```
    }
}

// multiplies each element of ar[] by r
void reassess(double r, double ar[], int n)
{
    for (int i = 0; i < n; i++)
            ar[i] *= r;
}
```

Here are two sample runs. Recall that input should quit when the user enters five properties or enters a negative number, whichever comes first. The first example illustrates reaching the five-property limit, and the second example illustrates entering a negative value.

```
Enter value #1: 100000
Enter value #2: 80000
Enter value #3: 222000
Enter value #4: 240000
Enter value #5: 118000
Property #1: $100000
Property #2: $80000
Property #3: $222000
Property #4: $240000
Property #5: $118000
Enter reassessment rate: 1.10
Property #1: $110000
Property #2: $88000
Property #3: $244200
Property #4: $264000
Property #5: $129800

Enter value #1: 200000
Enter value #2: 84000
Enter value #3: 160000
Enter value #4: -2
Property #1: $200000
Property #2: $84000
Property #3: $160000
Enter reassessment rate: 1.20
Property #1: $240000
Property #2: $100800
Property #3: $192000
```

Program Notes

We've already discussed the important programming details, so let's reflect on the process. We began by thinking about the data type and designed appropriate functions to handle the data. Then we assembled these functions into a program. This sometimes is called *bottom-up* programming because the design process moves from the component parts to the whole. This approach is well suited to OOP, which concentrates on data representation and manipulation first. Traditional procedural programming, on the other hand, has leaned towards *top-down* programming, in which you develop a modular grand design first, then turn your attention towards the details.

POINTERS AND CONST

Using `const` with pointers has some subtle aspects (pointers always seem to have subtle aspects), so let's take a closer look. You can use the `const` keyword two different ways with pointers. The first way is to make a pointer point to a constant object, and that prevents you from using the pointer to change the pointed-to value. The second way is to make the pointer itself constant, and that prevents you from changing where the pointer points. Now for the details.

First, let's declare a pointer *pt* that points to a constant:

```
int age = 39;
const int * pt = &age;
```

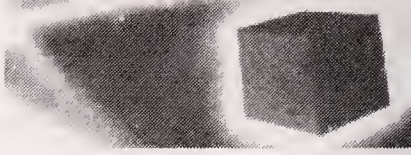
This declaration states that *pt* points to a `const int` (*39*, in this case). Therefore you can't use *pt* to change that value. In other words, the value **pt* is `const` and cannot be modified:

```
*pt += 1;          // INVALID because pt points to a const int
cin >> *pt;        // INVALID for the same reason
```

Now for a subtle point. Our declaration for *pt* doesn't necessarily mean that the value it points to is really a constant; it just means the value is a constant insofar as *pt* is concerned. For instance, *pt* points to *age*, and *age* is not *const*. So you can change the value of *age* directly by using the *age* variable, but you can't change the value indirectly via the *pt* pointer:

```
*pt = 20;          // INVALID because pt points to a const int
age = 20;          // VALID because age is not declared to be const
```

In the past, we've assigned the address of a regular variable to a regular pointer. Now we just assigned the address of a regular variable to a pointer-to-`const`. That leaves two other possibilities: assigning the address of a `const` variable to a pointer-to-`const`



and assigning the address of a `const` to a regular pointer. Are they both possible? The first is, and the second isn't:

```
const float g_earth = 9.80;
const float * pe = &g_earth;   // VALID

const float g_moon = 1.63;
float * pm = &g_moon;          // INVALID
```

For the first case, you can use neither *g_earth* nor *pe* to change the value *9.80*. C++ doesn't allow the second case for a simple reason—if you could assign the address of *g_moon* to *pm*, then you could cheat and use *pm* to alter the value of *g_moon*. That would make a mockery of *g_moon*'s `const` status, so C++ prohibits assigning the address of a `const` to a non-`const` pointer.

Now another subtle point: the declaration

```
int age = 39;
const int * pt = &age;
```

only prevents you from changing the value *pt* points to, which is *39*. It doesn't prevent you from changing the value of *pt* itself. That is, you can assign a new address to *pt*:

```
int sage = 80;
pt = &sage;         // okay to point to another location
```

But you still can't use *pt* to change the value it points to (now *80*).

The second way to use `const` makes it impossible to change the value of the pointer itself:

```
int sloth = 3;
const int * ps = &sloth;         // a pointer to const int
int * const finger = &sloth;     // a const pointer to int
```

Note that the last declaration has repositioned the keyword `const`. This form of declaration constrains *finger* to point only to *sloth*. However, it does allow you to use *finger* to alter the value of *sloth*. The middle declaration does not allow you to use *ps* to alter the value of *sloth*, but it does permit you to have *ps* point to another location. In short, *finger* and **ps* are both `const`, and **finger* and *ps* are not `const`. See Figure 7-5.

If you like, you can declare a `const` pointer to a `const` object:

```
double trouble = 2.0E30;
const double * const stick = &trouble;
```

Here *stick* can point only to *trouble*, and *stick* cannot be used to change the value of *trouble*. In short, both *stick* and **stick* are `const`.

Most often we'll be using the pointer-to-`const` form to protect data when we pass pointers as function arguments. For example, recall the `show_array()` prototype from Listing 7-5:

```
void show_array(const double ar[], int n);
```

Using `const` in that declaration means that `show_array()` cannot alter the values in any array passed to it.

Functions and Strings

A string, you'll recall, consists of a series of characters terminated by the null character. Much of what you've learned about designing array functions applies to string functions, too. But there are a few special twists to strings that we'll unravel now.

```
int gorp = 16;
int chips = 12;
const int * p_snack = &gorp;
```

NO

```
*p_snack = 20;
```

disallows changing value to which `p_snack` points

OK

```
p_snack = &chips;
```

`p_snack` can point to another variable

```
int gorp = 16;
int chips = 12;
int * const p_snack = &gorp;
```

OK

```
*p_snack = 20;
```

`p_snack` can be used to change value

NO

```
p_snack = &chips;
```

disallows changing variable to which `p_snack` points

Figure 7-5 *Pointers-to-*`const` *and* `const` *pointers*

Suppose you wish to pass a string as an argument to a function. You have three choices for representing a string:

- An array of `char`
- A quoted string constant
- A pointer-to-`char` set to the address of a string

All three choices, however, are type pointer-to-`char` (more concisely, type `char*`), so all three can be used as arguments to string-processing functions:

```
char ghost[15] = "galloping";
char * str = "galumphing";
int n1 = strlen(ghost);                  // ghost is &ghost[0]
int n2 = strlen(str);                    // pointer to char
int n3 = strlen("gambolling");           // address of string
```

Informally, we may say we're passing a string as an argument, but we're really passing the address of the first character in the string. This implies that a string function prototype should use type `char*` as the type for the formal parameter representing a string.

One important difference between a string and a regular array is that the string has a built-in terminating character. (Recall that a `char` array containing characters but no null character is just an array and not a string.) That means you don't have to pass the size of the string as an argument. Instead, the function can use a loop to examine each character in the string in turn until the loop reaches the terminating null character. Listing 7-8 illustrates that approach with a function that counts the number of times a given character appears in a string.

LISTING 7-8 strgfun.cpp

```
// strgfun.cpp -- functions with a string argument
#include <iostream.h>
int c_in_str(const char * str, char ch);
int main(void)
{
    char mmm[15] = "minimum";   // string in an array
// some systems require preceding char with static to
// enable array initialization

    char *wail = "ululate";     // wail points to string

    int ms = c_in_str(mmm, 'm');
    int us = c_in_str(wail, 'u');
    cout << ms << " m characters in " << mmm << "\n";
    cout << us << " u characters in " << wail << "\n";
    return 0;
}
```

```
// this function counts the number of ch characters
// in the string str
int c_in_str(const char * str, char ch)
{
    int count = 0;

    while (*str)          // quit when *str is '\0'
    {
            if (*str == ch)
                    count++;
            str++;        // move pointer to next char
    }
    return count;
}
```

Here's the output:

```
3 m characters in minimum
2 u characters in ululate
```

Program Notes

Because the `c_int_str()` function shouldn't alter the original string, it uses the `const` modifier when declaring the formal parameter *str*. Then, if you mistakenly let the function alter part of the string, the compiler will catch your error. Of course, you can use array notation instead for declaring *str* in the function heading:

```
int c_in_str(const char str[], char ch) // also okay
```

However, using pointer notation reminds us that the argument doesn't have to be the name of an array but can be some other form of pointer.

The function itself demonstrates a standard way to process the characters in a string:

```
while (*str)
{
    statement(s)
    str++;
}
```

Initially, *str* points to the first character in the string, so `*str` represents the first character itself. For instance, immediately after the first function call, **str* has the value *'m'*, the first character in *minimum*. As long as the character is not the null character (`'\0'`), **str* is nonzero, so the loop continues. At the end of each loop the expression

`str++` increments the pointer by one byte so it points to the next character in the string. Eventually, *str* points to the terminating null character, making `*str` equal to *0*, which is the numeric code for the null character. That condition terminates the loop.

FUNCTIONS THAT RETURN STRINGS

Now suppose you want to write a function that returns a string. Well, a function can't do that. But it can return the address of a string, and that's even better. Listing 7-9 for instance, defines a function called `buildstr()` that returns a pointer. This function takes two arguments: a character and a number. Using `new`, the function creates a string whose length equals the number, and then it initializes each element to the character. Then it returns a pointer to the new string.

LISTING 7-9 strgback.cpp

```
// strgback.cpp -- a function returning a pointer to char
#include <iostream.h>
char * buildstr(char c, int n);                    // prototype
int main(void)
{
    int times;
    char ch;

    cout << "Enter a character: ";
    cin >> ch;
    cout << "Enter an integer: ";
    cin >> times;
    char *ps = buildstr(ch, times);
    cout << ps << "\n";
    delete ps;                                     // free memory
    ps = buildstr('+', 20);                        // reuse pointer
    cout << ps << "-DONE-" << ps << "\n";
    return 0;
}

// builds string made of n c characters
char * buildstr(char c, int n)
{
    char * pstr = new char[n + 1];
```

```
    pstr[n] = '\0';              // terminate string
    while (n-- > 0)
            pstr[n] = c;         // fill rest of string
    return pstr;
}
```

Here's a sample run:

```
Enter a character: V
Enter an integer: 46
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
++++++++++++++++++++-DONE-++++++++++++++++++++
```

Program Notes

To create a string of *n* visible characters, you need storage for *n + 1* characters in order to have space for the null character. So the function asks for *n + 1* bytes to hold the string. Next, it sets the final byte to the null character. Then it fills in the rest of the array from back to front. The loop

```
while (n-- > 0)
        pstr[n] = c;
```

cycles *n* times as *n* decreases to zero, filling *n* elements. At the start of the final cycle, *n* has the value *1*. Because *n--* means use the value, then decrement it, the *while* loop test condition compares *1* to *0*, finds the test to be true, and continues. But after making the test, the function decrements *n* to *0*, so *pstr[0]* is the last element set to *c*. The reason for filling the string from back to front instead of front to back is to avoid using an additional variable. Using the other order would involve something like this:

```
int i = 0;
while (i < n)
    pstr[i++] = c;
```

Note that the variable *pstr* is local to the `buildstr` function, so when that function terminates, the memory used for *pstr* is freed. But because the function returns the value of *pstr*, the program is able to access the new string through the *ps* pointer in `main()`.

Just to keep in practice, we used `delete` to free memory used for the string after the string was no longer needed. We then reused *ps* to point to the new block of memory obtained for the next string.

Functions and Structures

Let's move from arrays to structures. It's easier to write functions for structures than for arrays. Although structure variables resemble arrays in that both can hold several data items, structure variables behave like basic, single-valued variables when it comes to functions. You can pass structures by value, just as you do with ordinary variables. In that case, the function works with a copy of the original structure. And a function can return a structure. There's no funny business like the name of an array being the address of its first element. The name of a structure is simply the name of the structure, and if you want its address, you have to use the `&` address operator.

The most direct way to program using structures is to treat them as you would treat the basic types; that is, pass them as arguments and use them, if necessary, as return values. However, there is one disadvantage to passing structures by value. If the structure is large, the space and effort involved in making a copy of a structure can increase memory requirements and slow the system down. For those reasons (and because, at first, C didn't allow passing structures by value), many C programmers prefer passing the address of a structure, then using a pointer to access the structure contents. C++ provides a third alternative, called passing by reference, that we'll discuss in Chapter 8, and we'll examine the other two choices now, beginning with passing and returning entire structures.

Passing and Returning Structures

Passing structures by value makes the most sense when the structure is relatively compact, so let's develop a couple of examples along those lines. The first example deals with travel time (not to be confused with time travel). Some maps will tell you that it is 3 hours, 50 minutes, from Thunder Falls to Bingo City and 1 hour, 25 minutes, from Bingo City to Grotesquo. We can use a structure to represent such times, using one member for the hour value and a second member for the minute value. Adding two times is a little tricky because you may have to transfer some of the minutes to the hours part. For instance, the two times above sum to 4 hours, 75 minutes, which should be converted to 5 hours, 15 minutes. Let's develop a structure to represent a time value, then develop a function that takes two such structures as arguments and returns a structure representing their sum.

Defining the structure is simple:

```
struct travel_time
{
    int hours;
    int mins;
};
```

Next, consider the prototype for a `sum()` function that returns the sum of two such structures. The return value should be type `travel_time`, and so should the two arguments. Thus the prototype should look like this:

```
travel_time sum(travel_time t1, travel_time t2);
```

To add two times, first add the minute members. Integer division by 60 yields the number of hours to carry over, and the modulus operator (`%`) yields the number of minutes left. Listing 7-10 incorporates this approach into the `sum()` function and adds a `show_time()` function to display the contents of a *travel_time* structure.

LISTING 7-10 travel.cpp

```
// travel.cpp -- using structures with functions
#include <iostream.h>
struct travel_time
{
    int hours;
    int mins;
};
const int Mins_per_hr = 60;

travel_time sum(travel_time t1, travel_time t2);
void show_time(travel_time t);

int main(void)
{

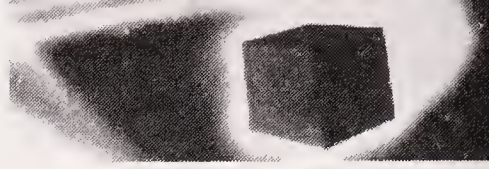
    travel_time day1 = {5, 45};          // 5 hrs, 45 min
    travel_time day2 = {4, 55};          // 4 hrs, 55 min
// Some implementations require using static travel_time
// to allow structure initialization

    travel_time trip = sum(day1, day2);
    cout << "Two-day total: ";
    show_time(trip);

    travel_time day3= {4, 32};
    cout << "Three-day total: ";
    show_time(sum(trip, day3));

    return 0;
}

travel_time sum(travel_time t1, travel_time t2)
{
```



```
    travel_time total;

    total.mins = (t1.mins + t2.mins) % Mins_per_hr;
    total.hours = t1.hours + t2.hours +
                                (t1.mins + t2.mins) / Mins_per_hr;
    return total;
}

void show_time(travel_time t)
{
    cout << t.hours << " hours, "
         << t.mins << " minutes\n";
}
```

Here *travel_time* acts just like a standard typename; you can use it to declare variables, function return types, and function argument types. Because variables like *total* and *t1* are *travel_time* structures, you can apply the dot membership operator to them. Note that because the `sum()` function returns a *travel_time* structure, it can be used as an argument for the `show_time()` function. Because C++ functions, by default, pass arguments by value, the `show_time(sum(trip, day3))` function call first evaluates the `sum(trip, day3)` function call in order to find its return value. The `show_time()` call then passes `sum()`'s return value, not the function itself, to `show_time()`. Here's the program output:

```
Two-day total: 10 hours, 40 minutes
Three-day total: 15 hours, 12 minutes
```

ANOTHER EXAMPLE

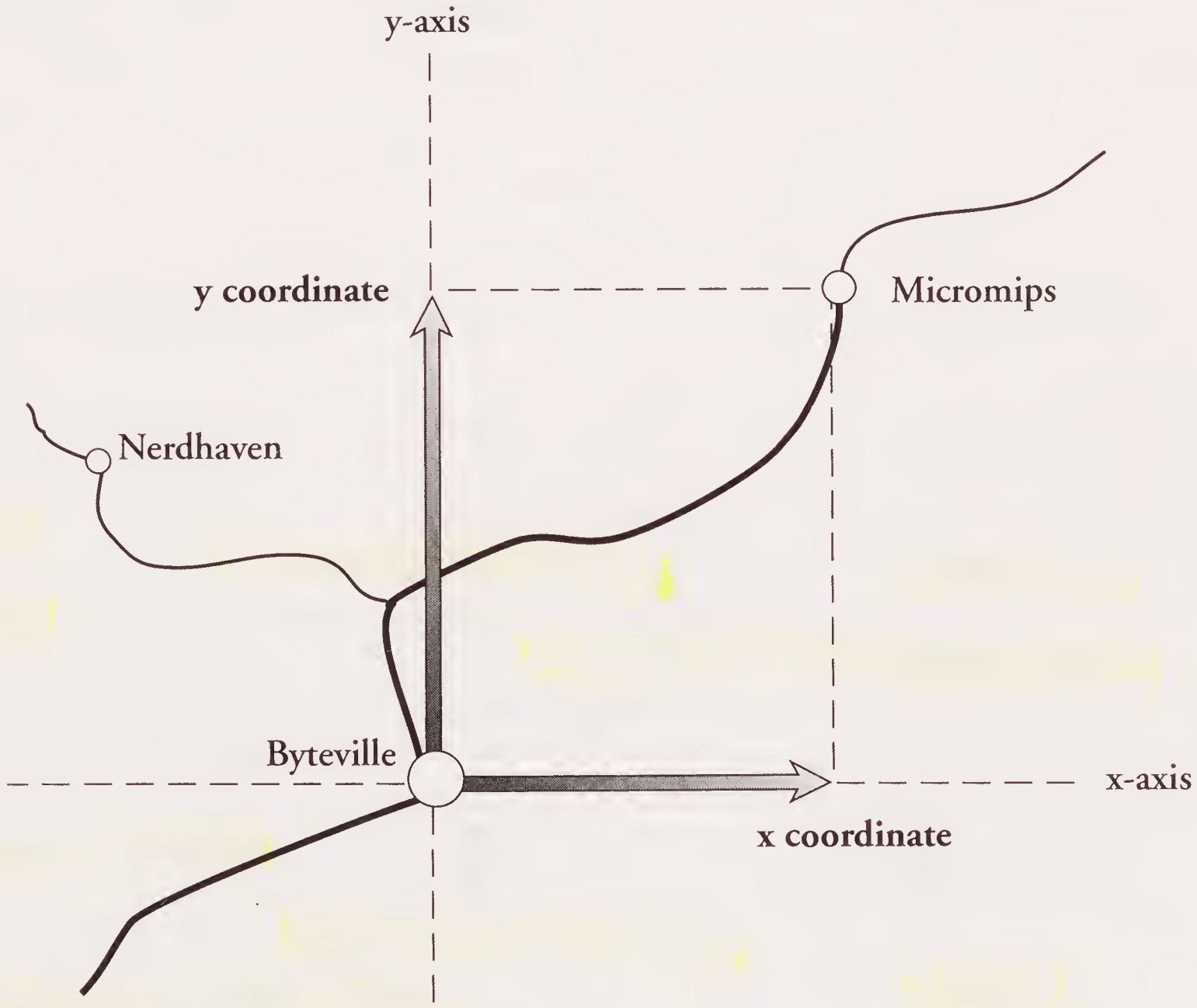
Much of what you learn about functions and C++ structures will carry over to C++ classes, so it's worth looking at a second example. This time we'll deal with space instead of time. (For simplicity, we'll ignore Einstein's unification of time and space into a space-time continuum in which what is space and what is time depends on the observer.) In particular, we'll define two structures representing two different ways of describing positions, then develop functions to convert one form to the other and to show the result. This example is a bit more mathematical than the last (and less so than a full relativistic treatment), but you don't have to follow the mathematics to follow the C++.

Suppose you want to describe the position of a point on the screen or a location on a map relative to some origin. One way is to state the horizontal offset and the vertical offset of the point from the origin. Traditionally, mathematicians use the symbol x to represent the horizontal offset and y to represent the vertical offset. See Figure 7-6.

Together, *x* and *y* constitute *rectangular coordinates.* We can define a structure consisting of two coordinates to represent a position:

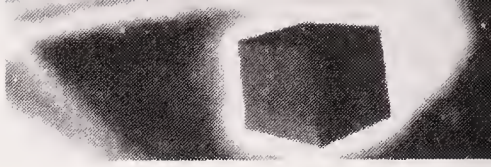
```
struct rect
{
    double x;          // horizontal distance from origin
    double y;          // vertical distance from origin
};
```

A second way to describe the position of a point is to state how far it is from the origin and in what direction it is (for example, 40 degrees north of east). Traditionally,



Rectangular coordinates of Micromips relative to Byteville

Figure 7-6 *Rectangular coordinates*



mathematicians measure the angle counterclockwise from the positive horizontal axis. See Figure 7-7. The distance and angle together constitute *polar coordinates.* We can define a second structure to represent this view of position:

```
struct polar
{
    double distance;    // distance from origin
    double angle;       // direction from origin
};
```

Let's construct a function that displays the contents of a type *polar* structure. The math functions in the C++ library assume angles are in radians, so we'll measure angles in that unit. But for display purposes, we'll convert radian measure to degrees. That means multiplying by 180/π, which is approximately 57.29577951. Here's the function:

```
// show polar coordinates, converting angle to degrees
void show_polar (polar dapos)
{
    const double Rad_to_deg = 57.29577951;

    cout << "distance = " << dapos.distance;
    cout << ", angle = " << dapos.angle * Rad_to_deg;
    cout << " degrees\n";
}
```

Notice that the formal variable is type *polar*. When you pass a *polar* structure to this function, the structure contents are copied into the *dapos* structure, and the function then uses that copy in its work. Because *dapos* is a structure, we use the membership (dot) operator (see Chapter 4) to identify structure members.

Next, let's try something more ambitious and write a function that converts rectangular coordinates to polar coordinates. We'll write the function so that we pass a *rect* structure to the function and have the function return a *polar* structure to the calling program. This involves using functions from the math library, so the program will have to include the `math.h` header file. Also, on some systems you'll have to tell the compiler to load the math library (see Chapter 1). We can use the Pythagorean theorem to get the distance from the horizontal and vertical components:

$$\text{distance} = \text{sqrt}(x \times x + y \times y)$$

The `atan2()` function from the math library calculates the angle from the *x* and *y* values:

$$\text{angle} = \text{atan2}(y, x)$$

(There's also an `atan()` function, but it doesn't distinguish between angles 180 degrees apart. That uncertainty is no more desirable in a math function than it is in a wilderness guide.)

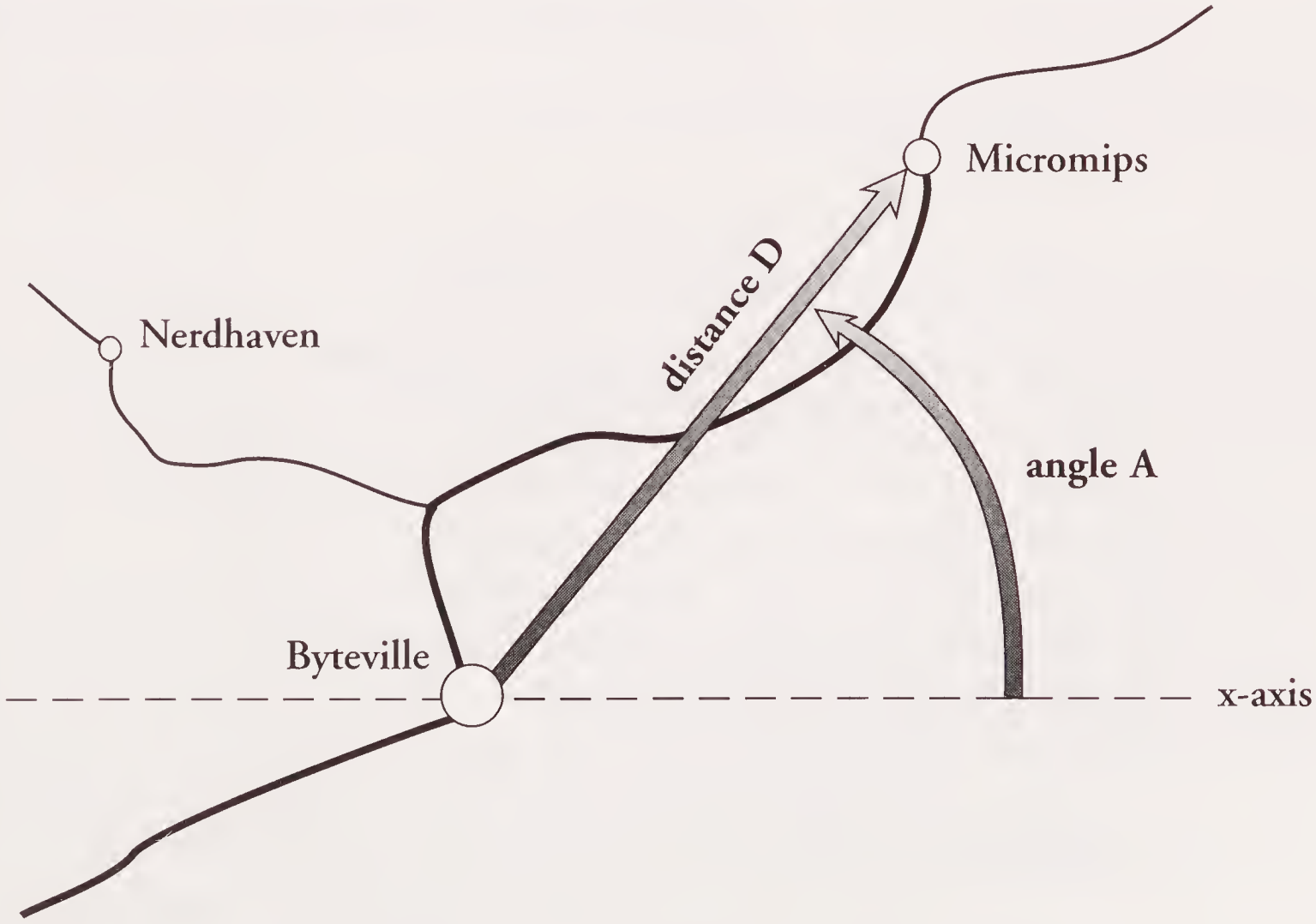
Given these formulas, we can write the function as follows:

```
// convert rectangular to polar coordinates
polar rect_to_polar(rect xypos)      // type polar
{
    polar answer;

    answer.distance =
           sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
    answer.angle = atan2(xypos.y, xypos.x);
    return answer;      // returns a polar structure
}
```

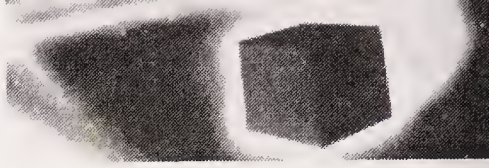
The main new point to notice is that in order to return a type *polar* structure, you have to declare the function to be type *polar*.

Now that we have the functions ready, we can write the rest of the program. Listing 7-11 presents the result.



Polar coordinates of Micromips relative to Byteville

Figure 7-7 *Polar coordinates*





LISTING 7-11 strctfun.cpp

```
// strctfun.cpp -- functions with a structure argument
#include <iostream.h>
#include <math.h>

// structure templates
struct polar
{
    double distance;     // distance from origin
    double angle;        // direction from origin
};
struct rect
{
    double x;            // horizontal distance from origin
    double y;            // vertical distance from origin
};

// prototypes
polar rect_to_polar(rect xypos);
void show_polar(polar dapos);

int main(void)
{
    rect rplace;
    polar pplace;

    cout << "Enter the x and y values: ";
    while (cin >> rplace.x >> rplace.y)  // slick use of cin
    {
            pplace = rect_to_polar(rplace);
            show_polar(pplace);
            cout << "Next two numbers (q to quit): ";
    }
    return 0;
}

// convert rectangular to polar coordinates
polar rect_to_polar(rect xypos)
{
    polar answer;

    answer.distance =
            sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
    answer.angle = atan2(xypos.y, xypos.x);
    return answer;     // returns a polar structure
}
```

```
// show polar coordinates, converting angle to degrees
void show_polar (polar dapos)
{
    const double Rad_to_deg = 57.29577951;

    cout << "distance = " << dapos.distance;
    cout << ", angle = " << dapos.angle * Rad_to_deg;
    cout << " degrees\n";
}
```

Here is a sample run:

```
Enter the x and y values: 30 40
distance = 50, angle = 53.130102 degrees
Next two numbers (q to quit): -100 100
distance = 141.421356, angle = 135 degrees
Next two numbers (q to quit): q
```

Program Notes (Including a Slick Feature)

We've already discussed the two functions, but the program has one more slick feature. It uses `cin` to control a `while` loop:

```
while (cin >> rplace.x >> rplace.y)
```

Recall that `cin` is an object of the *istream* class. The extraction operator (>>) is designed in such a way that `cin >> rplace.x` also is an object of that type. As you'll see in Chapter 10, class operators are implemented with functions. What really happens when you use `cin >> rplace.x` is that the program calls a function that returns a type *istream* value. Apply the extraction operator to the `cin >> rplace.x` object (`cin >> rplace.x >> rplace.y`), and you again get an object of the *istream* class. Thus the `while` loop test expression is an *istream* object, which normally is nonzero, hence true. If, however, the input data does not match `cin`'s expectations, the returned value is *0*, or false. In this loop, for example, `cin` expects the user to enter two numbers. If, instead, you enter *q*, as we did, *cin* >> recognizes that *q* is not a number. So it leaves the *q* in the input queue and returns a *0* value, terminating the loop.

Compare this approach for reading numbers to the one in Listing 7-7:

```
for (int i = 0; i < limit; i++)
    {
        cout << "Enter value #" << i + 1 << ": ";
        cin >> temp;
```

```
        if (temp < 0)
            break;
        ar[i] = temp;
    }
```

To terminate that loop early, you enter a negative number. That restricts input to nonnegative values. That restriction fits the needs of that program, but more typically you would want a means of terminating a loop that didn't exclude certain numeric values. Using `cin >>` as the test condition eliminates such restrictions, for it accepts all valid numeric input. So keep this trick in mind when you need an input loop for numbers.

PASSING STRUCTURE ADDRESSES

Suppose you want to save time and space by passing the address of a structure instead of passing the entire structure. This requires rewriting the functions so that they use pointers to structures. First, let's see how to rewrite the `show_polar()` function. You need to make three changes:

- When calling the function, pass it the address of the structure (*&pplace*) instead of the structure itself (*pplace*).
- Declare the formal parameter to be a pointer to *polar*, that is, type *polar **. Because the function shouldn't modify the structure, use the `const` modifier.
- Because the formal parameter is a pointer instead of a structure, use the indirect membership operator (->) instead of the membership operator (dot).

After the changes, the function looks like this:

```
// show polar coordinates, converting angle to degrees
void show_polar (const polar * pda)
{
    const double Rad_to_deg = 57.29577951;

    cout << "distance = " << pda->distance;
    cout << ", angle = " << pda->angle * Rad_to_deg;
    cout << " degrees\n";
}
```

Next, let's alter `rect_to_polar`. This is more complicated because the original `rect_to_polar` function returns a structure. To take full advantage of pointer efficiency, you should use a pointer instead of a return value. The way to do this is to pass *two* pointers to the function. The first will point to the structure to be converted, and the second will point to the structure that's to hold the conversion. So instead of returning a new structure, the function will modify an existing structure in the calling func-

tion. Otherwise, apply the same principles used to convert `show_polar()` to pointer arguments. Listing 7-12 shows the reworked program.

LISTING 7-12 strctptr.cpp

```
// strctptr.cpp -- functions with pointer to structure arguments
#include <iostream.h>
#include <math.h>

// structure templates
struct polar
{
    double distance;     // distance from origin
    double angle;        // direction from origin
};
struct rect
{
    double x;            // horizontal distance from origin
    double y;            // vertical distance from origin
};

// prototypes
void rect_to_polar(const rect * pxy, polar * pda);
void show_polar (const polar * pda);

int main(void)
{
    rect rplace;
    polar pplace;

    cout << "Enter the x and y values: ";
    while (cin >> rplace.x >> rplace.y)
    {
            rect_to_polar(&rplace, &pplace);  // pass addresses
            show_polar(&pplace);              // pass address
            cout << "Next two numbers (q to quit): ";
    }
    return 0;
}

// convert rectangular to polar coordinates
void rect_to_polar(const rect * pxy, polar * pda)
{
    pda->distance =
            sqrt(pxy->x * pxy->x + pxy->y * pxy->y);
```

```
    pda->angle = atan2(pxy->y, pxy->x);
}

// show polar coordinates, converting angle to degrees
void show_polar (const polar * pda)
{
    const double Rad_to_deg = 57.29577951;

    cout << "distance = " << pda->distance;
    cout << ", angle = " << pda->angle * Rad_to_deg;
    cout << " degrees\n";
}
```

From the user's standpoint, the program in Listings 7-12 behaves like that in Listing 7-11. The hidden difference is that 7-11 works with copies of structures, while 7-12 uses pointers to the original structures.

RECURSION

And now for something completely different. A C++ function has the interesting characteristic that it can call itself. This ability is termed *recursion.* Recursion is an important tool in certain types of programming, such as artificial intelligence, but we'll just take a superficial look (artificial shallowness) at how it works.

If a recursive function calls itself, then the newly called function calls itself, and so on ad infinitum unless the code includes something to terminate the chain of calls. The usual method is to make the recursive call part of an *if* statement. For instance, a type `void` recursive function called `recurs()` could have a form like this:

```
void recurs(argumentlist)
{
    statements1
    if (test)
        recurs(arguments)
    statements2
}
```

With luck or foresight, *test* eventually becomes false, and the chain of calls is broken.

Recursive calls produce an intriguing chain of events. As long as the *if* statement remains true, each call to `recurs()` executes *statements1*, then invokes a new incarnation of `recurs()` without reaching *statements2*. When the `if` statement becomes false, the current call then proceeds to *statements2*. Then, when the current call terminates,

program control returns to the previous version of `recurs()` that called it. Then that version of `recurs()` completes executing its *statements2* section and terminates, returning control to the prior call, and so on. Thus if `recurs()` undergoes five recursive calls, first the *statements1* section is executed five times in the order in which the functions were called, then the *statements2* section is executed five times opposite the order in which the functions were called. After going in five levels of recursion, the program then has to back out through the same five levels. Listing 7-13 illustrates this behavior.

LISTING 7-13 recur.cpp

```
// recur.cpp -- use recursion
#include <iostream.h>
void countdown(int n);

int main(void)
{
    countdown(4);                    // call the recursive function
    return 0;
}

void countdown(int n)
{
    cout << "Counting down ... " << n << "\n";
    if (n > 0)
            countdown(n-1);          // function calls itself
    cout << n << ": Kaboom!\n";
}
```

Here's the output:

```
Counting down ... 4          ← level 1 — beginning to add levels of recursion
Counting down ... 3          ← level 2
Counting down ... 2          ← level 3
Counting down ... 1          ← level 4
Counting down ... 0          ← level 5
0: Kaboom!                   ← level 5 — beginning to back out through the series of calls
1: Kaboom!                   ← level 4
2: Kaboom!                   ← level 3
3: Kaboom!                   ← level 2
4: Kaboom!                   ← level 1
```

Note that each recursive call creates its own set of variables, so by the time the program reaches the fifth call, it has five separate variables called *n*, each with a different value.

SUMMARY

Functions are the C++ programming modules. To use a function, you need to provide a definition and a prototype, and you have to use a function call. The function definition is the code that implements what the function does. The function prototype describes the function interface: how many and what kinds of values to pass to the function and what sort of return type, if any, to get from it. The function call causes the program to pass the function arguments to the function and to transfer program execution to the function code.

By default, C++ functions pass arguments by value. This means that the formal parameters in the function definition are new variables that are initialized to the values provided by the function call. Thus C++ functions protect the integrity of the original data by working with copies.

C++ treats an array name argument as the address of the first element of the array. Technically, this is still passing by value, for the pointer is a copy of the original address. But the function uses the pointer to access the contents of the original array. When declaring formal parameters for a function (and *only* then), the following two declarations are equivalent:

```
typename arr[];
typename * arr;
```

Both mean *arr* is a pointer to `typename`. When writing the function code, however, you can use *arr* as if it were an array name in order to access elements: `arr[i]`. Even when passing pointers, you can preserve the integrity of the original data by declaring the formal argument to be a pointer to a `const` type. Because passing the address of an array conveys no information about the size of the array, you normally would pass the array size as a separate argument.

C++ provides three ways to represent strings: a character array, a string constant, and a pointer to a string. All are type `char*` (pointer-to-`char`), so they are passed to a function as a type `char*` argument. C++ uses the null character (`'\0'`) to terminate strings, and string functions use that fact to determine the end of any string they are processing.

C++ treats structures the same as basic types, meaning that you can pass them by value and use them as function return types. However, if the structure is large, it may be more efficient to pass a pointer to the structure and let the function work with the original data.

A C++ function can be recursive; that is, the code for a particular function can include a call of itself.

Review Questions

1. What are the three steps in using a function?
2. Construct function prototypes that match the following descriptions:
   a. `igor()` takes no arguments and has no return value.
   b. `tofu()` takes an `int` argument and returns a `float`.
   c. `mpg()` takes two type `double` arguments and returns a `double`.
   d. `summation()` takes the name of a `long` array and an array size as values and returns a `long` value.
   e. `doctor()` takes a string argument and returns a `double` value.
   f. `ofcourse()` takes a *boss* structure as an argument and returns nothing.
   g. `plot()` takes a pointer to a *map* structure as an argument and returns a string.
3. Write a function that takes three arguments: the name of an `int` array, the array size, and an `int` value. Have the function set each element of the array to the `int` value.
4. Write a function that takes a `double` array name and an array size as arguments and that returns the largest value in that array. Note that this function shouldn't alter the contents of the array.
5. Why don't we use the `const` qualifier for function arguments that are one of the fundamental types?
6. What are the three forms a string can take in a C++ program?
7. Write a function having this prototype:

   ```
   int replace(char * str, char c1, char c2);
   ```

   Have the function replace every occurrence of *c1* in the string *str* with *c2*, and have the function return the number of replacements it makes.
8. What does the expression `*"pizza"` mean? What about `"taco"[2]`?
9. C++ lets you pass a structure by value and it lets you pass the address of a structure. If *glitz* is a structure variable, how would you pass it by value? How would you pass its address? What are the trade-offs of the two approaches?

10. Here is a structure template:

```
struct box
{
   char maker[40];
   float height;
   float width;
   float length;
   float volume;
};
```

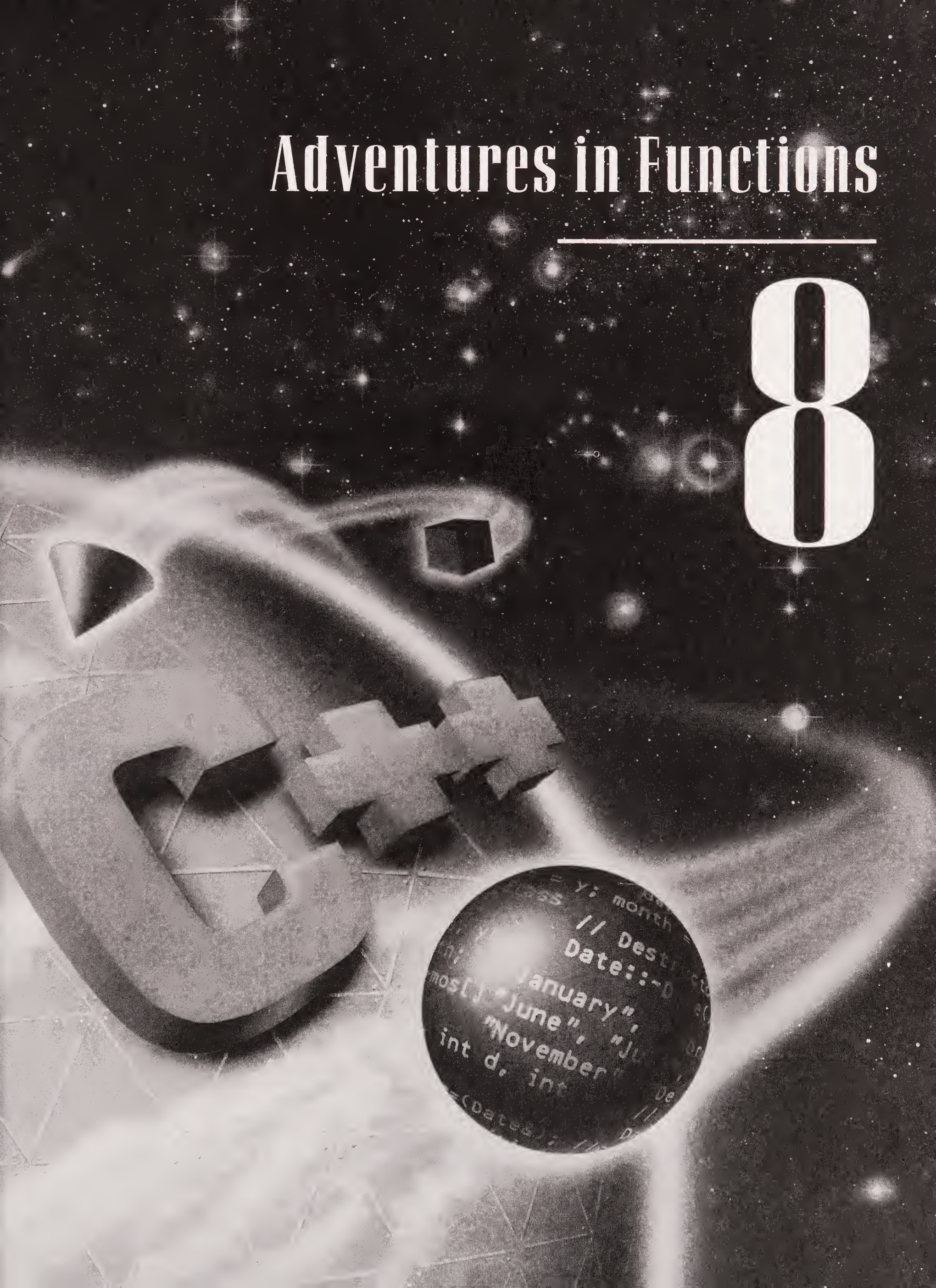
   a. Write a function that passes a *box* structure by value and that displays the value of each member.

   b. Write a function that passes the address of a *box* structure and that sets the *volume* member to the product of the other three dimensions.

11. Define a recursive function that takes an integer argument and returns the factorial of that argument. Recall that 3 factorial, written 3!, equals 3 x 2!, and so on, with 0! defined as 1. In general, n! = n * (n - 1)!.

Adventures in Functions

8



With the last chapter under your belt, you now know a lot about C++ functions, but there's much more to come. C++ provides many new function features that separate C++ from its C heritage. The new features include inline functions, default argument values, function overloading (polymorphism), and passing variables by reference. This chapter discusses these C++ enhancements to functions. Also, it examines multifile programs, C++'s varieties of storage classes, and pointers to functions. This chapter, more than any other you've read so far, explores features found in C++ but not C, so it marks our first major foray into plus-plussedness.

Inline Functions

Let's begin by examining *inline* functions, a C++ enhancement designed to speed up programs. The primary distinction between normal functions and inline functions is not in how you code them but in how the C++ compiler incorporates them into a program. To understand the distinction between inline functions and normal functions, you need to peer more deeply into a program's innards than we have so far. It's really not all that messy (you'll just be peering, not groping), so let's do that now.

The final product of the compilation process is an executable program, which consists of a set of machine language instructions. When you start a program, the operating system loads these instructions into the computer's memory, so each instruction has a particular memory address. The computer then goes through these instructions step by step. Sometimes, as when you have a loop or a branching statement, program execution will skip over instructions, jumping back or forward to a particular address. Normal function calls also involve having a program jump to another address (the function's address), then jumping back when the function terminates. Let's look at

that process in a little more detail. When the program reaches the function call instruction, the program stores the memory address of the instruction immediately following the function call, copies arguments to the stack (a block of memory reserved for that purpose), jumps to the memory location that marks the beginning of the function, executes the function code, perhaps placing a return value in a register, then jumps back to the instruction whose address it saved.[1] Jumping back and forth and keeping track of where to jump means that there is an overhead in elapsed time to using functions.

The C++ *inline* function provides an alternative. This is a function whose compiled code is "in line" with the other code in the program. That is, the compiler replaces the function call with the corresponding function code. With inline code, the program doesn't have to jump to another location to execute the code, then jump back. Inline functions thus run a little faster than regular functions. But there is a memory penalty. If a program calls an inline function ten times, then the program winds up with ten copies of the function inserted into the code. See Figure 8-1.

Most programmers recommend using inline functions sparingly, if at all. The speed gain is usually minimal unless the function itself is so short that the time needed to execute the function is comparable to the time spent jumping to and from the function. In that case, the function is already fast, so about the only time you would get much of a benefit is if the function were the main time-consumer in a crucial loop.

If you must use this feature, you have to do two things:

- Preface the function definition with the keyword *inline*.
- Place the function definition above all functions that call it.

Note that you have to place the entire definition (meaning the function header and all the function code), not just the prototype, above the other functions.

The compiler does not have to honor your request to make a function inline. It may decide the function is too large or notice that it calls itself (recursion is not allowed for inline functions), or the feature may not be implemented for your particular compiler.

Listing 8-1 illustrates the inline method with an inline `square()` function that squares its argument. Note that we've placed the entire definition on one line. That's not required, but if the definition doesn't fit on one line, the function probably is a poor candidate for being an inline function.



LISTING 8-1 inline.cpp

```
// inline.cpp -- use an inline function
#include <iostream.h>
```

[1] It's a bit like having to leave off reading some text to find out what a footnote says, then, upon finishing the footnote, returning to where you were reading in the text.

```
// an inline function must be defined before first use
inline double square(double x) { return x * x; }

int main(void)
{
    double a, b;
    double c = 13.0;

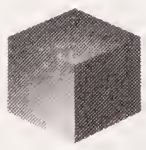
    a = square(5.0);
    b = square(4.5 + 7.5);   // can pass expressions
    cout << "a = " << a << ", b = " << b << "\n";
    cout << "c = " << c;
    cout << ", c squared = " << square(c++) << "\n";
    cout << "Now c = " << c << "\n";
}
```

Here's the output:

```
a = 25, b = 144
c = 13, c squared = 169
Now c = 14
```

The output illustrates that inline functions pass arguments by value just like regular functions do. If the argument is an expression such as *4.5 + 7.5*, the function passes the value of the expression, *12* in this case. This makes C++'s inline facility far superior to C's macro definitions. See the discussion on Inline versus Macros.

Even though we didn't provide a separate prototype, C++'s prototyping features are still in play. That's because the entire definition, coming before the function's first use, serves as a prototype. This means you can use `square()` with an `int` argument or a `long` argument, and the program will automatically type cast the value to type `double` before passing it to the function.

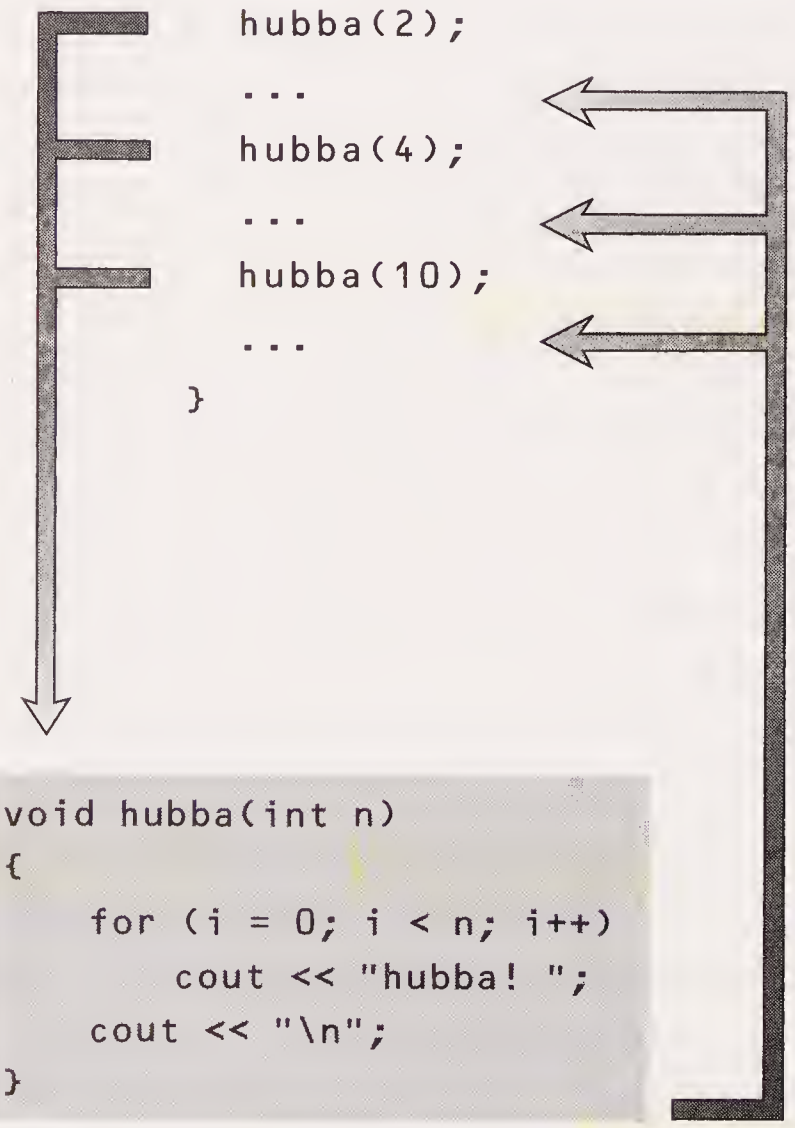


INLINE VERSUS MACROS

The `inline` facility is a C++ addition. C uses the preprocessor `#define` statement to provide *macros*, a crude implementation of inline code. For instance, here's a macro for squaring a number:

```
#define SQUARE(X) X*X
```

This works not by passing arguments but by text substitution, with the *X* acting as a symbolic label for the "argument":



A regular function transfers program execution to a separate function.

```
...
int main(void)
{
  ...
  {
    n = 2;
    for (i = 0; i < n; i++)
        cout << "hubba! ";
    cout << "\n";
  }
  ...
  {
    n = 4;
    for (i = 0; i < n; i++)
        cout << "hubba! ";
    cout << "\n";
  }
  ...
  {
    n = 10;
    for (i = 0; i < n; i++)
        cout << "hubba! ";
    cout << "\n";
  }
  ...
}
```

An inline function replaces a function call with inline code.

Figure 8-1 *Inline functions versus regular functions*

```
a = SQUARE(5.0); is replaced by a = 5.0*5.0;
b = SQUARE(4.5 + 7.5); is replaced by
b = 4.5 + 7.5 * 4.5 + 7.5;
SQUARE(c++) is replaced by c++*c++
```

Only the first example works properly. You can improve matters with a liberal application of parentheses:

```
#define SQUARE(X) ((X)*(X))
```

Still, the problem remains that macros don't pass by value. Even with this new definition, `SQUARE(c++)` increments *c* twice. But our inline `square()` function in Listing 8-1 evaluates *c*, passes that value to be squared, then increments *c* once.

The intent here is not to show you how to write C macros. Rather, it is to suggest that if you have been using C macros, consider converting them to C++ inline functions.

Default Arguments

Let's look at another topic from C++'s bag of new tricks—the *default argument*. A default argument is a value that's used automatically if you omit the corresponding actual argument from a function call. For example, if you set up `void wow(int n)` so that *n* has a default value of 1, then the function call `wow()` is the same as `wow(1)`. This gives you greater flexibility in how you use a function. Suppose you have a function called `left()` that returns the first *n* characters of a string, with the string and *n* being arguments. More precisely, the function will return a pointer to a new string consisting of the selected portion of the original string. For instance, the call `left("theory", 3)` would construct a new string *"the"* and return a pointer to it. Now suppose you establish a default value of *1* for the second argument. The call `left("theory", 3)` would work as before, with your choice of *3* overriding the default. But the call `left("theory")`, instead of being an error, would assume a second argument of *1* and return a pointer to the string *"t"*. This kind of default is helpful if your program often needs to extract a one-character string but occasionally needs to extract longer strings.

How do you establish a default value? You must use the function prototype. Because the compiler looks at the prototype to see how many arguments a function uses, the function prototype *also* has to alert the program to the possibility of default arguments. The method is to assign a value to the argument in the prototype. For instance, here's the prototype fitting this description of `left()`:

```
char * left(const char * str, int n = 1);
```

We want the function to return a new string, so its type is `char*`, or pointer-to-`char`. We want to leave the original string unaltered, so we use the `const` qualifier for the first argument. We want *n* to have a default value of *1*, so we assign that value to *n*. A default argument value is essentially an initialization value. Thus the prototype above initializes *n* to the value *1*. So if you leave *n* alone, it has the value *1*, but if you pass an argument, the new value overwrites the *1*.

When you use a function with an argument list, you must add defaults from right to left. That is, you can't provide a default value for a particular argument unless you also provide defaults for all the arguments to its right:

```
int harpo(int n, int m = 4, int j = 5);          // VALID
int chico(int n, int m = 6, int j);              // INVALID
int groucho(int n = 1, int m = 2, int n = 3);    // VALID
```

Listing 8-2 puts default arguments to use. Note that *only* the prototype indicates the default. The function definition is the same as it would have been without default arguments.

LISTING 8-2 left.cpp

```
// left.cpp -- string function with a default argument
#include <iostream.h>
const int Size = 80;
char * left(const char * str, int n = 1);
int main(void)
{
    char sample[Size];
    cout << "Enter a string:\n";
    cin.get(sample,Size);
    char *ps = left(sample, 4);
    cout << ps << "\n";
    ps = left(sample);
    cout << ps << "\n";
    return 0;
}

// This function returns a pointer to a new string
// consisting of the first n characters in the str string.
char * left(const char * str, int n)
{
    if(n < 0)
            n = 0;
    char * p = new char[n+1];
    for (int i = 0; i < n && str[i]; i++)
            p[i] = str[i];      // copy characters
    while (i <= n)
            p[i++] = '\0';      // set rest of string to '\0'
    return p;
}
```

Here's a sample run:

```
Enter a string:
bestir
best
b
```

Program Notes

The program uses `new` to create a new string for holding the selected characters. One awkward possibility is that an uncooperative user requests a negative number of characters. In that case, the function sets the character count to *0* and eventually returns the null string. Another awkward possibility is that an irresponsible user requests more characters than the string contains. The function protects against this by using a combined test:

```
i < n && str[i]
```

The `i<n` test stops the loop after *n* characters have been copied. The second part of the test, the expression `str[i]`, is the code for the character about to be copied. If the loop reaches the null character, the code is zero, and the loop terminates. The final `while` loop terminates the string with the null character, then sets the rest of the allocated space, if any, to null characters.

Another approach for setting the size of the new string would be to set *n* to the smaller of the passed value and the string length:

```
int len = strlen(str);
n = (n < len) ? n : len; // the lesser of n and len
char * p = new char[n+1];
```

This ensures that `new` doesn't allocate more space than what's needed to hold the string. That can be useful if you make a call like `left("Hi!", 32767)`. Our first approach copies the *"Hi!"* into an array of 32767 characters, setting all but the first three characters to the null character. The second approach copies *"Hi!"* into an array of four characters. But, by adding another function call (`strlen()`), it increases the program size, slows the process, and requires that you remember to include the `string.h` header file. C programmers have tended to opt for faster running, more compact code and leaving a greater burden on the programmer to use functions correctly. The C++ tradition, however, places greater weight on reliability. After all, a slower program working correctly is better than a fast program that works incorrectly. If that time taken to call `strlen()` turns out to be a problem, you can let `left()` determine the lesser of *n* and the string length directly. For instance, the following loop quits when *m* reaches *n* or the end of the string, whichever comes first:

```
int m = 0;
while ( m <= n && str[m] != '\0')
    m++;
char * p = new char[m+1]:
// use m instead of n in rest of code
```

Function Polymorphism (Function Overloading)

If you think default arguments are neat (cool, rad, super), *function polymorphism* will really turn you on. Default arguments let you call the same function using varying numbers of arguments. Function polymorphism, also called *function overloading*, lets you use multiple functions sharing the same name. The expression "polymorphism" means having many forms, so function polymorphism lets a function have many forms. Similarly, the expression "function overloading" means you can attach more than one function to the same name, thus overloading the name. Both expressions boil down to the same thing, but we'll usually use the expression function overloading—it sounds harder working. You can use function overloading to design a family of functions that do essentially the same thing, but using different argument lists.

Overloaded functions are analogous to verbs having more than one meaning. For instance, Miss Piggy can root at the ball park for the home team, and pigs can root in the soil for potatoes. The context (one hopes) tells you which meaning of *root* is intended in each case. Similarly, C++ uses the context to decide which version of an overloaded function is intended.

The key to function overloading is a function's argument list, also called the function *signature*. If two functions use the same number and types of arguments in the same order, they have the same signature; the variable names don't matter. C++ lets you define two functions by the *same* name providing the functions have *different* signatures. The signature can differ in the number of arguments or in the type of arguments, or both. For instance, you could define a set of `print()` functions with the following prototypes:

```
void print(const char * str, int width);     // #1
void print(double d, int width);             // #2
void print(long l, int width);               // #3
void print(int i, int width);                // #4
void print(const char *str);                 // #5
```

When you then use a `print()` function, the compiler matches your use to the prototype having the same signature:

```
print("Pancakes", 15);                       // use #1
print("Syrup");                              // use #5
print(1995.0, 10);                           // use #2
print(1995, 12);                             // use #4
print(1995L, 15);                            // use #3
```

For instance, `print("Pancakes", 15)` uses a string and an integer as arguments, and that matches prototype #1.

When using overloaded functions, be sure you use the proper argument types in the function call. For instance, consider the following statements:

```
unsigned int year = 3210;      // ambiguous call
print(year, 6);
```

Which prototype does the `print()` call match here? It doesn't match any of them. A lack of a matching prototype doesn't automatically rule out using one of the functions, for C++ will try to use standard type conversions to force a match. If, say, the only `print()` prototype were #2, the function call `print(year, 6)` would convert the *year* value to type `double`. But in reality, there are *three* prototypes that take a number as the first argument, providing three different choices for converting *year*. Faced with this ambiguous situation, C++ will reject the function call as an error.

Keep in mind that it's the signature, not the function type, that enables function overloading. For instance, the following two declarations are incompatible:

```
long gronk(int n, float m);          // same signatures,
double gronk(int n, float m);        // hence not allowed
```

Therefore C++ won't permit you to overload `gronk()` in this fashion. You can have different return types, but only if the signatures also are different:

```
long gronk(int n, float m);          // different signatures,
double gronk(float n, float m);      // hence allowed
```

AN OVERLOADING EXAMPLE

We've already developed a `left()` function that returns a pointer to the first *n* characters in a string. Let's add a second `left()` function, one that returns the first *n* digits in an integer. You could use it, for example, to examine the first three digits of a U.S. postal zip code stored as an integer, a useful act if you wish to sort for urban areas.

The integer function is a bit more difficult to program than the string version because we don't have the benefit of each digit stored in its own array element. One approach is to first compute the number of digits in the number. Dividing a number by 10 lops off one digit, so you can use division to count digits. More precisely, you can do so with a loop like this:

```
unsigned digits = 1;
while (n /= 10)
      digits++;
```

This loop counts how many times you can remove a digit from *n* until none are left. Recall that *n /= 10* is short for *n = n / 10*. If *n* is *8*, for example, the test condition assigns to *n* the value *8 / 10*, or *0*, since it's integer division. That terminates the loop, and *digits* remains at *1*. But if *n* is *238*, the first loop test sets *n* to *238 / 10*, or *23*. That's nonzero,

so the loop increases *digits* to *2*. The next cycle sets *n* to *23 / 10*, or *2*. Again, that's nonzero, so *digits* grows to *3*. The next cycle sets *n* to *2 / 10*, or *0*, and the loop quits, leaving *digits* set to the correct value, *3*.

Now suppose you know the number has 5 digits and you want to return the first 3 digits. You can get that value by dividing the number by 10, then dividing the answer by 10 again. Each division by 10 lops one more digit off the right end. We can calculate the number of digits to lop—just subtract the number of digits to be shown from the total number of digits. For instance, to show 4 digits of a 9-digit number, lop off the last 5 digits. You can code this approach as follows:

```
ct = digits - ct;
while (ct--)
    num /= 10;
return num;
```

Listing 8-3 incorporates this code into a new `left()` function. The function includes some additional code to handle special cases, such as asking for zero digits or asking for more digits than the number possesses. Because the signature of the new `left()` differs from that of the old `left()`, we can, and do, use both functions in the same program.

LISTING 8-3 leftover.cpp

```
// leftover.cpp -- overloading the left() function
#include <iostream.h>
unsigned long left(unsigned long num, unsigned ct);
char * left(const char * str, int n = 1);

int main(void)
{
     char * trip = "Hawaii!!";          // test value
     unsigned long n = 12345678;        // test value
    int i;

     for (i = 1; i < 10; i++)
    {
            cout << left(n, i) << "\n";
            cout << left(trip, i) << "\n";
    }
    return 0;

}

// This function returns the first ct digits of the number num.
unsigned long left(unsigned long num, unsigned ct)
```

```
{
    unsigned digits = 1;
    unsigned long n = num;

    if (ct == 0 || num == 0)
        return 0;                    // return 0 if no digits
    while (n /= 10)
        digits++;
     if (digits > ct)
    {
        ct = digits - ct;
        while (ct--)
            num /= 10;
        return num;                  // return left ct digits
     }
    else                             // if ct >= number of digits
        return num;                  // return the whole number
}

// This function returns a pointer to a new string
// consisting of the first n characters in the str string.
char * left(const char * str, int n)
{
    if(n < 0)
            n = 0;
    char * p = new char[n+1];
    for (int i = 0; i < n && str[i]; i++)
            p[i] = str[i];           // copy characters
    while (i <= n)
            p[i++] = '\0';           // set rest of string to '\0'
    return p;
}
```

Here's the output:

```
1
H
12
Ha
123
Haw
1234
Hawa
12345
Hawai
```

```
123456
Hawaii
1234567
Hawaii!
12345678
Hawaii!!
12345678
Hawaii!!
```

WHEN TO USE FUNCTION OVERLOADING

You may find function overloading fascinating, but don't overuse the facility. You should reserve function overloading for functions that perform basically the same task but with different forms of data. Also, you may want to check whether you can accomplish the same end with default arguments. For example, you can replace the single, string-oriented `left()` function with two overloaded functions:

```
char * left(const char * str, unsigned n);  // two arguments
char * left(const char * str);              // one argument
```

But using the single function with a default argument is simpler. There's just one function to write, instead of two, and the program will require memory for just one function instead of two. The function overloading approach, however, is the more powerful, for it enables you to use different types of arguments as well as different numbers of arguments.

REFERENCE VARIABLES

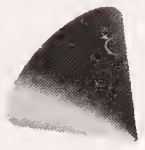
C++ adds a new derived type to the language—the *reference* variable. A reference is a name that acts as a name, or alternative name, for a previously defined variable. For instance, if you make *twain* a reference to the *clemens* variable, you can use *twain* and *clemens* interchangeably to represent that variable. Of what use is such an alias? Is it to help people who are embarrassed by their choice of variable names? Maybe, but the main use for a reference is as a formal argument to a function. By using a reference as an argument, the function works with the original data instead of with a copy. References provide a convenient alternative to pointers for processing large structures with a function, and they are essential to designing classes. Before seeing how to use references with functions, however, let's examine the basics of defining and using a reference.

CREATING A REFERENCE VARIABLE

You may recall that C and C++ use the `&` symbol to indicate the address of a variable. C++ assigns an additional meaning to the `&` symbol and presses it into service for declaring references. For example, to make *rodents* an alternative name for the variable *rats*, do the following:

```
int rats;
int & rodents = rats;    // makes rodents an alias for rats
```

In this context, `&` is *not* the address operator. Instead, it serves as part of the type identifier. Just as `char*` in a declaration means pointer-to-`char`, `int &` means reference-to-an-`int`. The reference declaration allows you to use *rats* and *rodents* interchangeably; both refer to the same value and the same memory location. Listing 8-4 illustrates the truth of this claim.



LISTING 8-4 firstref.cpp

```
// firstref.cpp -- defining and using a reference
#include <iostream.h>
int main(void)
{
    int rats = 101;
    int & rodents = rats;   // rodents is a reference

    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << "\n";
    rodents++;
    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << "\n";

// some implementations require type casting the following
// addresses to type unsigned
    cout << "rats address = " << &rats;
    cout << ", rodents address= " << &rodents << "\n";
    return 0;
}
```

Note that the `&` operator in the statement

```
int & rodents = rats;
```

is not the address operator, but declares that *rodents* is of type `int &`, that is, a reference to an `int` variable. But the `&` operator in the statement

```
cout<<", rodents address="<< &rodents << "\n";
```

is the address operator, with `&rodents` representing the address of the variable to which *rodents* refers. Here is the program's output:

```
rats = 101, rodents = 101
rats = 102, rodents = 102
rats address = 0x8fa2fff4, rodents address= 0x8fa2fff4
```

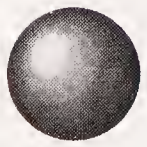
As you can see, both *rats* and *rodents* have the same value and the same address. And incrementing *rodents* by 1 affects both variables.(Keep in mind that while this example shows you how a reference works, it doesn't represent the typical use for a reference, which is as a function parameter, particularly for structure and object arguments. We'll look into these uses pretty soon.)

References tend to be a bit confusing to C veterans coming to C++ because they are tantalizingly reminiscent of pointers, yet somehow different. For instance, you can create both a reference and a pointer to refer to *rats*:

```
int rats = 101;
int & rodents = rats;   // rodents a reference
int * prats = &rats;    // prats a pointer
```

Then you could use the expressions *rodents* and **prats* interchangeably with *rats* and use the expressions *&rodents* and *prats* interchangeably with *&rats*. From this standpoint, a reference looks a lot like a pointer in disguised notation in which the * dereferencing operator is understood implicitly. And, in fact, that's more or less what a reference is. But there are differences besides those of notation. For one, in our example it was necessary to initialize the reference when declaring it; you can't declare the reference, then assign it a value later, the way you can with a pointer:

```
int rat;
int & rodent;
rodent = rat;   // No, you can't do this.
```



You should initialize a reference variable when you declare it. (You can consider a reference parameter in a function's argument list as being initialized to the passed argument.)

Also, a reference is more like a `const` pointer; once a reference pledges its allegiance to a particular variable, it sticks to its pledge. Listing 8-5 shows what happens if you try to make a reference change allegiance from a *rats* variable to a *bunnies* variable.

LISTING 8-5 secref.cpp

```
// secref.cpp -- defining and using a reference
#include <iostream.h>
int main(void)
{
    int rats = 101;
    int & rodents = rats;   // rodents is a reference

    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << "\n";

// some implementations require type casting the following
// addresses to type unsigned
    cout << "rats address = " << &rats;
    cout << ", rodents address= " << &rodents << "\n";

    int bunnies = 50;
    rodents = bunnies;      // can we change the reference?
    cout << "bunnies = " << bunnies;
    cout << ", rats = " << rats;
    cout << ", rodents = " << rodents << "\n";
// some implementations require type casting the following
// addresses to type unsigned
    cout << "bunnies address = " << &bunnies;
    cout << ", rodents address= " << &rodents << "\n";
    return 0;
}
```

Here's the output:

```
rats = 101, rodents = 101
rats address = 0x8f87fff4, rodents address= 0x8f87fff4
bunnies = 50, rats = 50, rodents = 50
bunnies address = 0x8f87fff2, rodents address= 0x8f87fff4
```

Initially, *rodents* refers to *rats*, but then the program attempts to make *rodents* a reference to *bunnies*:

```
rodents = bunnies;
```

For a moment, it looks as if this attempt has succeeded, for the value of *rodents* changes from *101* to *50*. But closer inspection reveals that *rats* also has changed to *50* and that *rats* and *rodents* still share the same address, which differs from the *bunnies* address. Because we made *rodents* an alias for *rats*, our assignment statement really means the same as the following:

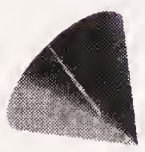
```
rats = bunnies;
```

That is, it means "assign the value of the *bunnies* variable to the *rat* variable."

REFERENCES AS FUNCTION PARAMETERS

Most often, references are used as function parameters, making a variable name in the function an alias for a variable in the calling program. This method of passing arguments is called *passing by reference*. Passing by reference allows a called function to access variables in the calling function. C++'s addition of the feature is a break from C, which only passes by value. Passing by value, recall, results in the called function working with copies of values from the calling program. See Figure 8-2. Of course, C lets you get around the passing by value limitation by using pointers.

Let's compare using references and using pointers in a common computer problem, swapping the values of two variables. A swapping function has to be able to alter values of variables in the calling program. That means the usual approach of passing variables by value won't work, since the function will end up swapping the contents of copies of the original variables instead of the variables themselves. If you pass references, however, the function can work with the original data. Or you can pass pointers in order to access the original data. Listing 8-6 shows all three methods, including the one that doesn't work, so that you can compare them.



LISTING 8-6 swaps.cpp

```
// swaps.cpp -- swapping with references and with pointers
#include <iostream.h>
void swapr(int & a, int & b);   // a, b are aliases for ints
void swapp(int * p, int * q);   // p, q are addresses of ints
void swapv(int a, int b);       // a, b are new variables
int main(void)
{
    int wallet1 = 300;
    int wallet2 = 350;

    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << "\n";

    cout << "Using references to swap contents:\n";
    swapr(wallet1, wallet2);   // pass variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << "\n";
```

```
    cout << "Using pointers to swap contents:\n";
    swapp(&wallet1, &wallet2); // pass addresses of variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << "\n";

    cout << "Trying to use passing by value:\n";
    swapv(wallet1, wallet2);  // pass values of variables
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << "\n";
    return 0;
}

void swapr(int & a, int & b)    // use references
{
    int temp;

    temp = a;     // use a, b for values of variables
    a = b;
    b = temp;
}

void swapp(int * p, int * q)    // use pointers
{
    int temp;

    temp = *p;     // use *p, *q for values of variables
    *p = *q;
    *q = temp;
}

void swapv(int a, int b)        // try using values
{
    int temp;

    temp = a;     // use a, b for values of variables
    a = b;
    b = temp;
}
```

Here's the output:

```
wallet1 = $300 wallet2 = $350                    ←original values
Using references to swap contents:
wallet1 = $350 wallet2 = $300                    ←values swapped
```

```
Using pointers to swap contents:
wallet1 = $300 wallet2 = $350          ← values swapped again
Trying to use passing by value:
wallet1 = $300 wallet2 = $350          ← swap failed
```

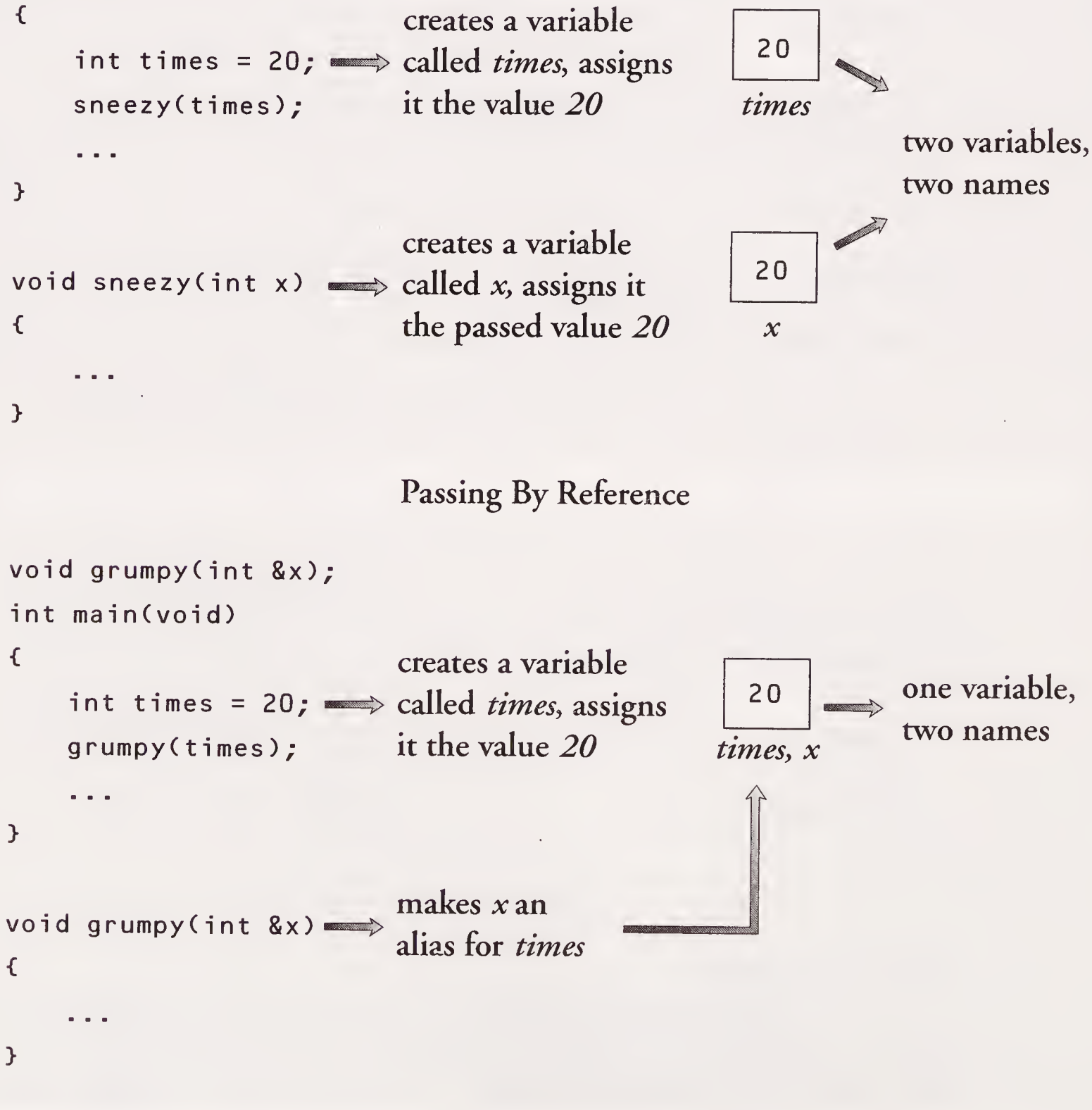


Figure 8-2 *Passing by value and passing by reference*

As we expected, the reference method and pointer method successfully swapped the contents of the two wallets, while the passing by value method failed.

Program Notes

First, note how each function is called:

```
swapr(wallet1, wallet2);      // pass variables
swapp(&wallet1, &wallet2);    // pass addresses of variables
swapv(wallet1, wallet2);      // pass values of variables
```

Passing by reference (`swapr(wallet1, wallet2)`) and passing by value (`swapv(wallet1, wallet2)`) look identical. The only way you can tell that `swapr()` passes by reference is by looking at the prototype or at the function definition. However, the presence of the address operator (`&`) makes it obvious when a function passes by address (`swapp(&wallet1, &wallet2)`). (Recall that the type declaration `int*p` means that *p* is a pointer to an `int` and therefore the argument corresponding to *p* should be an address, like `&wallet1`.)

Next, compare the code for the two functions `swapr()` (passing by reference) and `swapv()` (passing by value). The only outward difference between the two is how the function parameters are declared:

```
void swapr(int & a, int & b)
void swapv(int a, int b)
```

The internal difference, of course, is that in `swapr()`, the variables *a* and *b* serve as aliases for *wallet1* and *wallet2*, so swapping *a* and *b* swaps *wallet1* and *wallet2*. But in `swapv()`, the variables *a* and *b* are new variables that copy the values of *wallet1* and *wallet2*, so swapping *a* and *b* has no effect on *wallet1* and *wallet2*.

Finally, compare the functions `swapr()` (passing a reference) and `swapp()` (passing a pointer). The first difference is in how the function parameters are declared:

```
void swapr(int & a, int & b)
void swapp(int * p, int * q)
```

The second difference is that the pointer version requires using the * dereferencing operator throughout when the function uses *p* and *q*. Although the reference version and the pointer version are equally effective, the reference version has a cleaner look.

Reference Properties and Oddities

Interesting and surprising things can happen when you use reference arguments. Suppose, for instance, you define a function `byref()` that has a type `int &` (reference-to-

`int`) argument. Clearly, you can use a type `int` variable for an argument when calling the function. But there are many other possibilities, as Listing 8-7 shows. In it, the `byref()` function uses a loop to print and increment its argument three times. This lets us see what the value of the reference variable *rx* is, and it also lets us examine the effect incrementing *rx* has upon the original argument.

LISTING 8-7 refexamp.cpp

```
// refexamp.cpp -- varieties of reference arguments
#include <iostream.h>
void byref(int & rx);  // rx is a reference

int main(void)
{
// some implementations require static int vals[5] to
// enable array initialization
    int vals[5] = {2,4,6,8,10};

// some implementations require static struct parts to
// enable structure initialization
    struct parts {
            int a;
            float b;
            } thing = { 600, 3.14 };
    int x = 20;

    byref(vals[4]);      // pass an int element of an array
    cout << "vals[4] =   " << vals[4] << " after using byref()\n";
    byref(thing.a);      // pass an int member of a structure
    cout << "thing.a =   " << thing.a << " after using byref()\n";
    byref(101);          // pass a constant
    byref(2 * x + 3);    // pass an expression
    cout << "x = " << x << " after using byref()\n";
    long ing = 15;
    byref(ing);          // pass the wrong type variable
    cout << "ing = " << ing << " after using byref()\n";
    return 0;
}

void byref(int & rx)
{
    for(int i = 0; i < 3; i++)
            cout <<  rx++ << "... ";
    cout << "\n";
}
```

Here is the output:

```
10... 11... 12...
vals[4] = 13 after using byref()
600... 601... 602...
thing.a = 603 after using byref()
101... 102... 103...
43... 44... 45...
x = 20 after using byref()
15... 16... 17...
ing = 15 after using byref()
```

Startling Results and Anonymous Variables

The first thing to note is that the reference *rx* can refer to any data object having the properties of an `int` variable. Thus `vals[4]`, being an element of an array of `int`, is equivalent to an `int` variable. You can display it, add it, assign to it, and so on, just as with a regular variable. Therefore C++ lets you make *rx* be an alias for an individual array element. Similarly, while *thing* is a structure, *thing.a* is an `int`, so you can make *rx* an alias for that structure member. You'll notice that `byref()` uses the values of *vals[4]* and of *thing.a* in its display and that increasing *rx* by 3 in `byref()` increases *vals[4]* and *thing.a* by 3. That's because *rx* is *vals[4]* after the first function call and *rx* is *thing.a* after the second function call. A rose by any other name smells as sweet, and a structure member or an array element by any other reference is the same `int` it always was.

Although perhaps surprising, this behavior makes sense once you realize that array elements and structure members can act like simple variables. The next two examples of argument forms are a bit more startling. First, the program passes an argument of *100*. While *100* may be an `int`, it's hardly a variable. For instance, replacing *rx* with *100* produces *100++*, which is a syntax error. Yet the output clearly shows that *rx* starts out with the value 100 and that it allows itself to be incremented!

What's happening may become clearer with the subsequent example, in which we pass not a variable, not a constant, but an entire expression: *2 * x + 3*. Again, this is not a variable. Again, replacing *rx* with the expression yields *(2 * x + 3)++*, a syntax error. Yet the program correctly prints the values *43*, *44*, and *45*. The clue to what's really happening comes in the next output line, in which the program reveals that the value of *x* is still unchanged. Changing *rx* had no effect on *x*, so whatever *rx* refers to, it's not *x*. What has happened is that the program has created an *anonymous* and temporary type *int* variable, assigned it the value of the expression (or of *100* in the preceding example), and then made *rx* a reference to that anonymous variable.



If a function call argument does not match the type of a corresponding reference parameter, C++ creates an anonymous variable of the correct type, assigns the value of the function call argument to the anonymous variable, and has the parameter refer to that variable.

This rule enables functions with reference arguments to handle function calls that pass constants and expressions instead of the expected variable. It also explains what happened in the final example, in which we attempt to pass *ing*, a type `long` variable, to `byref()`. Because of the type mismatch, the program created an anonymous type `int` variable, assigned it the value *15*, and had *rx* refer to it. Thus the program printed the values *15*, *16*, and *17*, but left *ing* unchanged. In short, *rx* altered the anonymous variable, not *ing*.

Because C++ creates anonymous variables when you mismatch a variable to a reference, references don't work that well with C++'s basic types. The anonymous variable doesn't cause problems if you don't expect the function to modify the original variable, but the usual reason for using a reference is so that the function *will* modify the original variable. For this reason, many programmers prefer to use pointers instead of references when a function modifies a basic type in the calling program. The syntax is a little uglier, what with using the & and *operators, but even that is an advantage. When you see a function call like `swapp(&wallet1, &wallet2)`, you know without looking for the prototype that this function can alter variables in the calling program.

USING REFERENCES WITH A STRUCTURE

References may be less than ideal for C++'s basic types, but they work wonderfully with structures and classes, C++'s user-defined types. You're unlikely to try to pass the wrong type of structure, and if you do, the compiler will catch it.

The method for using a reference to a structure is the same as the method for using a reference to a basic variable; just use the & reference operator when declaring a structure parameter. The program in Listing 8-8 does exactly that. It also adds an interesting twist by having a function return a reference. This makes it possible to use a function invocation as an argument to a function. Well, that's true of any function with a return value. But it also makes it possible to assign to a function invocation, and that's possible only with a reference return type. We'll explain these points after showing the program's output. The program has a `use()` function that displays two members of a structure and increments a third member. Thus the third member can keep track of how many times a particular structure has been handled by the `use()` function.

LISTING 8-8 strtref.cpp

```
// strtref.cpp -- using structure references
#include <iostream.h>
struct thug
{
    char name[20];
    char quote[64];
    int used;
};

thug & use(thug & thugref);     // function with a reference return type

int main(void)
{
// NOTE: some implementations require using the keyword static
// in the two structure declarations to enable initialization
    thug muggins =
    {
            "Potato Nose Muggins",
            "Youse guys ain't gettin' nuttin' outa me!",
            0
    };
    use(muggins);                   // muggins is type thug
    cout << muggins.used << " use(s)\n";

    use (use(muggins));             // use(muggins) is type thug
    cout << muggins.used << " use(s)\n";

    thug morf =
    {
            "Polly Morf",
            "Would you like to pet my banana slug?",
            0
    };
    use(muggins) = morf; // can assign to function!
    cout << muggins.name << " says ciao\n";
    return 0;
}

// use() returns the reference passed to it
thug & use(thug & thugref)
{
    cout << thugref.name << " says:\n";
    cout << thugref.quote << "\n";
    thugref.used++;
    return thugref;
}
```

Here's the output:

```
Potato Nose Muggins says:
Youse guys ain't gettin' nuttin' outa me!
1 use(s)
Potato Nose Muggins says:
Youse guys ain't gettin' nuttin' outa me!
Potato Nose Muggins says:
Youse guys ain't gettin' nuttin' outa me!
3 use(s)
Potato Nose Muggins says:
Youse guys ain't gettin' nuttin' outa me!
Polly Morf says ciao
```

Program Notes

The program ventures into three new areas. The first is using a reference to a structure, illustrated by the first function call:

```
use(muggins);
```

It passes the structure *muggins* by reference to the `use()` function, making *thugref* a synonym for *muggins*. So when the `use()` function displays the *name* and *quote* members of *thugref* it really displays the members of *muggins*. Also, when the function increments *thugref.used* to *1*, it really increments *muggins.used*, as the program output shows:

```
Potato Nose Muggins says:
Youse guys ain't gettin' nuttin' outa me!
1 use(s)
```

The second new area is using a reference as a return value. Because `use()` returns a type *thug* reference, it can be used as an argument to any function expecting type *thug* reference arguments, such as `use()` itself. Thus the next function call in Listing 8-8 is really two function calls, with one function serving as the argument for the second:

```
use(use(muggins));
```

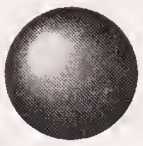
The inner function call prints the *name* and *quote* members and increments the *used* member to 2. The function returns *thugref*, reducing what's left to the following:

```
use(thugref) :
```

Because *thugref* is a reference to *muggins*, this function call is equivalent to the following:

```
use(muggins) :
```

So `use()` displays the two string members again, and it increments the *used* member to *3*.



A function that returns a reference is actually an alias for the referred-to variable.

This property of functions returning references turns out to be crucial to aspects of C++ class design. For instance, recall that the *ostream* class defines the << operator so that you can concatenate expressions as follows:

```
cout << muggins.used <<  " use(s)\n";
```

The C++ mechanism for defining (overloading) operators is to write a function describing how the operator works. Concatenating the operator, as above, turns out to be equivalent to having nested function calls, such as `use(use(muggins))`, in which the argument and return values are references to the same type. We'll return to this topic in Chapter 10.

The third new area the program explores is that you can assign to a function if that function has a reference type return value:

```
use(muggins) = morf;
```

For nonreference return values, this assignment would be a syntax error, but it's okay for `use()`. This is the order of events. First, the `use()` function is evaluated. That means *muggins* is passed by reference to `use()`. As usual, the function displays two members and increments the *used* member to *4*. Then the function returns the reference. Because the return value refers to *muggins*, this makes the final step equivalent to the following:

```
muggins = morf;
```

C++ allows you to assign one structure to another, so this copies the contents of the *morf* structure into *muggins*, as is shown when displaying *muggins.name* produces Morf's name and not Muggins's. In short, the statement

```
use(muggins) = morf;
```

is equivalent to the following:

```
use(muggins);
muggins = morf;
```



You can assign a value (including a structure or a class object) to a C++ function only if the function returns a reference to a variable or, more generally, to a data object. In that case, the value is assigned to the referred-to variable or data object.

This is another property that enables certain forms of operator redefinition. Chapter 11 will use it, for instance, to redefine the array subscript operator *[]* for a class that defines a more powerful version of the array.

Separate Compilation

C++, like C, allows you, even encourages you, to locate the component functions to a program in separate files. As Chapter 1 described, you can then compile the files separately, then link them into the final executable program. (A C++ compiler typically compiles programs and also manages the linker program.) If you modify just one file, you can recompile just that one file, then link it to the previously compiled versions of the other files. This facility makes it easier to manage large programs. Furthermore, most C++ environments provide additional facilities to help with the management. UNIX systems, for example, have the *make* program. It keeps track of which files a program depends upon and of when they were last modified. If you run *make* and it detects you've changed one or more source files since the last compilation, *make* remembers the proper steps needed to reconstitute the program. Zortech C++ also has a *make* feature, and the Turbo C++ environment provides a similar facility with its Project menu.

Let's look at a simple example. Instead of looking at compilation details, which will depend on the implementation, let's concentrate on more general aspects, such as design.

Suppose, for example, you decide to break up the program in Listing 7-11 by placing the functions in a separate file. That listing, recall, converted rectangular coordinates to polar coordinates, then displayed the result. You can't simply cut the original file on a dotted line after the end of *main()*. The problem is that *main()* and the other two functions all use the same structure templates, so you need to put the templates in both files. Simply typing them in is an invitation to err. Even if you copy the templates correctly, you have to remember to modify both sets of templates if you make changes later. In short, spreading a program over multiple files creates new problems.

Who wants more problems? The developers of C and C++ didn't, so they've provided the `#include` facility to deal with this sort of situation. Instead of placing the templates in each file, you can place them in a header file, then include that header file in each source code file. That way, if you modify the structure definition, you can do so just once, in the header file. Also, you can place the function prototypes in the header file. Thus you can divide the original program into three parts:

- A header file that contains the structure templates and prototypes for functions using those templates.

- A source code file that contains the code for the structure-related functions.
- A source code file that contains the code that calls upon those functions.

This is a useful strategy for organizing a program. If, for instance, you write another program that uses those same functions, just include the header file and add the function file to the project or *make* list. Also, this organization reflects the OOP approach. One file, the header file, contains the definition of the user-defined types. A second file contains the function code for manipulating the user-defined types. Together, they form a package you can use for a variety of programs.

Listings 8-9, 8-10, and 8-11 show the result of dividing Listing 7-11 into separate parts. Note that we use "`coordin.h`" instead of `<coordin.h>` when including the header file. If the file name is enclosed in brackets, the C++ compiler looks at the part of the host system's file system that holds the standard header files. But if the file name is enclosed in double quotation marks, the compiler first looks at the current working directory (or equivalent). If it doesn't find the header file there, it then looks in the standard location.

Figure 8-3 outlines the steps for putting this program together on a UNIX system. Note that you just give the *CC* compile command and the other steps follow automatically. Zortech C++ and Turbo C++ go through essentially the same steps, but, as outlined in Chapter 1, you initiate the process differently.

Let's quickly review the Turbo C++ method. (Borland C++ follows the same steps.) First, you open the Project menu and select Open project. In the dialog box, choose a name for the project file. To help keep track of things, a good choice is the name you've used for the file holding the `main()` function. If you enter, say `file1`, Turbo C++ automatically adds a `prj` extension indicating that `file1.prj` is a project file. Then choose Add from the Project menu and select the first file for the project; in this case, that would be `file1.cpp`. Then use Add to select the next file, `file2.cpp` in this case. And that's it. When you're finished with this program, use the Close command from the Project menu to close the project. (As long as you have a particular project open, the compiler will run that program regardless of what source code files are open.) To work on the program again, use the Open project command from the Project Menu to reopen the project.

LISTING 8-9 coordin.h

```
// coordin.h -- structure templates and function prototypes
// structure templates
struct polar
{
    double distance;     // distance from origin
    double angle;        // direction from origin
```

(Listing continued on p. 319)

1. Give UNIX compile command for two source files:

```
CC file1.cpp file2.cpp
```

2. Preprocessor combines included files with source code:

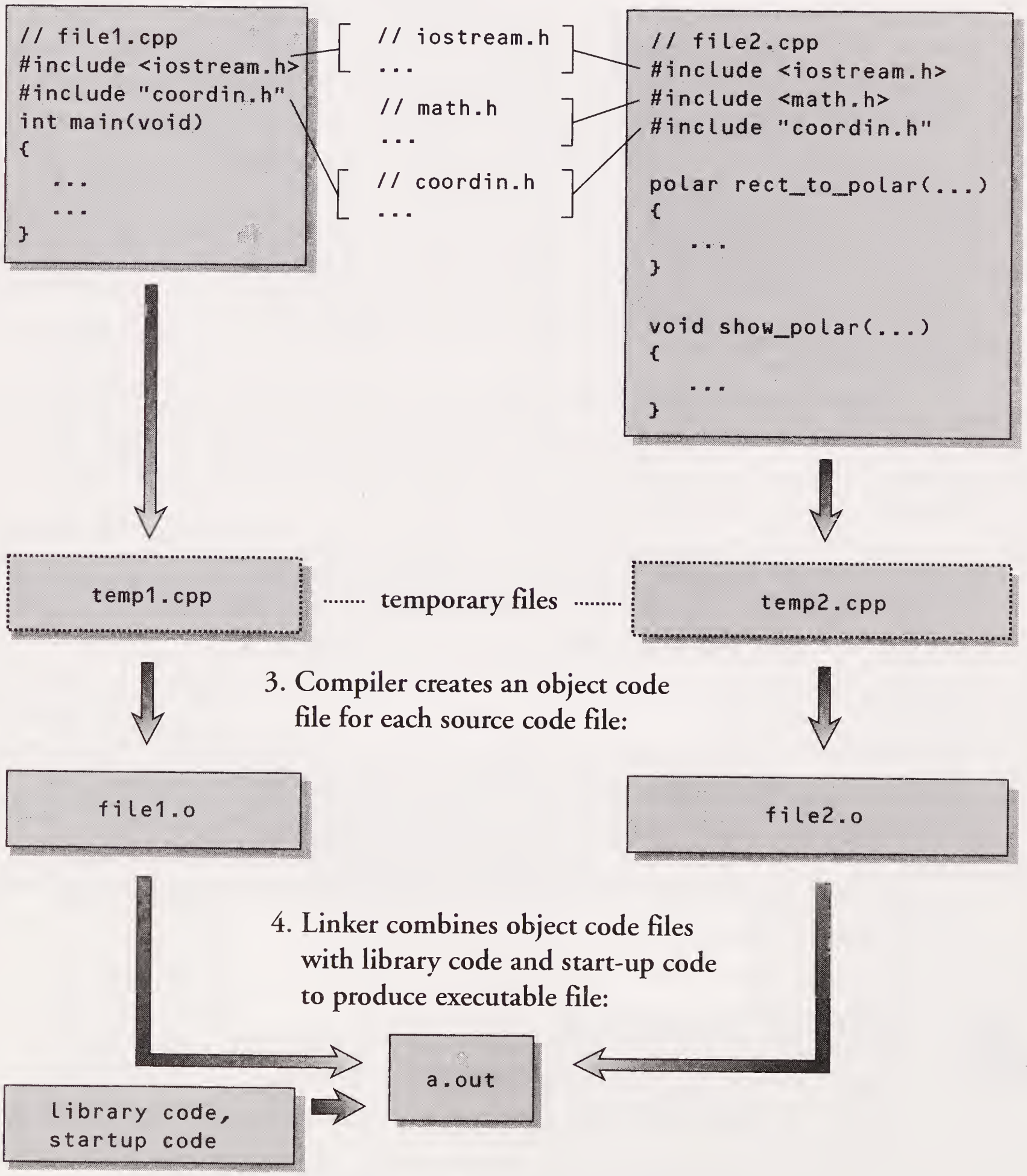


Figure 8-3 *Compiling a multifile C++ program on a UNIX system*

```
};
struct rect
{
    double x;              // horizontal distance from origin
    double y;              // vertical distance from origin
};

// prototypes
polar rect_to_polar(rect xypos);
void show_polar(polar dapos);
```

LISTING 8-10 file1.cpp

```
// file1.cpp -- example of a two-file program
#include <iostream.h>
#include "coordin.h" // structure templates, function prototypes
int main(void)
{
    rect rplace;
    polar pplace;

    cout << "Enter the x and y values: ";
    while (cin >> rplace.x >> rplace.y)  // slick use of cin
    {
            pplace = rect_to_polar(rplace);
            show_polar(pplace);
            cout << "Next two numbers (q to quit): ";
    }
    return 0;
}
```

LISTING 8-11 file2.cpp

```
// file2.cpp -- contains functions called in file1.cpp
#include <iostream.h>
#include <math.h>
#include "coordin.h" // structure templates, function prototypes

// convert rectangular to polar coordinates
polar rect_to_polar(rect xypos)
{
    polar answer;

    answer.distance =
            sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
```

```
    answer.angle = atan2(xypos.y, xypos.x);
    return answer;     // returns a polar structure
}

// show polar coordinates, converting angle to degrees
void show_polar (polar dapos)
{
    const double Rad_to_deg = 57.29577951;

    cout << "distance = " << dapos.distance;
    cout << ", angle = " << dapos.angle * Rad_to_deg;
    cout << " degrees\n";
}
```

Storage Classes, Scope, and Linkage

Now that you've seen a multifile program, it's a good time to extend the discussion of storage classes in Chapter 4, for storage classes affect how information can be shared across files. It may be a while since you last read Chapter 4, so let's review what it said about storage classes. C++ uses three separate schemes for storing data, and the schemes differ in how long they preserved data in memory.

- Automatic variables are those declared inside a function definition; that includes function parameters. They are created when program execution enters the function or block in which they are defined, and the memory used for them is freed when execution leaves the function or block.
- Static variables are those defined outside of a function definition or else using the keyword `static`. They persist for the entire time a program is running.
- Dynamic storage allocated by the `new` operator persists until freed with the `delete` operator or until the program ends, whichever comes first.

We'll fill you in on the rest of the story now, including fascinating details about when variables of different types are *in scope*, or *visible* (usable by the program), and about *linkage*, which determines what information is shared across files.

Scope and Linkage

Scope describes how widely visible a name is in a program. A C++ variable can have one of two scopes. A variable having *block scope* is known only within the block in which it is defined. A block, you remember, is a series of statements enclosed in braces. A function body, for example, is a block, but you can have other blocks nested within the

function body. A variable having *file scope* is known throughout the file after the point it is defined. Automatic variables have block scope, and a static variable can have either scope, depending on how it is defined. C++ functions (other than class member functions, which we'll discuss in later chapters) have file scope.

Linkage determines which names refer to variables or functions in one file only and which can be used in several files to refer to the same variable or function. A variable or function with *internal linkage* is known only to the file containing its definition. A variable or function with *external linkage* can be shared among different files.

Let's look in more detail at the scope and linkage properties of C++'s various storage classes.

AUTOMATIC VARIABLES

Function parameters and variables declared inside a function belong, by default, to the automatic storage class. These variables have *local* visibility, or scope. That is, if you declare a variable called *texas* in `main()` and declare another variable with the same name in a function called `oil()`, you've created two independent variables, each known only in the function in which it's defined. Anything you do to the *texas* in `oil()` has no effect on the *texas* in `main()`, and vice versa. Also, each variable is allocated when its function begins execution, and each fades from existence when its function terminates.

If you define a variable inside of a block, the variable's persistence and scope is confined to that block. Suppose, for instance, you define a variable called *teledeli* at the beginning of `main()`. Now suppose you start a new block within `main()` and define a new variable, also called *teledeli*, in that block. The program interprets the *teledeli* name to mean the local block variable while the program executes statements within the block. We say the new definition *hides* the prior definition. The new definition is *in scope*, and the old definition temporarily is *out of scope*. When the program leaves the block, the original definition comes back into scope. See Figure 8-4.

Listing 8-12 illustrates how automatic variables are localized to the function or block containing them.

LISTING 8-12 auto.cpp

```
// auto.cpp -- illustrating scope of automatic variables
#include <iostream.h>
void oil(int x);
int main(void)
{
// NOTE: some implementations require that you type cast the
// addresses in this program to type unsigned
```

```
    int texas = 31;
    int year = 1992;
    cout << "In main(), texas = " << texas << ", &texas =";
    cout << &texas << "\n";
    cout << "In main(), year = " << year << ", &year =";
    cout << &year << "\n";
    oil(texas);
    cout << "In main(), texas = " << texas << ", &texas =";
    cout << &texas << "\n";
    cout << "In main(), year = " << year << ", &year =";
    cout << &year << "\n";
    return 0;
}

void oil(int x)
{
    int texas = 5;

    cout << "In oil(), texas = " << texas << ", &texas =";
    cout << &texas << "\n";
    cout << "In oil(), x = " << x << ", &x =";
    cout << &x << "\n";
    {                              // start a block
        int texas = 113;
        cout << "In block, texas = " << texas;
        cout << ", &texas = " << &texas << "\n";
        cout << "In block, x = " << x << ", &x =";
        cout << &x << "\n";
    }                              // end a block
    cout << "Post-block texas = " << texas;
    cout << ", &texas = " << &texas << "\n";
}
```

Here is the output:

```
In main(), texas = 31, &texas =0x8f95fff4
In main(), year = 1992, &year =0x8f95fff2
In oil(), texas = 5, &texas =0x8f95ffea
In oil(), x = 31, &x =0x8f95fff0
In block, texas = 113, &texas = 0x8f95ffe8
In block, x = 31, &x =0x8f95fff0
Post-block texas = 5, &texas = 0x8f95ffea
In main(), texas = 31, &texas =0x8f95fff4
In main(), year = 1992, &year =0x8f95fff2
```

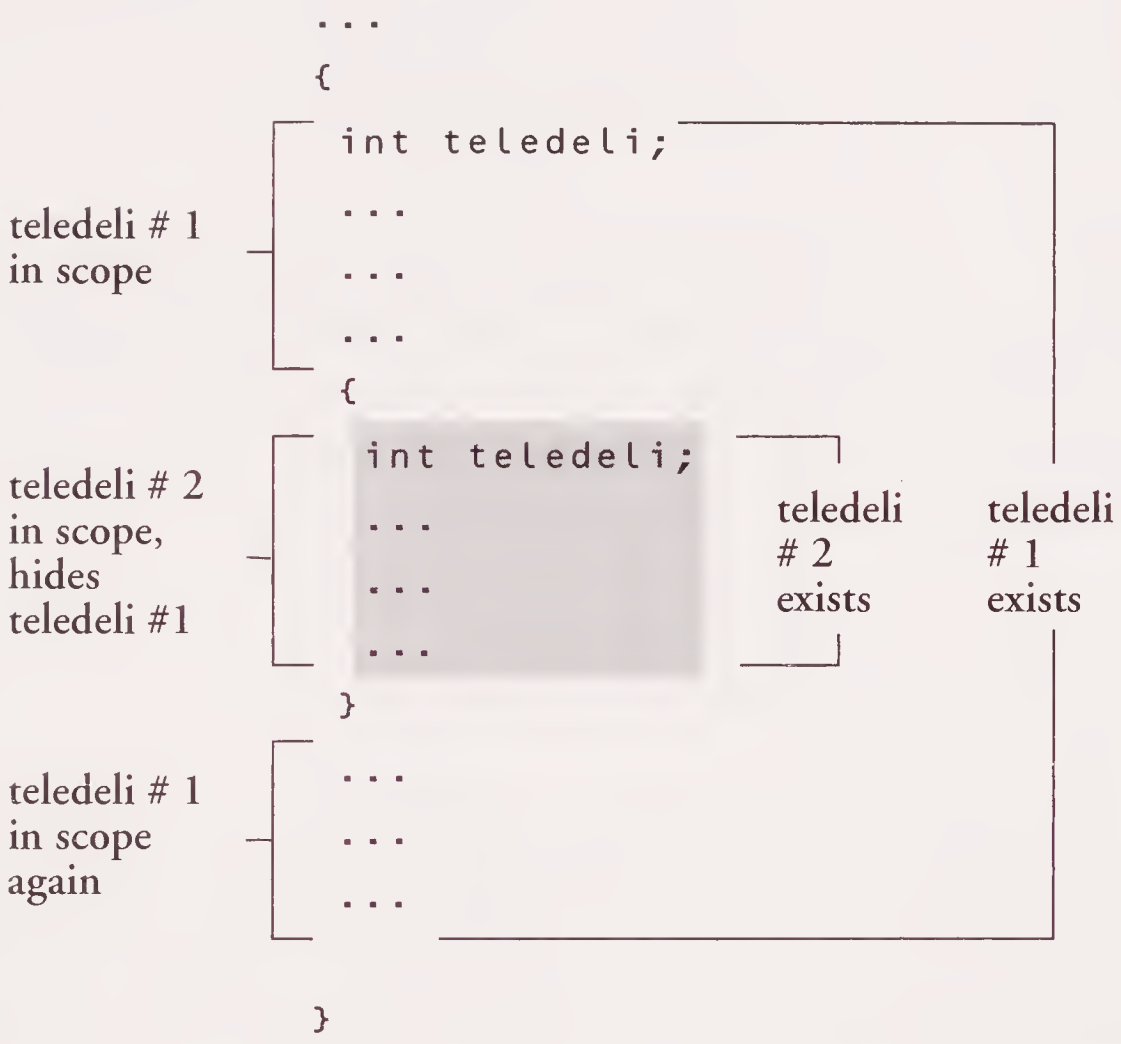


Figure 8-4 *Blocks and scope*

Notice how each of the three *texas* variables has its own distinct address and how the program uses only the particular variable in scope at the moment, so assigning the value *113* to the *texas* in the inner block in `oil()` has no effect on the other variables of the same name.

Let's summarize the sequence of events. When `main()` starts, the program allocates space for *texas* and *year*, and these variables come into scope. When the program calls `oil()`, these variables remain in memory but pass out of scope. Two new variables, *x* and *texas*, are allocated and come into scope. When program execution reaches the inner block in `oil()`, the new *texas* passes out of scope as it is superseded by an even newer definition. The variable *x*, however, stays in scope because the block doesn't define a new *x*. When execution exits the block, the memory for the newest *texas* is freed, and *texas* number 2 comes back into scope. When the `oil()` function terminates, that *texas* and *x* expire, and the original *texas* and *year* come back into scope.

Incidentally, you can use the C++ (and C) keyword *auto* to indicate the storage class explicitly:

```
int froob(int n)
{
    auto float ford;
     ...
}
```

Because one can use the `auto` keyword only with variables that already are automatic by default, programmers rarely bother using the `auto` keyword. Occasionally, it's used to clarify code to the reader. For example, you can use it to indicate that you purposely are creating an automatic variable that overrides a global definition, such as those we'll be discussing shortly.

Automatic Variables and the Stack

You may gain a better understanding of automatic variables by seeing how a typical C++ compiler implements them. Because the number of automatic variables grows and shrinks as functions start and terminate, the program has to manage automatic variables as it runs. The usual means is to set aside a section of memory and treat it as a *stack* for managing the flow and ebb of variables. It's called a stack because new data figuratively is stacked atop old data, then removed from the stack when a program is finished with it. The default size of the stack depends on the implementation, but a compiler generally provides the option of changing the size. The program keeps track of the stack by using two pointers. One points to the base of the stack, where the memory set aside for the stack begins, and one points to the top of the stack, which is the next free memory location. When a function is called, its automatic variables are added to the stack, and the pointer to the top points to the next available free space following the variables. When the function terminates, the top pointer is reset to the value it had before the function was called, effectively freeing the memory that had been used for the new variables.

A stack is a LIFO (Last In, First Out) design, meaning the last variables added to the stack are the first to go. (To date, variables haven't mounted a legal challenge to this practice.) The design simplifies argument passing. The function call places the values of its arguments on top of the stack and resets the top pointer. The called function uses the description of its formal arguments to determine the addresses of each argument. Figure 8-5, for example, shows a `fib()` function that, when called, passes a two-byte *int* and a four-byte `long`. These values go on the stack. When `fib()` begins execution, it associates the names *real* and *tell* with the two values. When `fib()` terminates, the top-of-stack pointer is relocated to its former position. The new values aren't erased, but they no longer are labeled, and the space they occupy will be used by the next process that

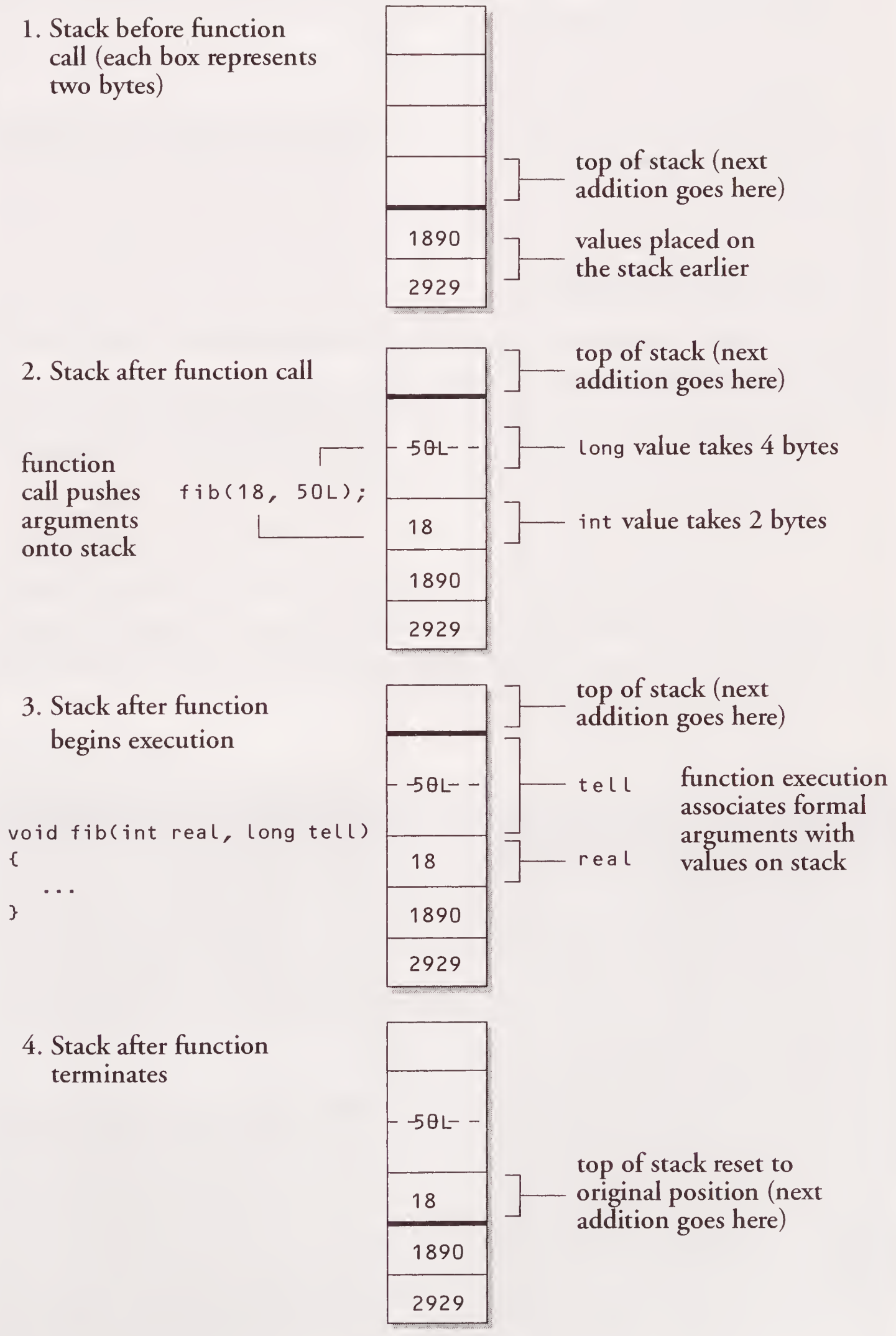


Figure 8-5 *Passing arguments with a stack*

places values on the stack. (The figure is somewhat simplified, for function calls may pass additional information, such as a return address.)

You may have noticed that addresses in Listing 8-12 decreased rather than increased as new variables were added. That's because this particular C++ compiler implements the stack upside down. This changes the direction the stack grows, but it retains the basic concept.

Type register Variables

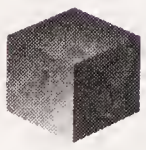
C++, like C, supports the *register* keyword for declaring automatic variables. This keyword is a hint to the compiler that you would like it to use a CPU register instead of the stack to handle a particular variable. The idea is that the CPU can access a value in one of its registers more rapidly than it can access memory in the stack. To declare a *register* variable, preface the type with the keyword:

```
register int count_fast;  // request for a register variable
```

You've probably noticed the qualifying words "hint" and "request." The compiler doesn't have to honor the request. For example, the registers may already be occupied, or you might request a type that doesn't fit in a register. Current feeling is that modern compilers are often smart enough not to need the hint. If you write a `for` loop, for example, the compiler may take it upon itself to use a register for the loop index.

THE STATIC STORAGE CLASS

Variables that belong to the static storage class last for the duration of the program; they are less ephemeral than automatic variables. Because the number of static variables doesn't change as the program runs, the program doesn't need a special device like a stack to manage them. Instead, the compiler allocates a fixed block of memory to hold all the static variables, and those variables stay present as long as the program executes. Also, if you don't explicitly initialize a static variable, the compiler sets it to zero. Static arrays and structures have each element or member set to zero by default.



COMPATIBILITY NOTE

Classic K&R C did not allow you to initialize automatic arrays and structures, but it did allow you to initialize static arrays and structures. ANSI C and C++ 2.0 allow you to initialize both kinds. But some C++ 2.0 translators use C compilers that are not fully ANSI C compliant. If you are using

such an implementation, you may need to use one of the three varieties of static storage classes for initializing arrays and structures.

C++, like C, provides three varieties of static variables: external, static, and external static. If that looks a little confusing to you, you're right. Unfortunately, C++ uses the word static in two different senses. One is that of a variable that persists for the duration of the program. In that sense, all three varieties are static. The second sense limits how widely known the variable is; it affects the scope and linkage. The external variable is available to all files of a program (file scope, external linkage), the external static variable is available to all the functions in a single file (file scope, internal linkage), and the static variable is confined to a single function (block scope, internal linkage). Table 8-1 summarizes the storage class features; you've seen some of this information already, but we'll take a closer look now.

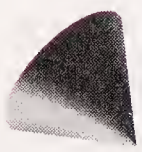
External Variables

External variables are called external because they are defined outside of, hence external to, any function. For example, they could be declared above the `main()` function. You

Table 8-1 Storage classes

automatic	default for function parameters and variables declared inside a function	block	internal	while defining block is being executed
external	default for variables declared outside of any function	file	external	while program is running
static	by applying the `static` keyword to a variable declared inside a function	block	internal	while program is running
extern static	by applying the `static` keyword to a variable declared outside any function	file	internal	while program is running

can use an external variable in any function that follows the external variable's definition in the file. Thus external variables are often termed *global* variables in contrast to automatic variables, which are *local* variables. However, if you define an automatic variable having the same name as an external variable, the automatic variable is the one in scope when the program executes that particular function. Listing 8-13 illustrates these points. It also shows how you can use the keyword `extern` to redeclare an external variable defined earlier and how you can use C++'s scope resolution operator to access an otherwise hidden external variable.



LISTING 8-13 external.cpp

```
// external.cpp -- external variables
#include <iostream.h>
// external variable
double warming = 0.3;

// function prototypes
void update(double dt);
void local(void);

int main(void)                    // uses global variable
{
    cout << "Global warming is " << warming << " degrees.\n";
    update(0.1);                  // call function to change warming
    cout << "Global warming is " << warming << " degrees.\n";
    local();                      // call function with local warming
    cout << "Global warming is " << warming << " degrees.\n";
    return 0;
}

void update(double dt)            // modifies global variable
{
    extern double warming;        // optional redeclaration
    warming += dt;
    cout << "Updating global warming to " << warming;
    cout << " degrees.\n";
}

void local(void)                  // uses local variable
{
    double warming = 0.8;         // new variable hides external one

    cout << "Local warming = " << warming << " degrees.\n";
            // Access global variable with the
```

```
            // scope resolution operator
    cout << "But global warming = " << ::warming;
    cout << " degrees.\n";
}
```

Here is the output:

```
Global warming is 0.3 degrees.
Updating global warming to 0.4 degrees.
Global warming is 0.4 degrees.
Local warming = 0.8 degrees.
But global warming = 0.4 degrees.
Global warming is 0.4 degrees.
```

Program Notes

The program output illustrates that both `main()` and `update()` can access the external variable *warming*. Note that the change that *update()* makes to *warming* shows up in subsequent uses of the variable.

The `update()` function redeclares the *warming* variable using the keyword `extern`. This keyword means "use the variable by this name previously defined externally." Since that is what `update()` would do anyway if you omitted the entire declaration, this declaration is optional. It serves to document that the function is designed to use the external variable. The original declaration

```
double warming = 0.3;
```

is called a *defining declaration*, or, simply, a *definition*. It causes storage for the variable to be allocated. The redeclaration

```
extern double warming;
```

is called a *referencing declaration*, or, simply, a *declaration*. It does not cause storage to be allocated, for it refers to an existing variable. You can use the `extern` keyword *only* in declarations referring to previously defined variables (or functions—more on that later). Also, you cannot initialize a variable in a referencing declaration:

```
extern double warming = 0.5;   // INVALID
```

You can initialize a variable in a declaration only if the declaration allocates the variable, that is, only in a defining declaration. After all, the term initialization means assigning a value to a memory location when that location is allocated.

The `local()` function demonstrates that when you define a local variable having the same name as a global variable, the local version hides the global version. The

`local()` function, for example, uses the local definition of *warming* when displaying warming's value.

C++ takes a step beyond C by offering the scope resolution operator (`::`). When prefixed to the name of a variable, this operator means to use the global version of that variable. Thus `local()` displays *warming* as *0.8*, but it displays `::warming` as *0.4*. You'll encounter this operator again in a different context when we discuss classes.



GLOBAL OR LOCAL?

Now that you have a choice of using global or local variables, which should you use? At first, global variables have a seductive appeal—because all functions have access to a global variable, you don't have to bother passing arguments. But this easy access has a heavy price—unreliable programs. Computing experience has shown that the better job your program does of isolating data from unnecessary access, the better job the program does in preserving the integrity of the data. So most often, you should use local variables and pass data to functions on a need-to-know basis rather than making data available indiscriminately with global variables. As you will see, OOP takes this data isolation a step further.

Global variables do have their uses, however. For example, you may have a block of data that's to be used by several functions, such as an array of month names or of the atomic weights of the elements. The external storage class is particularly suited to representing constant data, for then you can use the keyword *const* to protect the data from change.

```
const char * const months[12] =
{
    "January", "February", "March", "April", "May",
    "June", "July", "August", "September", "October",
    "November", "December"
};
```

The first `const` protects the strings from change, and the second `const` makes sure that each pointer in the array remains pointing to the same string it pointed to initially.

THE STATIC MODIFIER (LOCAL VARIABLES)

The `static` modifier can be used either with a local variable or with a global variable. We'll look at the local case now. When used inside a function, `static` makes a local variable have the static storage class. That means that even though the variable is known

only to that function, it exists even while the function is inactive. Thus a static local variable can preserve its value between function calls. (Static variables would be useful for reincarnation—you could use them to pass bank deposit information to your next appearance.) Also, if you initialize a static local variable, the program initializes the variable once, when the program starts up. Subsequent calls to the function don't reinitialize the variable, the way they do for automatic variables. Listing 8-14 illustrates these points.

LISTING 8-14 static.cpp

```
// static.cpp -- using a static local variable
#include <iostream.h>
// constants
const int Size = 80;

// function prototype
void strcount(char *str);

int main(void)
{
    char input[Size];
    cout << "Enter a line:\n";
// NOTE: if your implementation doesn't support getline(), you'll
// have to rewrite the program using get(char *, int) and
// get(char)

    cin.getline(input, Size);
    while (input[0] != '\0')
    {
            strcount(input);
            cout << "Enter next line (empty line to quit):\n";
            cin.getline(input, Size);
    }
    cout << "Bye\n";
    return 0;
}

void strcount(char * str)
{
    static int total = 0;          // static local variable
    int count = 0;                 // automatic local variable

    while (*str++)                 // go to end of string
            count++;
```

```
    total += count;
    cout << count << " characters in this string\n";
    cout << total << " characters total\n";
}
```

Here is the program's output:

```
Enter a line:
Say, haven't we met somewhere before?
37 characters in this string
37 characters total
Enter next line (empty line to quit):
There's really something quite special about you!
49 characters in this string
86 characters total
Enter next line (empty line to quit):

Bye
```

Note that the automatic variable *count* is reset to *0* each time the function is called. However, the static variable *total* is set to *0* once, at the beginning. After that, *total* retains its value between function calls, so it's able to maintain a running total.

LINKAGE, EXTERNAL VARIABLES, AND THE STATIC AND EXTERNAL MODIFIERS

Applying the `static` modifier to an external variable becomes meaningful in multifile programs. In that context, a `static` external variable is local to the file containing it. It has *internal linkage*. But a regular external variable has *external linkage*, meaning it can be used in different files. In that case, one and only one file can contain the external definition for the variable. Other files wishing to use that variable must use the keyword `extern` in a reference declaration. See Figure 8-6.

If a file doesn't provide the *extern* declaration of a variable, it can't use an external variable defined elsewhere:

```
// file1
int errors = 20;          // external declaration
...
-----------------------------------------
// file 2
...                       // missing an extern int errors declaration
void froobish()
{
    cout << errors;       // doomed attempt to use errors
..
```

If a file attempts to define a second external variable by the same name, that's an error:

```
// file1
int errors = 20;          // external declaration
...
-------------------------------------------
// file 2
int errors;               // invalid declaration
void froobish()
{
    cout << errors;       // doomed attempt to use errors
..
```

The correct approach is to use the keyword `extern` in the second file:

```
// file1
int errors = 20;          // external declaration
...
-------------------------------------------
// file 2
extern int errors;        // refers to errors from file1
void froobish()
{
    cout << errors;       // uses errors defined in file1
..
```

But if a file declares a `static` external variable having the same name as an ordinary external variable declared in another file, the `static` version is the one in scope for that file:

```
// file1
int errors = 20;          // external declaration
...
-------------------------------------------
// file2
static int errors = 5;    // known to file2 only
void froobish()
{
    cout << errors;       // uses errors defined in file2
..
```



In a multiple-file program, you can define an external variable in one and only one file. All other files using that variable have to declare that variable with the `extern` keyword.

Use a regular external variable to share data among different parts of a multifile program. Use a `static` external variable to share data among functions found in just one file. Also, if you make an external variable static, you needn't worry about its name conflicting with external variables found in other files.

Listings 8-15 and 8-16 show how C++ handles external and static external variables. Listing 8-15 (`twofile1.cpp`) defines the external variables *tom* and *dick* and the static external variable *harry*. The `main()` function in that file displays the addresses of the three variables, then calls the `remote_access()` function, which is defined in a second file. Listing 8-16 (`twofile2.cpp`) shows that file. In addition to defining `remote_access()`, the file uses the `extern` keyword to share *tom* with the first file. Next, the file defines a static variable called *dick*. The `static` modifier makes this variable local to the file and overrides the global definition. Then the file defines an external variable called *harry*. It can do so without conflicting with the *harry* of the first file because the first *harry* is local to `twofile1.cpp`. Then the `remote_access()` function displays the addresses of these three variables so that you can compare them

```
// file1.cpp
#include <iostream.h>

// function prototypes
#include "mystuff.h"

// defining an external variable
int process_status = 0;

int main(void)
{
    ...
}

void promise(void)
{
    ...
}
```

This file defines the variable *process_status*, causing the compiler to allocate space for it.

```
// file2.cpp
#include <iostream.h>

// function prototypes
#include "mystuff.h"

// referencing an external variable
extern int process_status;

int manipulate(int n)
{
    ...
}

char * remark(char * str)
{
    ...
}
```

This file uses `extern` to instruct the program to use the variable *process_status* that was defined in another file.

Figure 8-6 *Defining declaration and referencing declaration*

with the addresses of the corresponding variables in the first file. Remember to compile both files together to get the complete program.

LISTING 8-15 twofile1.cpp

```
// twofile1.cpp -- external and static external variables
#include <iostream.h>    // to be compiled with two file2.cpp
int tom = 3;             // external variable definition
int dick = 30;           // external variable definition
static int harry = 300;  // static external variable definition

// function prototype
void remote_access(void);

int main(void)
{
// NOTE: some implementations require that you type cast
// the addresses to type unsigned

    cout << "main() reports the following addresses:\n";
    cout << &tom << " = &tom, " << &dick << " = &dick, ";
    cout << &harry << " = &harry\n";
    remote_access();
    return 0;
}
```

LISTING 8-16 twofile2.cpp

```
// twofile2.cpp -- external and static external variables
#include <iostream.h>
extern int tom;          // tom defined elsewhere
static int dick = 10;    // overrides external dick
int harry = 200;         // external variable definition,
                         // no conflict with twofile1 harry

void remote_access(void)
{
// NOTE: some implementations require that you type cast
// the addresses to type unsigned

    cout << "remote_access() reports the following addresses:\n";
    cout << &tom << " = &tom, " << &dick << " = &dick, ";
    cout << &harry << " = &harry\n";
}
```

Here is the output:

```
main() reports the following addresses:
0x8fa000a8 = &tom, 0x8fa000aa = &dick, 0x8fa000ac = &harry
remote_access() reports the following addresses:
0x8fa000a8 = &tom, 0x8fa000f8 = &dick, 0x8fa000fa = &harry
```

As you can see, both files use the same *tom* variable, but different *dick* and *harry* variables.

STORAGE CLASSES AND CONST

In C++ (but not C), the `const` modifier alters the default storage classes slightly. While an external variable has *external* linkage by default, a `const` external variable has *internal* linkage by default. That is, C++ treats an external *const* definition like a `static` external definition. You can think of the keyword `const`, when used in an external definition, as being preceded by an implicit `static` keyword.

```
const int fingers = 10; // same as static const int fingers;
int main(void)
{
...
```

C++ bent the rules for constant types to make life easier for you. Suppose, for instance, you have a set of constants that you'd like to place in a header file and that you use this header file in several files in the same program. After the preprocessor includes the header file contents in each source file, each source file will contain definitions like this:

```
const int fingers = 10;
const char * warning = "Wak!";
```

If external `const` declarations had external linkage like regular variables , this would be an error, for you can define an external variable in one file only. That is, only one file could contain the above declaration, while the other files would have to use the `extern` keyword. Moreover, only the declarations without the `extern` keyword could initialize values:

```
// extern would be required if const had external linkage
extern const int fingers;              // can't be initialized
extern const char * warning;
```

So you would need one set of definitions for one file and a different set of declarations for the other files. But, because externally defined `const` data have internal linkage, you can use the same declarations in all files.

Internal linkage also means that each file gets its own set of constants rather than sharing them. Each definition is private to the file containing it.

If, for some reason, you want to make a constant have external linkage, you can use the *extern* keyword to override the default internal linkage:

```
extern const states = 50;   // external linkage
```

You must use the `extern` keyword to declare the constant in *all* files using the constant. This differs from regular external variables, in which you don't use the keyword `extern` when defining a variable, but use `extern` in other files using that variable. Also, unlike regular variables, you can initialize an `extern const` value. Indeed, you have to, for `const` data requires initialization.

When you declare a `const` within a function or block, it has block scope, meaning the constant is usable only when the program is executing code within the block. This means that you can create constants within a function or block and not have to worry about the name conflicting with constants defined elsewhere.

STORAGE CLASSES AND FUNCTIONS

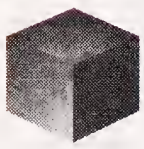
Functions, too, have storage classes, although the selection is more limited than for variables. C++, like C, does *not* allow you to define one function inside another, so all functions automatically have a static storage class, meaning they are all present as long as the program is running. By default, functions are external, meaning they have external linkage and can be shared across files. You can, in fact, use the keyword `extern` in a function prototype to indicate the function is defined in another file, but that is optional. (For the program to find the function in another file, that file must be one of the files being compiled as part of the program or a library file searched by the linker.) You can also use the keyword `static` to give a function file scope. You would apply this keyword to the prototype and to the function definition:

```
static int private(double x);
...
static int private(double x)
{
    ...
}
```

That means the function will be known only in that file. It also means you can use the same name for a different function in another file. As with variables, a static function overrides an external definition for the file containing the static declaration. So if you defined your own `strlen()` function in one file of a multifile program and declared the function `static` that one file would use your definition of `strlen()`, while the

other files would use the library version. If you defined your own `strlen()` function and did not declare it as `static`, then the compiler would use your version in all the files of your program. See the discussion on Where C++ Finds Functions.

Inline functions behave differently from regular functions. By default, they have internal linkage and hence are local to the file containing them.



WHERE C++ FINDS FUNCTIONS

Suppose you call a function in a particular file in a program. Where does C++ look for the function definition? If the function prototype in that file indicates that function is static, the compiler will look only in that file for the function. Otherwise, the compiler (and we use the word to include the linker, too) looks in all the program files. If it finds two definitions, the compiler sends you an error message, for you can have only one definition for an external function. If it fails to find any definition in your files, the function then searches the libraries. This implies that if you define a function having the same name as a library function, the compiler will use your version rather than the library version.

STORAGE CLASSES AND DYNAMIC ALLOCATION

Storage classes describe memory allocated for variables (including arrays and structures) and functions. They don't apply to memory allocated by using the C++ `new` operator (or by the older C `malloc()` function). We'll call that kind of memory *dynamic* memory. As you saw in Chapter 4, dynamic memory is controlled by the `new` and `delete` operators, not by scope and linkage rules. Thus dynamic memory can be allocated from one function and freed from another function. Unlike automatic memory, dynamic memory is not LIFO; the order of allocation and freeing depends upon when and how `new` and `delete` are used. Typically, the compiler will use three separate memory chunks: one for static variables (this chunk may be subdivided), one for automatic variables, and one for dynamic storage.

Although storage class concepts don't apply to dynamic memory, they do apply to the pointer variables used to keep track of dynamic memory. For instance, suppose we have the following statement inside a function:

```
float * p_fees = new float [20];
```

The eighty bytes (assuming a `float` is four bytes) of memory allocated by `new` remains in memory until the program expires or until the `delete` operator frees it. But the

p_fees pointer passes from existence when the function containing this declaration terminates. So if you wish to have the eighty bytes of allocated memory available to another function, you need to pass or return its address to that function. On the other hand, if you make the same declaration externally, then the *p_fees* pointer will be available to all the functions following that declaration in the file. And by using

```
extern float * p_fees;
```

in a second file, you make that same pointer available in the second file.

Pointers to Functions

No discussion of C or C++ functions would be complete without mention of pointers to functions. We'll take a quick look at this topic, leaving the full exposition of the possibilities to more advanced texts.

Functions, like data items, have addresses. A function's address is the memory address at which the stored machine language code for the function begins. Normally, it's neither important nor useful for us or the user to know that address. But it can be useful to a program. For example, it's possible to write a function that takes the address of another function as an argument. That enables the first function to find the second function and run it. This approach is more awkward than simply having the first function call the second one directly, but it leaves open the possibility of passing different function addresses at different times. And that means the first function can make use of whatever function for which we pass the address.

Function Pointer Basics

Let's clarify this process with an example. Suppose you want to design an `estimate()` function that estimates the amount of time needed to write a given number of lines of code, and you want different programmers to use the function. Part of the code for `estimate()` will be the same for all users, but the function will allow each programmer to provide his or her own algorithm for estimating time. The mechanism for that will be to pass to `estimate()` the address of the particular algorithm function the programmer wishes to use. To implement this plan, you need to be able to do the following:

- Take the address of a function.
- Declare a pointer to a function.
- Use a pointer to a function to invoke the function.

Obtaining the Address of a Function

Taking the address of a function is simple: just use the function name without trailing parentheses. That is, if `think()` is a function, then `think` is the address of the function. So to pass a function as an argument, pass the function name. Be sure you distinguish between passing the *address* of a function and passing the *return value* of a function:

```
process(think);     // passes address of think() to process()
thought(think());   // passes return value of think() to thought()
```

The `process()` call enables the `process()` function to invoke the `think()` function from within `process()`. The `thought()` call first invokes the `think()` function, then passes the return value of `think()` to the `thought()` function.

Declaring a Pointer to a Function

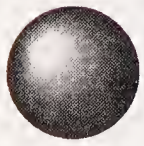
When we've declared pointers to data types, the declaration has had to specify exactly what type to which the pointer points. Similarly, a pointer to a function has to specify the type of function to which the pointer points. This means the declaration should identify the function's return type and the function's signature (its argument list). For instance, suppose Pam LeCoder has written a time-estimating function with the following prototype:

```
double pam(int); // prototype
```

Here's what a declaration of an appropriate pointer type looks like:

```
double (*pf)(int);      // pf points to a function that takes
                        // one int argument and that
                        // returns type double
```

Note that this looks just like the `pam()` declaration, with `(*pf)` playing the part of *pam*. Since *pam* is a function, so is `(*pf)`. And if `(*pf)` is a function, then `pf` is a pointer to a function.



In general, to declare a pointer to a particular kind of function, you can first write a prototype for a regular function of the desired kind, then replace the function name by an expression in the form of `(*pf)`. That will make `pf` a pointer to a function of that type.

The declaration requires the parentheses around `(*pf)` to provide the proper operator precedence. Parentheses have a higher precedence than the `*` operator, so

`*pf(int)` means `pf()` is a function that returns a pointer, while `(*pf)(int)` means `pf` is a pointer to a function:

```
double (*pf)(int);  // pf points to a function
double *pf(int);    // pf() a function that returns a pointer
```

Once you've declared `pf` properly, you can assign it the address of a matching function:

```
double pam(int);
double (*pf)(int);
pf = pam;           // pf now points to the pam() function
```

Note that `pam()` has to match `pf` in both signature and return type. The compiler rejects nonmatching assignments:

```
double ned(double);
int ted(int);
double (*pf)(int);
pf = ned;           // invalid -- mismatched signature
pf = ted;           // invalid -- mismatched return types
```

Let's return to the `estimate()` function we mentioned earlier. Suppose you want to pass it the number of lines of code to be written and the address of an estimating algorithm, such as the `pam()` function. Then it could have the following prototype:

```
void estimate(int lines, double (*pf)(int));
```

This declaration says the second argument is a pointer to a function having an *int* argument and a `double` return value. To have `estimate()` use the `pam()` function, we pass it `pam()`'s address:

```
estimate(50, pam);
```

Clearly, the tricky part about using pointers to functions is writing the prototypes, while passing the address is very simple.

Using a Pointer to Invoke a Function

Now we get to the final part of the technique, which is using a pointer to call the pointed-to function. The clue comes in the pointer declaration. There, recall, `(*pf)` played the same role as a function name. Thus all we have to do is use `(*pf)` as if it were a function name:

```
double pam(int);
double (*pf)(int);
```

```
pf = pam;                  // pf now points to the pam() function
double x = pam(4);         // call pam() using the function name
double y = (*pf)(5);       // call pam() using the pointer pf
```

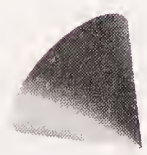
Actually, C++ also allows you to use *pf* as if it were a function name:

```
double y = pf(5);          // also call pam() using the pointer pf
```

We'll use the first form. It is uglier, but it provides a strong visual reminder that the code is using a function pointer.

(Holy syntax! How can `pf` and `(*pf)` be equivalent? Historically, one school of thought maintained that since `pf` is a pointer to a function, then `*pf` is a function, hence you should use `(*pf)()` as a function call. A second school maintained that since the name of a function is a pointer to that function, then a pointer to that function should act like the name of a function, hence you should use `pf()` as a function call. C++ takes the compromise view that both forms are correct, even though they are logically inconsistent with each other. Before you judge that compromise too harshly, reflect that the ability to hold views that are not logically self-consistent is a hallmark of the human mental process.)

Listing 8-17 demonstrates using function pointers in a program. It calls the `estimate()` function twice, once passing the `betsy()` function address, and once passing the `pam()` function address. In the first case, `estimate()` uses `betsy()` to calculate the number of hours needed, and in the second case, `estimate()` uses `pam()` for the calculation. This design facilitates future program development. When Ralph develops his own algorithm for estimating time, he doesn't have to rewrite `estimate()`. Instead, he merely need supply his own `ralph()` function, making sure it has the correct signature and return type. Of course, rewriting `estimate()` isn't a difficult task, but the same principle applies to more complex code. Also, the function pointer method allows Ralph to modify how `estimate()` works even if he doesn't have access to the source code for `estimate()`.



LISTING 8-17 fun_ptr.cpp

```
// fun_ptr.cpp -- pointers to functions
#include <iostream.h>
double betsy(int);
double pam(int);

// second argument is pointer to a type double function that
// takes a type int argument
void estimate(int lines, double (*pf)(int));
```

```
int main(void)
{
    int code;

    cout << "How many lines of code do you need? ";
    cin >> code;
    cout << "Here's Betsy's estimate:\n";
    estimate(code, betsy);
    cout << "Here's Pam's estimate:\n";
    estimate(code, pam);
    return 0;
}

double betsy(int lns)
{
    return 0.05 * lns;
}

double pam(int lns)
{
    return 0.03 * lns + 0.0004 * lns * lns;
}

void estimate(int lines, double (*pf)(int))
{
    cout << lines << " lines will take ";
    cout << (*pf)(lines) << " hour(s)\n";
}
```

Here are two sample runs:

```
How many lines of code do you need? 30
Here's Betsy's estimate:
30 lines will take 1.5 hour(s)
Here's Pam's estimate:
30 lines will take 1.26 hour(s)

How many lines of code do you need? 100
Here's Betsy's estimate:
100 lines will take 5 hour(s)
Here's Pam's estimate:
100 lines will take 7 hour(s)
```

SUMMARY

C++ has expanded C function capabilities. By using the *inline* keyword with function definition and by placing that definition ahead of the first call to that function, you suggest to the C++ compiler that it make the function inline. That is, instead of having the program jump to a separate section of code to execute the function, the compiler replaces the function call with the corresponding code inline. An inline facility should be used only when the function is short.

C++ prototypes let you define default values for arguments. If a function call omits the corresponding argument, the program uses the default value. If the function includes an argument value, the program uses that value instead of the default. Default arguments can be provided only from left to right in the argument list. Thus if you provide a default value for a particular argument, you must also provide default values for all arguments to the right of that argument.

A function's signature is its argument list. You can define two functions having the same name providing they have different signatures. This is called function polymorphism, or function overloading. Typically, you would overload functions to provide essentially the same service to different data types.

C++ encourages you to use multiple files in developing programs. An effective organizational strategy is to use a header file to define user types and provide function prototypes for functions to manipulate the user types. Use a separate source code file for the function definitions. Together, the header file and the source file define and implement the user-defined type and how it can be used. Then `main()` and other functions using those functions can go into a third file.

C++'s storage classes determine how long variables remain in memory and what parts of a program have access to them (scope and linkage). Automatic variables are those defined within a block, such as a function body or a block within the body. They exist and are known only while the program executes statements in the block containing the definition.

Static variables exist for the duration of the program. A variable defined outside of any function has external storage class. It's known to all functions in the file following its definition (file scope) and is made available to other files in the program (external linkage). For another file to use such a variable, it must declare it using the `extern` keyword. A variable defined outside any function but qualified with the keyword `static` has file scope but is not made available to other files (internal linkage). A variable defined inside a block but qualified with the keyword `static` is local to that block but retains its value for the duration of the program.

C++ functions, by default, have external storage class, so they can be shared across files. But inline functions and functions qualified with the keyword *static* have internal linkage and are confined to the defining file.

The name of a C++ function acts as the address of the function. By using a function argument that is a pointer to a function, you can pass to a function the name of a second function that you wish the first function to evoke.

Review Questions

1. What kinds of functions are good candidates for `inline` status?
2. Suppose the `song()` function has this prototype:

   ```
   void song(char * name, int times);
   ```

   a. How would you modify the prototype so that the default value for *times* is *1*?

   b. What changes would you make in the function definition?

   c. Can you provide a default value of *"O, My Papa"* for *name?*
3. Write overloaded versions of `iquote()`, a function that displays its argument enclosed in double quotation marks. Write three versions: one for an `int` argument, one for a `double` argument, and one for a string argument.
4. Here is a structure template:

   ```
   struct box
   {
      char maker[40];
      float height;
      float width;
      float length;
      float volume;
   };
   ```

   a. Write a function that passes a *box* structure by reference and that displays the value of each member.

   b. Write a function that passes a *box* structure by reference and sets the *volume* member to the product of the other three dimensions.
5. Here are some desired effects. Indicate whether each can be accomplished with default arguments, with function overloading, with both, or with neither. Provide appropriate prototypes.

   a. `mass(density, volume)` returns the mass of an object having a density of *density* and a volume of *volume*, while `mass(density)` returns the mass having a density of *density* and a volume of 1.0 cubic meters. All quantities are type `double`.

b. `repeat(10, "I'm OK")` displays the indicated string 10 times, while `repeat("But you're kind of stupid")` displays the indicated string 5 times.

c. `average(3,6)` returns the `int` average of two `int` arguments, while `average(3.0, 6.0)` returns the `double` average of two `double` values.

d. `mangle("I'm glad to meet you")` returns the character *'I'* or a pointer to the string *"I'm mad to gleet you"* depending on whether you assign the return value to a `char` variable or to a `char*` variable.

e. `average(3,6)` returns an `int` average of the two `int` arguments when called in one file, and it returns a `double` average of the two `int` arguments when called in a second file in the same program.

6. What storage class would you use for the following situations?

a. *homer* is a formal argument (parameter) to a function.

b. The *secret* variable is to be shared by two files.

c. The *topsecret* variable is to be shared by the functions in one file but hidden from other files.

d. *beencalled* keeps track of how many times the function containing it has been called.

7. Write a function that normally takes one argument, the address of a string, and prints that string once. However, if a second, type `int` argument is provided and is nonzero, the function prints the string a number of times equal to the number of times that function has been called to that point. (Yes, this is a silly function, but it makes you use some of the techniques discussed in this chapter.)

8. Design a function `calculate()` that takes two type `double` values and a pointer to a function that takes two `double` arguments and returns a `double`. The `calculate()` function also should be type `double`, and it should return the value that the pointed-to function calculates using the `double` arguments to `calculate()`. For example, suppose we have this definition for the `add()` function:

```
double add(double x, double y)
{
    return x + y;
}
```

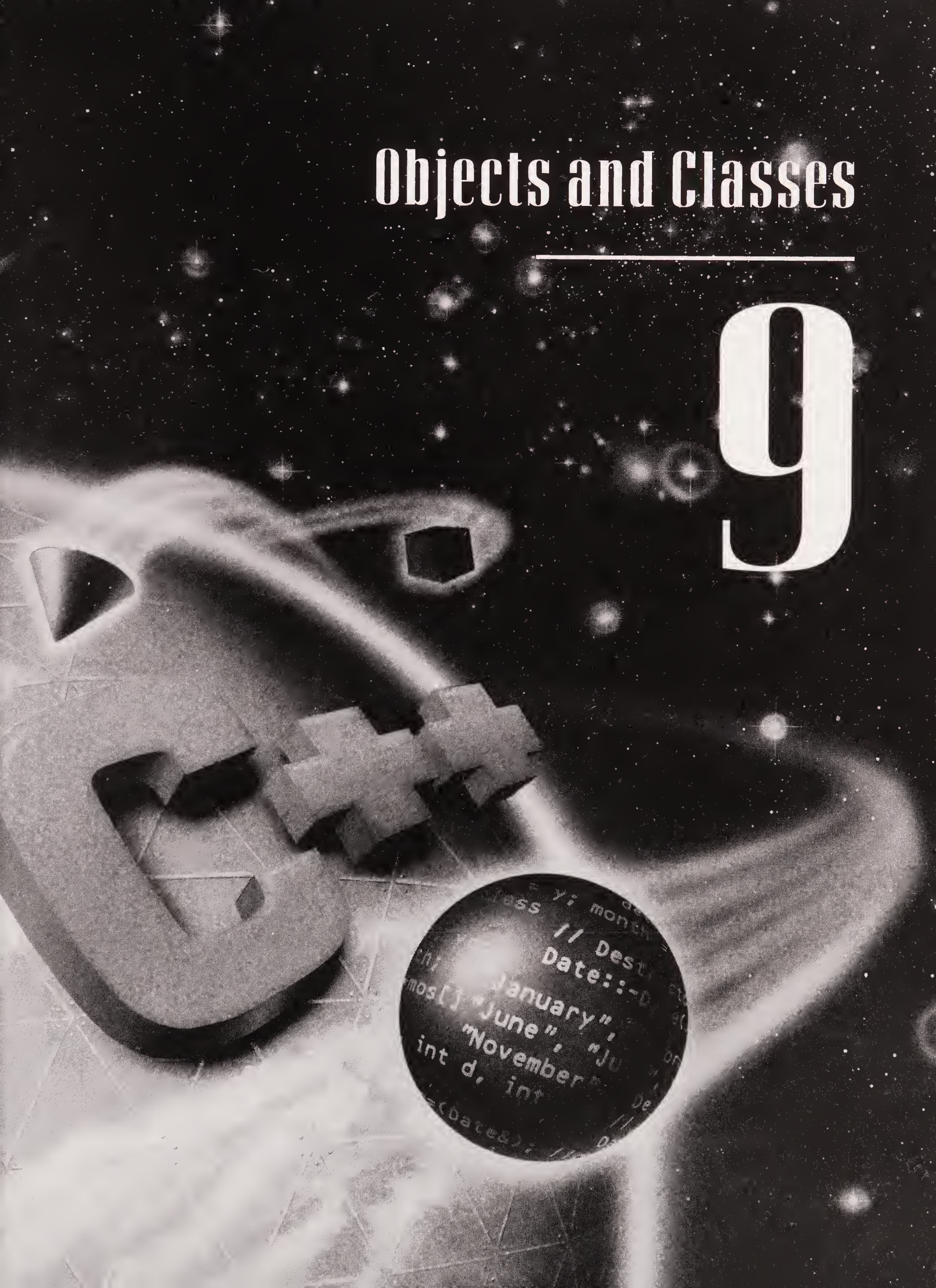
Then the function call in

```
double q = calculate(2.5, 10.4, add);
```

would cause `calculate()` to pass the value *2.5* and *10.4* to the `add()` function and then to return the `add()` return value (*12.9*).

Objects and Classes

9



Object-oriented programming (OOP) is a particular conceptual approach to designing programs, and C++ has enhanced C with features that ease the way to applying that approach. Perhaps the most important OOP features are these:

- Data abstraction
- Encapsulation and data hiding
- Polymorphism
- Inheritance

The class is the single most important C++ enhancement for implementing those features and tying them together. This chapter begins our examination of classes, explaining data abstraction, encapsulation, and data hiding, and showing how classes implement these features. It discusses how to define a class, how to provide a class with public and private sections, and how to create member functions that work with the class data. Also, the chapter acquaints you with *constructors* and *destructors*, special member functions for creating and disposing of objects belonging to a class. Finally, you'll meet the `this` pointer, an important component of some class programming. The following two chapters will extend the discussion to operator overloading (another variety of polymorphism) and inheritance.

Procedural and Object-Oriented Programming

Although we have explored the OOP perspective on programming every so often, we've usually stuck pretty close to the standard procedural approach of languages such as C, Pascal, and BASIC. Let's look at an example that clarifies how the OOP outlook differs

from that of procedural programming. As the newest member of the Genre Giants softball team, you've been asked to keep the team statistics. Naturally, you turn to your computer for help. If you were a procedural programmer, you might think along these lines:

> Let's see, I want to enter the name, times at bat, number of hits, batting averages[1], and all those other great basic statistics for each player. Wait, the computer is supposed to make life easier for me, so let's have it figure out some of that stuff, like the batting average. And I want the program to report the results. How should I organize this? I guess I should do things right and use function. Yeah, I'll make `main()` call a function to get the input, then call a function to make the calculations, and then call a function to report the results. Hmmm, what happens when I get data from the next game? I don't want to start from scratch again. Okay, I can add a function to update the statistics. Golly, maybe I'll need a menu in `main()` to select between entering, calculating, updating, and showing the data. Hmm—how am I going to represent the data? I could use an array of strings to hold the players' names, another array to hold the at bats for each player, another array to hold the hits, and so on. No, that's dumb. I can design a structure to hold all the information for a single player, then use an array of those structures to represent the whole team.

In short, you would first concentrate upon the procedures you would follow, then think about how to represent the data. (Note: So that you don't have to keep the program running the whole season, you'll probably also want to be able to save data to a file and read data from a file. But as we haven't covered files yet, we'll ignore that complication for now.)

Now let's see how your perspective changes when you don your OOP hat (an attractive polymorphic design). You would start by thinking about the data. Furthermore, you would think about the data not only in terms of how to represent it, but in terms of how it's to be used:

> Let's see—what am I keeping track of? A ball player, of course. So I want an object that represents the whole player, not just her batting average or times at bat. Yeah, that'll be my FDU (fundamental data unit), an object representing the name and statistics for a player. I'll need some methods to handle this object. Hmmm, I guess I need a method to get basic information into this unit. The computer should calculate some of the stuff, like the batting averages—I can add methods to do calculations. And the

[1]For those who don't follow baseball or softball, the batting average is the number of hits a player has divided by his official times at bats. An at bat terminates when a player gets on base or makes an out, but certain events, such as walking, don't count as official times at bat.

program should do those calculations automatically, without the user having to remember to ask to have them done. Also, I'll need methods for updating the information and displaying it. So the user gets three ways to interact with the data: initialization, updating, and reporting. That's the user interface.

In short, you would concentrate upon the object as the user perceives it, thinking about the data needed to describe the object and about the operations that will describe the user's interaction with the data. After developing a description of that interface, you'd move on to deciding how to implement the interface and data storage. Finally, you'd put together a program to use your new design.

Data Abstraction and Classes

Data abstraction is the crucial step of representing information in terms of its interface with the user. In the softball statistics example, the interface describes how the user initializes, updates, and displays the data. From data abstraction, it is a short step to the *user-defined type*, which in C++ is a class design that implements that interface.

What's a Type?

Let's think a little more about what constitutes a type. For example, what is a nerd? If you subscribe to the popular stereotype, you may think of a nerd in visual terms—thick, black-rimmed glasses, pocket protector full of pens, and so on. After a little reflection, you may conclude that a nerd is better defined operationally, for example, in how he or she responds to an awkward social situation. We have a similar situation, if you don't mind stretched analogies, with a procedural language like C. At first, you tend to think of a data type in terms of its appearance—how it is stored in memory. A `char`, for example, typically is one byte of memory, and a `double` is often eight bytes of memory. But a little reflection leads one to conclude that a data type is also defined in terms of the operations that can be performed upon it. For instance, the `int` type can use all the arithmetic operations. You can add, subtract, multiply, and divide integers. You can also use the modulus operator (`%`) with them. On the other hand, consider pointers. A pointer may very well require the same amount of memory as an `int`. It may even be represented internally as an integer. But a pointer doesn't allow the same operations that an integer does. You can't, for example, multiply two pointers times each other. The concept makes no sense, so C++ doesn't implement it. Thus when you declare a variable as an `int` or as a pointer-to-`float`, you're not just allocating memory—you are also

establishing which operations can be performed with the variable. In short, specifying a basic type does two things:

- It determines how much memory will be needed for a data object.
- It determines what operations, or methods, can be performed using the data object.

For built-in types, this information is built into the compiler. But when you define a user-defined type in C++, you have to provide the same kind of information yourself. In exchange for this extra work, you gain the power and flexibility to custom-fit new data types to match real-world requirements.

THE CLASS

The *class* is the C++ vehicle for translating a data abstraction to a user-defined type. It combines data representation and methods for manipulating that data into one classy package. Let's look at a class representing the softball information mentioned earlier. Generally, a class specification has two parts:

- A class definition, which describes the component members (both data members and function members) of the class
- The class method definitions, which describe how certain class member functions are implemented

Roughly speaking, the class definition provides a class overview, while the method definitions supply the details.

Listing 9-1 presents a class definition for a s*oftball* class. You'll notice it looks like a structure template with a few additional wrinkles, such as member functions and *public* and *private* sections. Incidentally, in concentrating on the spirit of a class definition, we've limited the statistical aspect to the basics. Softball and baseball, of course, are the stuff of a statistician's dreams. In addition to our meagre offerings, you could have daytime average, nighttime average, average against left-handed pitching, average against right-handed pitching, walks, on by errors, hit by pitches, on base average, slugging average, doubles, triples, home runs, average with players on base, with the bases loaded, with players in scoring position, and so on. And you would want to distinguish between official at bats and total at bats, for the official at bats excludes walks and sacrifices. One nice thing about the OOP approach is that once you've set up the fundamental structure for a program, it's easy to go back and add new data elements.

LISTING 9-1 class1st.cpp

```
// class1st.cpp -- our first class definition
class softball                      // softball is name of new type
{
private:                            // can be accessed only through
    char firstname[15];             // member functions
    char lastname[15];
    unsigned atbats;
    unsigned hits;
    unsigned rbis;                  // runs batted in
    float ave;
    float calc_ave(void);           // private member function
public:                             // defines public interface
    void setall(void);              // public member functions
    void update(void);
    void show_stats(void);
};
```

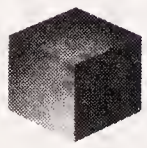
You'll get a closer look at the class definition details later, but first let's examine the more general features. To begin, the C++ keyword `class` identifies this code as being the template, or definition, of a class. The syntax identifies *softball* as being the type name for this new class. This definition enables us to declare variables, called *objects*, or *instances*, of the *softball* type. Each individual object would represent a single player. For example, the declarations

```
softball sally;
softball solly;
```

would create two *softball* objects called *sally* and *solly*.

Next, notice that the definition includes data members, such as *firstname* and *rbis*, and function members, such as `calc_ave()` and `update()`. Here prototypes represent the functions. We've talked about how the *istream* and *ostream* classes have member functions, such as `get()` and `getline()`. The function prototypes in the *softball* class definition demonstrate how member functions are established. The `iostream.h` header file, for example, has a `getline()` prototype in the *istream* class definition. Class member functions are also termed *class methods*. The full definitions for the member functions will come later, but the prototypes suffice to describe the function interfaces. This wrapping of data and methods into the same package is termed *encapsulation*. The binding of data and methods into a single unit is the most striking feature of the class. Because of encapsulation, creating a *softball* object automatically establishes the rules governing how that object can be used.

Also new are the keywords `private` and `public`. These labels describe the "visibility,"or accessibility, of class members. Any program using an object of a particular class can access the `public` portions directly. But a program can access the `private` members[2] of an object *only* by using the `public` member functions. For instance, the only way to alter the *atbats* member of the softball class is to use the `setall()` or `update()` member function. Thus the `public` member functions act as go-betweens between the program and an object's `private` members; they provide the interface between object and program. This insulation of data from direct access by a program is called *data hiding*. (C++ provides a third visibility keyword, `protected`, which we'll discuss when we cover class inheritance in Chapter 11.) See Figure 9-1. While data hiding may be an unscrupulous act in, say, a stock fund prospectus, it's a good practice in computing because it preserves the integrity of the data.



OOP AND C++

Object-oriented programming is a programming style that you can use with any language. Certainly, you can incorporate many OOP ideas into ordinary C programs. For instance, Chapter 8 provided an example in which a header file contained a structure prototype along with the prototypes for functions to manipulate that structure. Thus the `main()` function simply defined variables of that structure type and used the associated functions to handle those variables; `main()` did not directly access structure members. In essence, that example defined an abstract type that encapsulated the storage format and the function prototypes in a header file, hiding the actual data representation from *main()*. C++, however, includes features specifically intended to implement the OOP approach, so it lets you take the process a few steps further than you can with C. First, placing the data representation and the function prototypes into a single class definition instead of into a file tightens the encapsulation by placing everything in one class definition. Second, making the data representation `private` enforces the stricture that that data be accessed by authorized functions only. If, in the C example, `main()` directly accessed a structure member, it would be violating the spirit of OOP, but it wouldn't be breaking any C language rules. But trying to access, say, the *hits* member of a *softball* object directly does break a C++ language rule, and the compiler will catch it.

Note that data hiding not only prevents you from accessing the data directly, but it also absolves you from needing to know how the data is represented. To use the

[2]Some readers may find the expression "private members" vaguely unsettling, but that is the usual C++ terminology.

softball class, you don't need to know that it has a *hits* member or that the type for that member is `unsigned int`. What you do need to know is what the different member functions accomplish; that is, you need to know what kinds of arguments a member function takes and what kind of return value it has. The principle is to separate the details of the implementation from the design of the interface. If you later find a better way to implement the data representation or the details of the member functions, you can change those details without changing the program interface, and that makes programs much easier to maintain.

Public or Private?

Class members, whether they be data items or member functions, can be declared either in the `public` section or the `private` section of a class. But because one of the main OOP precepts is to hide the data, data items normally go into the `private` section. The member functions that constitute the class interface go into the `public` section; otherwise, you couldn't call those functions from a program. As our sample definition shows, you can also put member functions in the `private` section. You can't call such functions directly from a program, but the `public` methods can use them. Typically, you

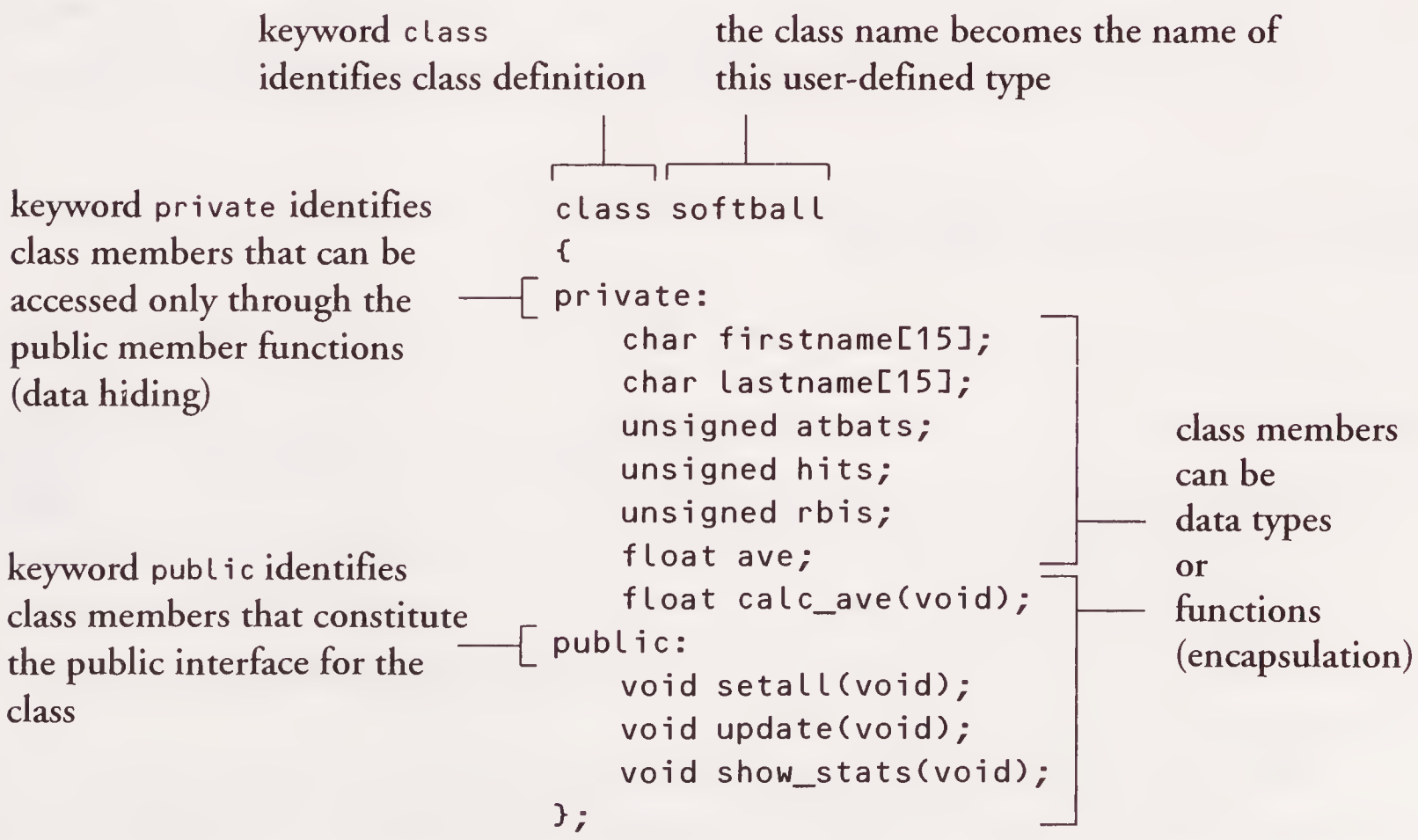


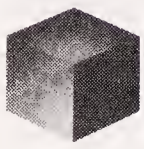
Figure 9-1 *The* softball *class*

would use `private` member functions to handle implementation details that don't form part of the public interface.

You don't have to use the keyword `private` in class definitions, for that is the default visibility for class objects:

```
class world
{
    float mass;                    // private by default
    char name[20];                 // private by default
public:
    void tellall(void);
    ...
}
```

However, we'll use the `private` label explicitly in order to emphasize the concept of data hiding.



CLASSES AND STRUCTURES

Class descriptions look much like structure templates with the addition of member functions and of the `public` and `private` visibility labels. In fact, C++ has extended to structures the same features classes have. The only difference is that the default visibility type for a structure is `public`, while the default type for the class is `private`. But C++ programmers commonly use classes to implement class descriptions while restricting structures to represent pure data objects.

IMPLEMENTING CLASS MEMBER FUNCTIONS

We still have the second part of the class specification to do: providing code for the member functions. Let's look at that next. Member function definitions are much like regular function definitions. They have a function heading and a function body. They can have return types and arguments. But they also have two special characteristics:

- When defining a member function, you use the scope operator (*::*) to identify the class to which the function belongs.
- Class methods can access the `private` components of the class.

Let's look at these points now.

First, the function heading for a member function uses the scope operator (*::*) to indicate to which class the function belongs. For example, the heading for the `setall()` member function looks like this:

```
void softball::setall()
```

This notation means we are defining the `setall()` function that is a member of the *softball* class. Not only does this identify `setall()` as a member function, it means we can use the same name for a member function for a different class. For instance, the heading for a `setall()` function for a *buffoon* class would have this function heading:

```
void buffoon::setall()
```

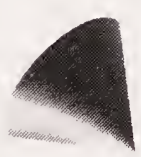
Thus the scope resolution operator resolves the question of which class a method definition applies to. We say that the identifier `setall()` has *class scope*. Other member functions of the *softball* class can, if necessary, use the `setall()` method without using the scope resolution operator. That's because they belong to the same class, making `setall()` in scope. Using `setall()` outside of the class definition and method definitions, however, will require special measures, which we'll get to soon.

The second special characteristic for methods is that the function can access the `private` members of a class. For instance, the `show_stats()` method can use code like this:

```
cout << "Player: " << firstname << " " << lastname << "\n";
```

Here *firstname* and *lastname* are private data members of the *softball* class. If you try to use a nonmember function to access these data members, the compiler will stop you cold in your tracks. (However, *friend* functions, discussed in Chapter 10, provide an exception.)

With these two points in mind, we can implement the class methods as shown in Listing 9-2. These method definitions can go in a separate file or in the same file with the class definition. Since we are starting simple, we'll assume that these definitions follow the class definition in the same file. This is the easiest, although not the best, way to make the class definition available to the method definitions. (The best way, which we'll apply later, is to use a header file for the class definition and a source code file for the class member function definitions.)



LISTING 9-2 class1st.cpp

```
//more class 1st.cpp -- implementing the class member functions
#include <iostream.h>
float softball::calc_ave(void) // belongs to softball class
{
    if (atbats == 0)
            return 0;
    else
            return float(hits) / atbats;
}

void softball::setall(void)    // sets initial values
```

```
{
    cout << "Enter player's first name: ";
    cin >> firstname;
    cout << "Enter player's last name: ";
    cin >> lastname;
    cout << "Enter number of at bats: ";
    cin >> atbats;
    cout << "Enter number of hits: ";
    cin >> hits;
    cout << "Enter number of RBIs: ";
    cin >> rbis;
    ave = calc_ave();
    cout << "\n";
}


void softball::update(void)    // revises stats
{
    int temp;

    cout << "Enter stats for " << firstname << " ";
    cout << lastname << ":\n";
    cout << "Enter additional number of at bats: ";
    cin >> temp;
    atbats = atbats + temp;
    cout << "Enter additional number of hits: ";
    cin >> temp;
    hits = hits + temp;
    cout << "Enter additional number of RBIs: ";
    cin >> temp;
    rbis = rbis + temp;
    ave = calc_ave();
    cout << "\n";
}


void softball::show_stats(void)       // display data
{
    cout << "Player: " << firstname << " " << lastname << "\n";
    cout << "At bats: " << atbats << "   Hits: " << hits;
    cout << "   RBIS: " << rbis << "   BA: " << ave << "\n\n";
}
```

Member Function Notes

We made `calc_ave()` a `private` method because the user doesn't need direct access to this function. Instead, the `setall()` and `update()` functions call `calc_ave()` to calculate a player's batting average once they've determined the number of hits and the number of at bats. By the way, note the following statement from `calc_ave`:

```
return float(hits) / atbats;
```

The number of hits and the number of at bats are integer quantities, which means the expression `hits/atbats` represents integer division. That would evaluate to zero if *hits* is less than *atbats*, so we've used a C++ style type cast to make the numerator type `float`, compelling the compiler to use floating-point division. Notice that we did not use the form `float(hits/atbats)`, for that would still use integer division, then convert the result (most likely a zero) to a floating-point. You also can use the older C notation for a type cast:

```
return (float) hits / atbats;
```

The `setall()` method prompts the user to enter name information along with some batting statistics. The `update()` method lets the user enter information about the number of additional at bats and the like. It uses the name information already provided by `setall()`. Finally, `show_stats()` displays all the information about a player.

Defining a Method within the Class Definition

The last example separated the method definitions from the class definition, but you don't have to do that. For instance, you could define the *softball* class this way:

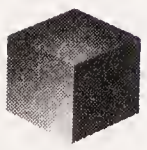
```
// class1st.cpp -- our first class definition
class softball                  // softball is name of new type
{
private:                        // can be accessed only through
    char firstname[15];         // member functions
    char lastname[15];
    unsigned atbats;
    unsigned hits;
    unsigned rbis;              // runs batted in
    float ave;
    float calc_ave()            // method definition
    {
```

```
            if (atbats == 0)
                    return 0;
            else
                    return float(hits) / atbats;
    }
public:                                 // defines public interface
    void setall(void);                  // public member functions
    void update(void);
    void show_stats(void);
};
```

Here we've replaced the prototype for `calc_ave()` with the complete function definition. When you define a function within a class definition, C++ makes that function an inline function (Chapter 8). Thus, if you use this approach at all, you should use it only with very short functions. We will continue to define member functions outside the class definition because that is more in accord with the principle of separating details of implementation from the class description. You can, if you like, define a member function outside the class definition and still make it inline. To do so, just use the `inline` qualifier when defining the function in the class implementation section:

```
class softball
{
private:
      ...
      float calc_ave();                 // definition kept separate
public:
      ...
};

inline float softball::calc_ave()       // use inline in definition
{
    ...
}
```



COMPATIBILITY NOTE

This two-part approach to creating inline member functions didn't work properly in early versions of Turbo C++ if the inline member functions were used in a file other than the one containing the definition. However, this mechanism does work in Borland C++ and newer releases of Turbo C++.

Which Object?

Now we've come to one of the most important aspects of using objects: how you apply a class method to an object. Code such as

```
cout << "Player: " << firstname << " " << lastname << "\n";
```

uses the *firstname* and *lastname* members of an object. But which object? That's an excellent question! To answer it, first consider how you create an object. The simplest way is to declare class variables:

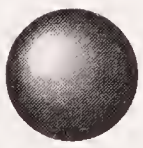
```
softball kate, joe;
```

This creates two objects of the *softball* class, one named *kate* and one named *joe*.

Next, consider how to use a member function with one of these objects. The answer, as with structures and structure members, is to use the membership operator:

```
kate.show_stats();     // the kate object calls the member function
joe.show_stats();      // the joe object calls the member function
```

The first call invokes `show_stats()` as a member of the *kate* object. This means the method will interpret *firstname* as *kate.firstname* and *lastname* as *kate.lastname*. Similarly, the call `joe.show_stats()` makes the `show_stats()` method interpret *firstname* and *lastname* as *joe.firstname* and *joe.lastname*.



When you call a member function, it uses the data members of the particular object used to invoke the member function.

Similarly, the function call `kate.setall()` invokes the `calc_ave()` function as if it were `kate.calc_ave()`, causing that function to get its data from the *kate* object.

Each new object you create contains storage for its own internal variables, the class members. But all objects of the same class share the same set of class methods. Suppose, for instance, that *kate* and *joe* are *softball* objects. Then `kate.hits` occupies one chunk of memory and `joe.hits` occupies a second chunk of memory. But `kate.show_stats()` and `joe.show_stats()` both invoke the same method, that is, both execute the same block of code. They just apply the code to different data. Calling a member function is what some OOP languages term sending a message. Thus sending the same message to two different objects invokes the same method but applies it to two different objects. See Figure 9-2.

USING A CLASS

Now you've seen how to define a class and its class methods. The next step is to produce a program that creates and uses objects of a class. The C++ goal is to make using classes as similar as possible to using the basic, built-in types, such as `int` and `char`. So you can create a class object by declaring a class variable or using *new* to allocate an object of a class type. You can pass objects as arguments, return them as function return values, and assign one object to another. C++ provides facilities for initializing objects, for teaching `cin` and `cout` to recognize objects, and even for providing automatic type conversions between objects of similar classes. It will be a while before you can do all these things,

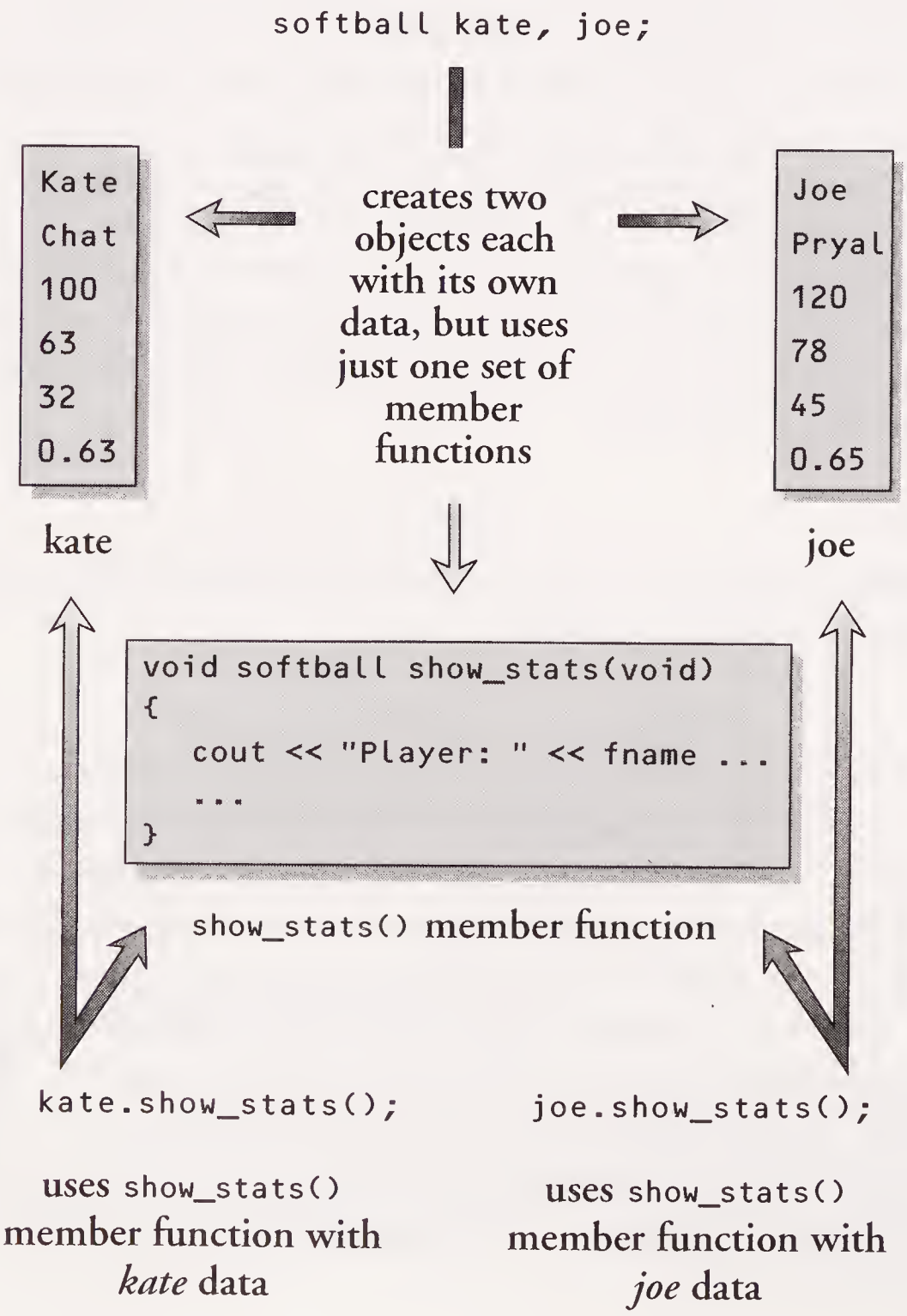


Figure 9-2 *Objects, data, and member functions*

but let's start now with the simpler properties. Indeed, you've already seen how to declare a class object and to call a member function. Listing 9-3 combines those techniques with the class definition and the member function definitions to form a complete program. It creates two objects, one representing the position of left field, and one representing center field. The program is simple, but it does test the features we built into the class.

LISTING 9-3 class1st.cpp

```
// class1st.cpp complete -- use a class
#include <iostream.h>

// class definition
class softball
{
private:                              // can be accessed only through
    char firstname[15];               // member functions
    char lastname[15];
    unsigned atbats;
    unsigned hits;
    unsigned rbis;
    float ave;
    float calc_ave();                 // private member function
public:                               // defines public interface
    void setall();                    // public member functions
    void update();
    void show_stats();
};

// class methods
float softball::calc_ave()            // belongs to softball class
{
    if (atbats == 0)
            return 0;
    else
            return float(hits) / atbats;
}

void softball::setall()
{
    cout << "Enter player's first name: ";
    cin >> firstname;
    cout << "Enter player's last name: ";
    cin >> lastname;
    cout << "Enter number of at bats: ";
```

```
    cin >> atbats;
    cout << "Enter number of hits: ";
    cin >> hits;
    cout << "Enter number of RBIs: ";
    cin >> rbis;
    ave = calc_ave();
    cout << "\n";
}

void softball::update() // revises stats
{
    cout << "Enter stats for " << firstname << " ";
    cout << lastname << ":\n";
    cout << "Enter additional number of at bats: ";
    int temp;
    cin >> temp;
    atbats = atbats + temp;
    cout << "Enter additional number of hits: ";
    cin >> temp;
    hits = hits + temp;
    cout << "Enter additional number of RBIs: ";
    cin >> temp;
    rbis = rbis + temp;
    ave = calc_ave();
    cout << "\n";
}

void softball::show_stats()
{
    cout << "Player: " << firstname << " " << lastname << "\n";
    cout << "At bats: " << atbats << "   Hits: " << hits;
    cout << "   RBIS: " << rbis << "   BA: " << ave << "\n\n";
}

// test program
int main(void)
{
    cout << "Batter OOP!\n";

    softball left, center;      // try with two objects
    left.setall();
    center.setall();
    left.show_stats();
    center.show_stats();
    left.update();
    center.update();
```

```
    left.show_stats();
    center.show_stats();
    cout << "bye\n";
    return 0;
}
```

Here's a sample run:

```
Batter OOP!
Enter player's first name: Kimball
Enter player's last name: Kinnison
Enter number of at bats: 520
Enter number of hits: 232
Enter number of RBIs: 143

Enter player's first name: Hari
Enter player's last name: Seldon
Enter number of at bats: 480
Enter number of hits: 135
Enter number of RBIs: 38

Player: Kimball Kinnison
At bats: 520      Hits: 232      RBIS: 143      BA: 0.446154

Player: Hari Seldon
At bats: 480      Hits: 135      RBIS: 38       BA: 0.28125

Enter stats for Kimball Kinnison:
Enter additional number of at bats: 24
Enter additional number of hits: 11
Enter additional number of RBIs: 4

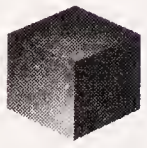
Enter stats for Hari Seldon:
Enter additional number of at bats: 21
Enter additional number of hits: 9
Enter additional number of RBIs: 3

Player: Kimball Kinnison
At bats: 544      Hits: 243      RBIS: 147      BA: 0.446691

Player: Hari Seldon
At bats: 501      Hits: 144      RBIS: 41       BA: 0.287425

bye
```

As promised, calling `left.setall()` causes the `setall()` method to use the *left* object, and so on. Note that `main()` is just a vehicle for testing the design of the *softball* class. Given that the class works as we like, we can now use the *softball* class as a user-defined type in other programs. The critical point in using the new type is understanding what the member functions do; you shouldn't have to think about the implementation details. See the discussion on the Client-Server Model.



THE CLIENT-SERVER MODEL

OOP programmers often discuss program design in terms of a client-server model. In this conceptualization, the client is a program that uses the class. The class definition, including the class methods, constitute the server, which is a resource available to the programs that need it. The client uses the server through the publicly defined interface only. This means the client's only responsibility, and, by extension, the client's programmer's only responsibility, is to know that interface. The server's responsibility, and, by extension, the server's designer's responsibility, is to see that the server reliably and accurately performs according to that interface. Any changes the server designer makes to the class design should be to details of implementation, not to the interface. This allows programmers to improve the client and the server independently of each other, without changes in the server having unforeseen repercussions in the client's behavior.

OUR STORY TO DATE

The first step in specifying a class design is providing a class definition. The class definition is modeled after a structure template and can include data members and function members. The definition has a `private` section, and members declared in that section can be accessed only through the member functions. The definition also has a `public` section, and members declared there can be accessed by an outside program. Typically, data members go into the `private` section and member functions go into the `public` section, so a typical class definition has this form:

```
class classname
{
private:
data member declarations
public:
member function prototypes
};
```

Putting the data in the `private` section protects the integrity of the data and is called data hiding. Bundling the data together with the member functions describing how the data can be used is called *encapsulation*. Thus the class is the C++ way of making it easy to implement the OOP goals of data hiding and encapsulation.

The second step in specifying the class design is to implement the class member functions. You can use a complete function definition instead of a function prototype in the class definition, but the usual practice, except, possibly, for very brief functions, is to provide the function definitions separately. In that case, you need to use the scope operator to indicate to which class a member function belongs. For instance, suppose the *bozo* class has a member function called `retort()` that returns a pointer to a `char`. Then the function heading would look like this:

```
char * bozo::retort();
```

In other words, `retort()` is not just a type `char *` function; it is a type `char *` function that belongs to the *bozo* class.

To create an object, which is a particular instance, or example, of a class, use the class name as if it were a type name:

```
bozo bozetta;
```

This works because a class *is* a user-defined type.

A class member function, or method, can be invoked only by a class object. You can do so by using the dot membership operator:

```
cout << bozetta.retort();
```

This invokes the `retort()` member function, and whenever the code for that function refers to a particular data member, the function will use the value that member has in the *bozetta* object.

Class Constructors and Destructors

Meanwhile, there's more to be done with the *softball* class. There are certain standard functions, called *constructors* and *destructors*, you usually should provide for a class. Let's see why they are needed and how to write them.

One of C++'s aims is to make using class objects similar to using standard types. However, you can't yet initialize a *softball* object the way you can an ordinary `int` or `struct`:

```
int year = 2001;                                    // okay
struct thing
{
    char * pn;
```

```
    int m;
};
thing amabob = {"widget", -23};                          //okay
softball babe = {"Babe", "Whoosh", 200, 10, 2};          // NO!
```

The reason you can't initialize a *softball* object this way is because the data parts have `private` visibility, which means a program cannot access the data members directly. As you've seen, the only way a program can access the data members is through a member function. Therefore, we need to devise an appropriate member function if we're to succeed in initializing an object. (You could initialize a class object as shown above *if* you made the data members `public` instead of private, but making the data `public` goes against one of the main justifications for classes, that is, data hiding.)

C++ provides a special member function, called a class `constructor`, especially for constructing a new object and assigning values to its data members. More precisely, C++ provides a name for this member function and a syntax for using it, and you provide the method definition. The name is the same as the class name. For instance, the constructor for the *softball* class is a member function called `softball()`. The constructor prototype and heading have an interesting property—although the constructor has no return value, it's not declared type `void`. In fact, a constructor has no declared type.

DECLARING AND DEFINING A CONSTRUCTOR

Let's build a *softball* constructor. Because a *softball* object has five values to be provided from the outside world, we'll give the constructor five arguments. (The sixth value, the *ave* member, is calculated from *atbats* and *hits*, so we don't have to provide it to the constructor.) Possibly, we'll just want to set the two name members and set the other values to zero, so we'll provide default arguments (see Chapter 8). Thus the prototype would look like this:

```
// constructor prototype with some default arguments
softball(char * fn, char * ln, unsigned ab = 0,
         unsigned h = 0, unsigned rbi = 0);
```

The two arguments are pointers to the two strings that will be used to initialize the *firstname* and *lastname* character array class members. The *ab*, *h*, and *rbi* variables will provide values for the *atbats*, *hits*, and *rbi* members. Note that there is no return type. The prototype goes in the `public` section of the class definition.

Next, here's one possible definition for the constructor:

```
// constructor definition
softball:: softball(char * fn, char * ln, unsigned ab,
              unsigned h, unsigned rbi)      // class constructor
{
    strcpy(firstname, fn);
    strcpy(lastname, ln);
    atbats = ab;
    hits = h;
    rbis = rbi;
    ave = calc_ave();
}
```

The string arguments to this function are copied to the corresponding character arrays in the object using the standard C library function `strcpy()`. Note that you'll have to include the `string.h` file to provide prototypes for the `strcpy()` function. Next, the function copies the other arguments to the corresponding object members, then uses the private member function to calculate a value for the *ave* member. You should place the code for the constructor along with the code for the other member functions.

Incidentally, the definition doesn't check to see if the initialization string fits into the target array. If this bothers you, you can use the `strncpy()` function instead. The call `strncpy(s2, s1, n)` copies *s1* to *s2* or else up to *n* characters from *s1* to *s2*, whichever comes first. If *s1* contains fewer characters than *n*, the `strncpy()` function pads *s2* with null characters. That is, `strncpy(firstname,"Tim", 6)` would copy the characters *T*, *i*, and *m* to *firstname*, then add 3 null characters to bring the total to 6 characters. But if *s1* is longer than *n*, no null characters are appended. That is, `strncpy(firstname, "Priscilla", 4)` just copies the characters *P*, *r*, *i*, and *s* to *firstname*, making it a character array, but, since it lacks a terminating null character, not a string. So to make sure the constructor creates a true string, you could use code like this:

```
// constructor definition
softball:: softball(char * fn, char * ln, unsigned ab,
              unsigned h, unsigned rbi)      // class constructor
{
    strncpy(firstname, fn, 14);              // use first 14 elements
    firstname[14] = '\0';                    // set 15th element to '\0'
    ...
}
```

We'll continue, however, to assume benign input that doesn't require such special treatment.

USING A CONSTRUCTOR

C++ provides two ways to initialize an object using the constructor. The first is to call the constructor explicitly:

```
softball third = softball("James", "Kirk", 20, 9, 5);
```

This sets the *firstname* member of the *third* object to the string *"James"*, the *lastname* member to *"Kirk"*, and so on.

The second way is to call the constructor implicitly:

```
softball rover("Perry", "Mason", 18, 4, 1);
```

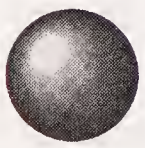
This more compact form is equivalent to the following explicit call:

```
softball rover = softball("Perry", "Mason", 18, 4, 1);
```

If you provide a class constructor, C++ uses it whenever you create an object of that class, even when you use *new* for dynamic memory allocation. Here's how to use the constructor with *new*:

```
softball *psoft = new softball("Winnie", "Pooh", 23, 2, 0);
```

This statement creates a *softball* object, initializes it to the values provided by the arguments, and assigns the address of the object to the *psoft* pointer. In this case, the object doesn't have a name, but you can use the pointer to manage the object. We'll put off further discussion of pointers to objects until Chapter 10.



A class constructor, if defined, is called whenever a program creates an object of that class.

DESTRUCTORS

When you use a constructor to create an object, the program undertakes the responsibility of tracking that object until it expires. At that time, the program automatically calls a special member function bearing the formidable title of `destructor`. The destructor should clean up any debris, so it actually serves a constructive purpose. For instance, if your constructor uses `new` to allocate memory, the destructor should use `delete` to free that memory. Our constructor doesn't do anything fancy like using `new`, so it doesn't really need a destructor. But it's a good idea to provide one anyway in case a future class revision needs one.

Like a constructor, the destructor has a special name: the class name preceded by a tilde (~). Thus the destructor for the *softball* class would be called *~softball()*. Also, like a constructor, a destructor can have no return value and has no declared type. Unlike the constructor, the destructor can have no arguments. Thus the prototype for a *softball* destructor must be this:

```
~softball(void);
```

Because a *softball* destructor has no vital duties, we can code it as a do-nothing function:

```
softball::~softball(void)
{
}
```

In fact, if you fail to provide a destructor explicitly, the compiler provides the equivalent of this do-nothing destructor implicitly.

However, just so that we can see when the destructor is called, we'll code it this way:

```
softball::~softball(void)              // class destructor
{
    cout << "Bye, " << firstname << "!\n";
}
```

IMPROVING THE SOFTBALL CLASS

The next step is to incorporate the constructor and the destructor into the class and method definitions. This time we'll follow the usual C++ practice and organize the program into separate files. We'll place the class definition in a header file called `softbal1.h`. (As the name suggests, we have future revisions in mind.) The class methods go into a file called `softbal1.cpp`. In general, the header file containing the class definition and the source code file containing the methods definitions should have the same base name so that you can keep track of which files belong together. Using separate files for the class definition and the member functions separates the abstract definition of the interface (the class definition) from the details of implementation (the member functions). You could, for example, distribute the class definition as a text header file but distribute the function definitions as compiled code. Finally, we'll place the program using these resources in a third file, which we'll call `constrct.cpp`.

Listing 9-4 shows the header file. It adds prototypes for the constructor and destructor functions to the original class definition.

LISTING 9-4 softball.h

```
// softbal1.h -- header file for softball class
class softball
{
private:
    char firstname[15];
    char lastname[15];
    unsigned atbats;
    unsigned hits;
    unsigned rbis;
    float ave;
    float calc_ave(void);
public:
    softball(char * fn, char * ln, unsigned ab = 0,
       unsigned h = 0, unsigned rbi = 0);    // class constructor
    ~softball(void);                         // class destructor
    void setall(void);
    void update(void);
    void show_stats(void);
};
```

Listing 9-5 provides the method definitions. It includes the `softbal1.h` file in order to provide the class definition. Also, it includes the system `iostream.h` and `string.h` files because the methods use `cin`, `cout`, and `strcmp()`. This file adds the constructor and destructor method definitions to the prior methods.

LISTING 9-5 softbal1.cpp

```
// softbal1.cpp -- methods for softball class
#include <iostream.h>
#include <string.h>
#include "softbal1.h"

softball:: softball(char * fn, char * ln, unsigned ab,
        unsigned h, unsigned rbi)     // class constructor
{
    strcpy(firstname, fn);
    strcpy(lastname, ln);
    atbats = ab;
    hits = h;
    rbis = rbi;
    ave = calc_ave();
}
```

```
softball::~softball(void)                    // class destructor
{
    cout << "Bye, " << firstname << "!\n";
}

float softball::calc_ave(void)
{
    if (atbats == 0)
            return 0;
    else
            return float(hits) / atbats;
}

void softball::setall(void)
{
    cout << "Enter player's first name: ";
    cin >> firstname;
    cout << "Enter player's last name: ";
    cin >> lastname;
    cout << "Enter number of at bats: ";
    cin >> atbats;
    cout << "Enter number of hits: ";
    cin >> hits;
    cout << "Enter number of RBIs: ";
    cin >> rbis;
    ave = calc_ave();
    cout << "\n";
}

void softball::update(void)
{
    cout << "Enter stats for " << firstname << " ";
    cout << lastname << ":\n";
    cout << "Enter additional number of at bats: ";
    int temp;
    cin >> temp;
    atbats = atbats + temp;
    cout << "Enter additional number of hits: ";
    cin >> temp;
    hits = hits + temp;
    cout << "Enter additional number of RBIs: ";
    cin >> temp;
    rbis = rbis + temp;
    ave = calc_ave();
    cout << "\n";
}
```

```
void softball::show_stats(void)
{
    cout << "Player: " << firstname << " " << lastname << "\n";
    cout << "At bats: " << atbats << "   Hits: " << hits;
    cout << "   RBIS: " << rbis << "   BA: " << ave << "\n\n";
}
```

Listing 9-6 provides a short program for testing the new methods. Like `softbal1.cpp`, it includes the `softbal1.h` file to provide the class definition. The program creates two objects, one representing the first baseman and one representing the shortstop. (For the uninitiated, these terms describe where on the field a softball player plays when his or her team is on the defensive.) The players are literary detectives of various degrees of fame. To compile the complete program, use the techniques for multifile programs described in Chapters 1 and 8.

LISTING 9-6 constrct.cpp

```
// constrct.cpp -- class constructor and destructor
// compile with softbal1.cpp
#include <iostream.h>
#include "softbal1.h"

int main(void)
{
    // You can use constructors to initialize new objects
    softball first = softball("Jane", "Marples",
            200, 60, 32);                          // syntax 1
    softball shortstop("Nero", "Wolfe");           // syntax 2

    first.show_stats();
    shortstop.show_stats();
    // You can use constructors to reset object values
    shortstop = softball("Freddie", "ThePig", 322, 103, 43);
    shortstop.show_stats();
    cout << "bye\n";
    return 0;
}
```

Here is the program output:

```
Player: Jane Marples
At bats: 200   Hits: 60   RBIS: 32   BA: 0.3

Player: Nero Wolfe
At bats: 0   Hits: 0   RBIS: 0   BA: 0
```

```
Player: Freddie ThePig
At bats: 322   Hits: 103   RBIS: 43   BA: 0.319876

bye
Bye, Freddie!
Bye, Freddie!
Bye, Jane!
```

Program Notes

The statement

```
softball first = softball("Jane", "Marples",
          200, 60, 32);
```

creates a *softball* object called *first* and initializes its data members to the indicated values. The statement

```
softball shortstop("Nero", "Wolfe");
```

uses the second variety of syntax to create an object called *shortstop* and to initialize its name members to *"Nero"* and *"Wolfe"*. Because the program doesn't provide further arguments, the constructor uses the default values for the remaining arguments. As the output shows, the defaults are all *0*.

You can use the constructor for more than initializing a new object. For instance, the program has this statement in `main()`:

```
shortstop = softball("Freddie", "ThePig", 322, 103, 43);
```

The *shortstop* object already exists. Thus instead of initializing s*hortstop*, this statement assigns new values to the object.

Next, note the destructor at work.

```
Bye, Freddie!
Bye, Freddie!
Bye, Jane!
```

The *first* and *shortstop* objects are local variables, so they expire when the containing block expires. In this case, the containing block is the `main()` function, so when `main()` terminates, the objects die, and the destructor function goes into action. Because these objects are automatic variables, they're represented by a stack; that means the last object created is the first deleted and the first object created is the last deleted. As you can see, *first (Jane)* got deleted last, and *shortstop* (now *Freddie*) got deleted just prior to that. Curiously, although we only declared two objects, we get three calls to the destructor function! The reason is that when we used the constructor to reassign Freddie's values to the *shortstop* object, the program first created a temporary, nameless

object having the provided values. Then those values were copied to the *shortstop* object. Finally, when the temporary object expired, the program called the destructor function for it. So, the first *Freddie* deleted is the temporary object, and the second *Freddie* deleted is the *shortstop* object. Incidentally, C++ doesn't really specify when a temporary object is disposed. This implementation (Turbo C++ 2.0) treats one like an automatic variable and deleted the temporary object when the function terminated, but you can't count on a program waiting that long.

FURTHER REFINEMENTS: THE DEFAULT CONSTRUCTOR

Once you define a constructor, a program *must* use it when creating an object. Because the `softball()` constructor requires at least two arguments (the first and last names of the player), you now always have to explicitly initialize a *softball* object when creating one. That is, you no longer can create uninitialized objects as in Listing 9-3:

```
softball left, center; // not possible with current constructor
```

Yet this is something you might want to do. For example, the `setall()` method lets you use keyboard input to provide values to an uninitialized object.

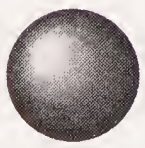
The key to creating uninitialized objects when you have a constructor is to define a *default constructor*. This is a constructor that takes no arguments. You can define a default constructor two ways. One is to provide default values for all the arguments to the existing constructor:

```
softball(char * fn = "", char * ln = "", unsigned ab = 0,
         unsigned h = 0, unsigned rbi = 0);
```

The second is to use function overloading to define a second constructor that has no arguments:

```
softball(void);
```

(Prior to version 2.0 of C++, you could use only the second method for creating a default constructor.)



When you create constructors for a class, provide a default constructor if you wish to create uninitialized objects.

Once you've used either method to create the default constructor, then you can declare object variables without initializing them explicitly:

```
softball first;   // calls default constructor implicitly
softball first = softball();   // calls it explicitly
softball *prelief = new softball;      // calls it implicitly
```

However, don't be misled by the implicit form of the nondefault constructor:

```
softball first("Han", "Solo");      // calls constructor
softball second();                  // declares a function
softball catcher;                   // calls default constructor
```

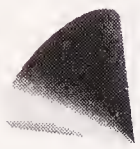
The first declaration calls the nondefault constructor, that is, the one that takes arguments. The second declaration states that `second()` is a function that returns a *softball* object. So when calling the default constructor implicitly, don't use parentheses.

Listing 9-7 is the header file for a new class definition incorporating the default constructor. To avoid confusion with the previous header file, we'll call this one `softbal2.h`.

LISTING 9-7 softbal2.h

```
// softbal2.h -- header file for softball class
class softball
{
private:
    char firstname[15];
    char lastname[15];
    unsigned atbats;
    unsigned hits;
    unsigned rbis;
    float ave;
    float calc_ave(void);
public:
    softball(char * fn, char * ln, unsigned ab = 0,
         unsigned h = 0, unsigned rbi = 0);       // constructor
    softball(void);                               // default constructor
    ~softball(void);                              // class destructor
    void setall(void);
    void update(void);
    void show_stats(void);
};
```

Listing 9-8 adds the default constructor to the methods file. It defines a constructor that does nothing. The constructor's sole purpose is to exist so that you can declare objects without explicitly initializing them. Note that a default constructor still can initialize an object implicitly. This one doesn't, but you can define the function so that it sets the names to empty strings and the other values to zero.



LISTING 9-8 softbal2.cpp

```
// softbal2.cpp -- methods for softball class
#include <iostream.h>
#include <string.h>
#include "softbal2.h"

softball::softball(void)        // default constructor
{
}

softball:: softball(char * fn, char * ln, unsigned ab,
                    unsigned h, unsigned rbi)
{
    strcpy(firstname, fn);
    strcpy(lastname, ln);
    atbats = ab;
    hits = h;
    rbis = rbi;
    ave = calc_ave();
}

softball::~softball(void)       // class destructor redefined
{
    cout << "Bye, " << firstname << "!\n";
}

float softball::calc_ave(void)
{
    if (atbats == 0)
            return 0;
    else
            return float(hits) / atbats;
}

void softball::setall(void)
{
    cout << "Enter player's first name: ";
    cin >> firstname;
    cout << "Enter player's last name: ";
    cin >> lastname;
    cout << "Enter number of at bats: ";
    cin >> atbats;
    cout << "Enter number of hits: ";
    cin >> hits;
    cout << "Enter number of RBIs: ";
    cin >> rbis;
```

```
    ave = calc_ave();
    cout << "\n";
}

void softball::update(void)
{
    cout << "Enter stats for " << firstname << " ";
   cout << lastname << ":\n";
    cout << "Enter additional number of at bats: ";
    int temp;
    cin >> temp;
    atbats = atbats + temp;
    cout << "Enter additional number of hits: ";
    cin >> temp;
    hits = hits + temp;
    cout << "Enter additional number of RBIs: ";
    cin >> temp;
    rbis = rbis + temp;
    ave = calc_ave();
    cout << "\n";
}

void softball::show_stats(void)
{
    cout << "Player: " << firstname << " " << lastname << "\n";
    cout << "At bats: " << atbats << "   Hits: " << hits;
    cout << "   RBIS: " << rbis << "   BA: " << ave << "\n\n";
}
```

Finally, Listing 9-9 puts the new constructor to use. Don't forget that you must compile it together with Listing 9-8.

LISTING 9-9 dfltcnst.cpp

```
// dfltcnst.cpp -- use default class constructor
// compile with softbal2.cpp
#include <iostream.h>
#include "softbal2.h"

int main(void)
{
    softball first = softball("Jane", "Marples",
            200, 60, 32);
    softball shortstop("Nero", "Wolfe");
```

```
    softball tops;                  // uses default constructor

    cout << "Playing first base is\n";
    first.show_stats();
    cout << "Playing shortstop is \n";
    shortstop.show_stats();
    tops = first;                   // class assignment
    cout << "The top batter is\n";
    tops.show_stats();
    cout << "bye\n";
    return 0;
}
```

Here is the program's output:

```
Playing first base is
Player: Jane Marples
At bats: 200      Hits: 60      RBIS: 32      BA: 0.3

Playing shortstop is
Player: Nero Wolfe
At bats: 0        Hits: 0       RBIS: 0       BA: 0

The top batter is
Player: Jane Marples
At bats: 200      Hits: 60      RBIS: 32      BA: 0.3

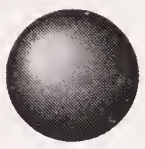
bye
Bye, Jane!
Bye, Nero!
Bye, Jane!
```

Program Notes

The program uses the original constructor to initialize the *first* and *shortstop* objects, and it uses the default constructor to create the empty *tops* object. After displaying the contents of the first two objects, the program uses the assignment operator to copy the contents of the *first* object to the *tops* object:

```
tops = first;                   // class assignment
```

C++ allows you to assign one object to another of the same class, just as it allows structure assignment.



When you assign one object to another of the same class, C++, by default, copies the contents of each data member of the source object to the corresponding data member of the target object.

Using `show_stats()` with the *tops* object demonstrates that the copying was successful. And when `main()` ends, the program calls the destructor function for each of the three objects.

CONSTRUCTORS AND DESTRUCTORS IN REVIEW

Now that we've gone through a few examples of constructors and destructors, you may wish to pause and assimilate what has passed. To help you, here is a summary of these methods.

A constructor is a special class member function that's called whenever an object of that class is created. A class constructor has the same name as its class, but, through the miracle of function overloading, you can have more than one constructor with the same name, providing each has its own signature, or argument list. Also, a constructor has no declared type. Usually, the constructor is used to initialize members of a class object. Your initialization should match the constructor's argument list. For instance, suppose the *bozo* class has the following prototype for a class constructor:

```
bozo(char * fname, char * lname);      // constructor prototype
```

Then you would use it to initialize new objects as follows:

```
bozo bozetta = bozo("Bozetta", "Biggens");      // primary form
bozo fufu("Fufu", "O'Dweeb");                   // short form
bozo *pc = new bozo("Popo", "Le Peu");          // dynamic object
```

If a constructor has just one argument, that constructor is invoked if you initialize an object to a value having the same type as the constructor argument. For instance, suppose you have this constructor prototype:

```
bozo(int age);
```

Then you can use any of the following forms to initialize an object:

```
bozo dribble = bozo(44);                  // primary form
bozo roon(66);                            // secondary form
bozo tubby = 32;    // special form for one-argument constructors
```

(Actually, this a new point, not a review point, but it seemed like a nice time to tell you about it.)

Once a constructor has been defined, you cannot create an object of that class without using a constructor. Therefore, if you want to create an object without initializing it, you must also define a default constructor, which is one with no arguments:

```
bozo();                    // default constructor prototype
```

The program uses the default constructor for uninitialized objects:

```
bozo bubi;          // use default
bozo *pb = new bozo;      // use default
```

Just as a program invokes a constructor when an object is created, it invokes a destructor when an object is destroyed. You can have only one destructor per class. It has no return type, not even void, it has no arguments, and its name is the class name preceded with a tilde. The *bozo* class destructor, for example, has the following prototype:

```
~bozo();                   // class destructor
```

Class destructors become necessary when class constructors use `new`.

Knowing Your Objects: the this Pointer

There's still more to be done with the *softball* class. So far each class member function has dealt with but a single object, which has been the object that invokes it. Sometimes, however, a method may need to deal with two objects, and doing so may involve a curious C++ pointer called `this`. Let's see how this need may unfold.

In Listing 9-9, the program assigns the object representing the higher batting average to the *tops* object. But the program didn't determine for itself which was the higher average; it merely accepted the choice provided by the assignment statement. The problem is that the program has no way of knowing what the batting averages are, for those are `private` data. If you want the program to determine the higher average, you have to provide a member function to do the job. This raises some interesting questions, and we'll look into them now.

First, how do you provide the member function with two objects to compare? Suppose, for example, you decide to name the method `topbat()`. Then the message `first.topbat()` accesses the data of the *first* object, while the message `shortstop.topbat()` accesses the data of the *shortstop* object. If you want the method to compare *two* objects, you have to pass the second object as an argument. For efficiency, pass the argument by reference. That is, have the `topbat()` method use a type `softball &` argument.

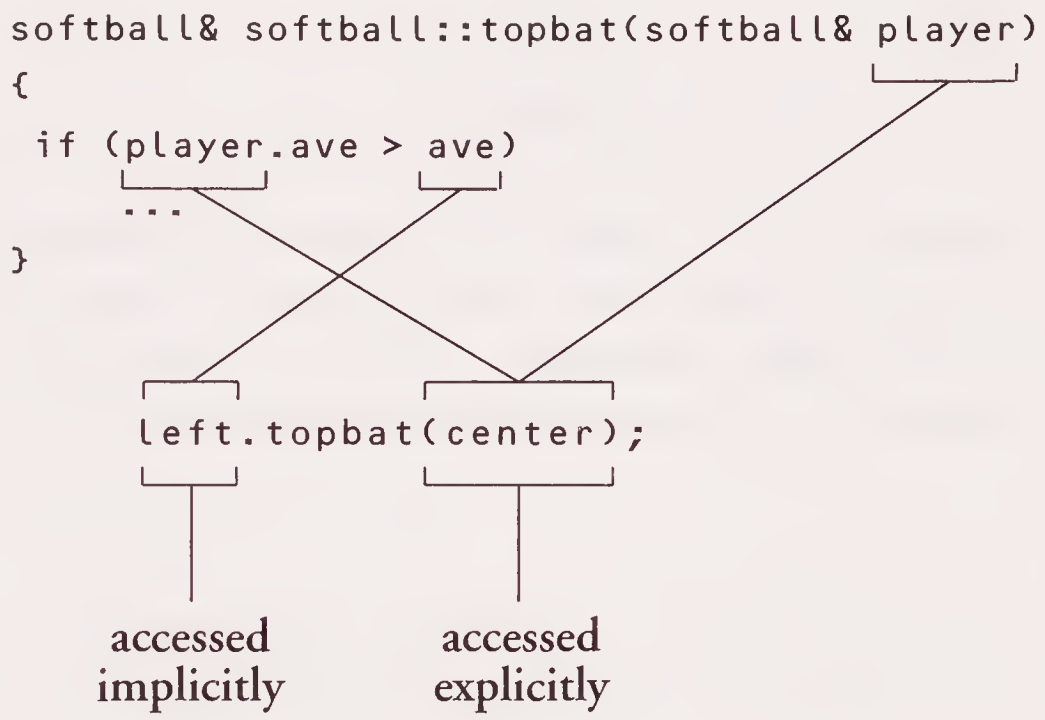


Figure 9-3 *Accessing two objects with a member function*

Second, how do you communicate the method's answer back to the calling program? The most direct way is to have the method return a reference to the object having the larger batting average. Thus the comparison method should have the following prototype:

```
softball & topbat(softball & player);
```

This function accesses one object implicitly, one object explicitly, and it returns a reference to one of those two objects.

Suppose, then, that you want to compare *softball* objects *left* and *shortstop* and assign the one with the greater batting average to the object *top*. You can use either of the following statements:

```
top = left.topbat(shortstop);
top = shortstop.topbat(left);
```

The first form accesses *left* implicitly and *shortstop* explicitly, while the second accesses *left* explicitly and *shortstop* implicitly. See Figure 9-3. Either way, the method compares the two objects and returns a reference to the winner.

Actually, this notation is a bit confusing. It would be clearer if we could somehow use the relational operator > to compare the two objects. And you can do so with operator overloading, which we'll discuss in Chapter 10.

Meanwhile, there's still the implementation of *topbat()* to attend to. And that raises a slight problem. Here's a partial implementation that highlights the problem:

```
softball& softball::topbat(softball& player)
{
```

```
        if (player.ave > ave)
                return player;
        else
                return ????;
}
```

Here `player.ave` is the average for the object passed as an argument, and *ave* is the average for the object to which the message is sent. If `player.ave` is greater than *ave*, the function returns *player*. Otherwise, it returns the object used to evoke the method. (In OOP talk, that would be the object to which the *topbat* message is sent.) The prob-

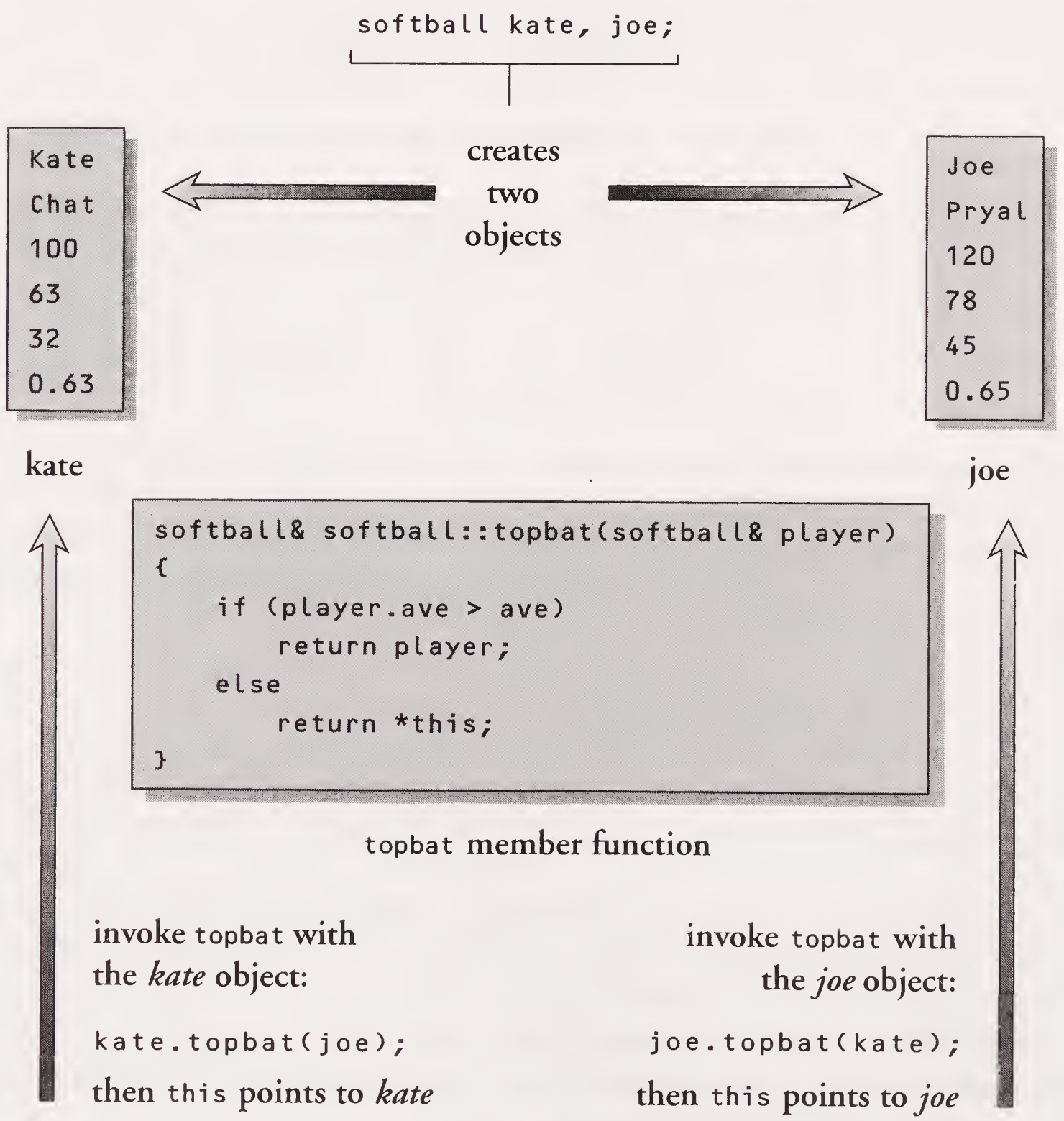


Figure 9-4 `this` *points to the invoking object*

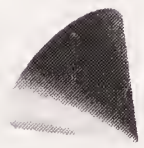
lem is, what do you call that object? If you make the call `first.topbat(shortstop)`, then *player* is a reference for *shortstop* (i.e., an alias for *shortstop*), but there is no alias for *first*.

The C++ solution to this problem is a special pointer called `this`. The `this` pointer points to the object used to invoke a member function. Thus the function call `first.topbat(shortstop)` sets `this` to the address of the *first* object and makes that pointer available to the `topbat()` method. Similarly, the function call `shortstop.topbat(first)` sets `this` to the address of the *shortstop* object. In general, all class methods have a `this` pointer set to the address of the object invoking the method. Indeed, *ave* in `topbat()` is just shorthand notation for `this->ave`. (Recall from Chapter 4 that you use the -> operator to access structure members via a pointer. The same is true for class members.) See Figure 9-4.

What we want to return, however, is not `this`, for it is the address of the object. We want to return the object itself, and that is symbolized by `*this`. (Recall that applying the dereferencing operator * to a pointer yields the value to which the pointer points.) Now we can complete the method definition by using `*this` as an alias for the invoking object.

```
softball& softball::topbat(softball& player)
{
    if (player.ave > ave)
        return player;          // argument object
    else
        return *this;           // invoking object
}
```

Listing 9-10 shows the new header file.



LISTING 9-10 softbal3.h

```
// softbal3.h -- header file for softball class
class softball
{
private:
    char firstname[15];
    char lastname[15];
    unsigned atbats;
    unsigned hits;
    unsigned rbis;
    float ave;
    float calc_ave(void);
public:
    softball(char * fn, char * ln, unsigned ab = 0,
       unsigned h = 0, unsigned rbi = 0);      // constructor
```

```
    softball(void);                              // default constructor
    ~softball(void);                             // class destructor
    void setall(void);
    void update(void);
    void show_stats(void);
    softball & topbat(softball & player);        //new method
};
```

Listing 9-11 presents the revised class methods file. It includes the new `topbat()` method. Also, now that we've seen how the destructor method works, we'll replace it with a do-nothing version.

LISTING 9-11 softbal3.cpp

```
// softbal3.cpp -- methods for softball class
#include <iostream.h>
#include <string.h>
#include "softbal3.h"

softball:: softball(void)
{
}

softball:: softball(char * fn, char * ln, unsigned ab,
                    unsigned h, unsigned rbi)
{
    strcpy(firstname, fn);
    strcpy(lastname, ln);
    atbats = ab;
    hits = h;
    rbis = rbi;
    ave = calc_ave();
}

softball::~softball(void)   // redefine destructor
{
}

float softball::calc_ave(void)
{
    if (atbats == 0)
        return 0;
    else
```

```
        return float(hits) / atbats;
}

void softball::setall(void)
{
    cout << "Enter player's first name: ";
    cin >> firstname;
    cout << "Enter player's last name: ";
    cin >> lastname;
    cout << "Enter number of at bats: ";
    cin >> atbats;
    cout << "Enter number of hits: ";
    cin >> hits;
    cout << "Enter number of RBIs: ";
    cin >> rbis;
    ave = calc_ave();
    cout << "\n";
}

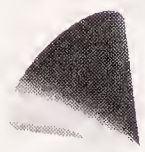
void softball::update(void)
{
    cout << "Enter stats for " << firstname << " ";
    cout << lastname << ":\n";
    cout << "Enter additional number of at bats: ";
    int temp;
    cin >> temp;
    atbats = atbats + temp;
    cout << "Enter additional number of hits: ";
    cin >> temp;
    hits = hits + temp;
    cout << "Enter additional number of RBIs: ";
    cin >> temp;
    rbis = rbis + temp;
    ave = calc_ave();
    cout << "\n";
}

void softball::show_stats(void)
{
    cout << "Player: " << firstname << " " << lastname << "\n";
    cout << "At bats: " << atbats << "   Hits: " << hits;
    cout << "   RBIS: " << rbis << "   BA: " << ave << "\n\n";
}

softball& softball::topbat(softball& player)
{
    if (player.ave > ave)
```

```
            return player;
    else
            return *this;
}
```

Of course, we want to see if the `this` pointer works, and Listing 9-12 provides a short program for checking it out. (Don't forget to compile Listings 9-11 and 9-12 together.)



LISTING 9-12 usethis.cpp

```
// usethis.cpp -- the this pointer and reference variables
// compile with softbal3.ccp
#include <iostream.h>
#include "softbal3.h"

int main(void)
{
    softball catcher = softball("Andrew", "Dalziel",
            420, 160, 86);
    softball pitcher("Adam", "Dalgliesh", 218, 78, 38);
    softball tops;

    catcher.show_stats();
    pitcher.show_stats();
    tops = catcher.topbat(pitcher);   // reference argument
    cout << "Here is the high batting average:\n";
    tops.show_stats();
    cout << "bye\n";
    return 0;
}
```

Here is the program output:

```
Player: Andrew Dalziel
At bats: 420      Hits: 160     RBIS: 86      BA: 0.380952

Player: Adam Dalgliesh
At bats: 218      Hits: 78      RBIS: 38      BA: 0.357798

Here is the high batting average:
Player: Andrew Dalziel
At bats: 420      Hits: 160     RBIS: 86      BA: 0.380952

bye
```

Well, the `this` pointer worked! Here, `this` pointed to the *catcher* object, and `topbat()` returned a reference to that object.

Each member function, including constructors and destructors, can use its own `this` pointer. The only special property of the `this` pointer is that it initially points to the invoking object. Because it is not a `const` pointer, you *can* make it point to some other object. But that doesn't mean you *should* use it that way. Most often, if you do use `this`, particularly in constructors and destructors, you should not change its value.

An Array of Objects

Often, as with the *softball* examples so far, you'll want to create several objects of the same class. You can create separate object variables, as we have done so far, but it may make more sense to create an array of objects. That may sound like a major leap into the unknown, but, in fact, you declare an array of objects the same way you would an array of any of the standard types:

```
softball outfield[4]; // creates an array of 4 softball objects
```

Recall that a program always calls a class constructor, if one exists, when creating class objects. So this declaration requires either that we have no constructor defined or, as in our case, that we have a default constructor defined. Each element—*outfield[0], outfield[1],* and so on—is a *softball* object and thus can be used with the *softball* methods:

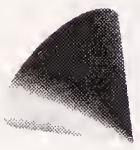
```
outfield[0].setall();          // apply setall() to 1st element
outfield[3].show_stats();      // apply show_stats() to 4th element
softball tops = outfield[2].topbat(outfield[1]);
                               // compare 3rd and 2nd elements
```

You can use a constructor to initialize the array elements. In that case, you have to call the constructor for each individual element:

```
softball outfield[4] =
{
    softball("Agatha", "Christie", 240, 82, 34),
    softball("Jack", "Vance", 202, 77, 36),
    softball("Martha", "Grimes", 234, 75, 28),
    softball("Robert", "Heinlein", 222, 80, 49)
};
```

Here we've used the standard form for initializing an array: a comma-separated list of values enclosed in braces. In this case, each value is represented by a call to the constructor method.

Listing 9-13 applies these principles to a short program that initializes four array elements, displays their contents, and tests the elements to find the one with the highest batting average. Because `topbat()` examines just two objects at a time, we use a `for` loop to examine the whole array. We've made no further changes in the class definition, so you can use the header file and methods file shown in Listings 9-10 and 9-11.



LISTING 9-13 objarray.cpp

```
// objarray.cpp -- using an array of objects
// compile with softbal3.ccp
#include <iostream.h>
#include "softbal3.h"
const int Num = 4;
int main(void)
{
// NOTE: some implementations may require using the keyword
// static when declaring the outfield array

    softball outfield[Num] =
    {
            softball("Agatha", "Christie", 240, 82, 34),
            softball("Jack", "Vance", 202, 77, 36),
            softball("Martha", "Grimes", 234, 75, 28),
            softball("Robert", "Heinlein", 222, 80, 49)
    };
    softball tops;

    for (int i = 0; i < Num; i++)
            outfield[i].show_stats();

    tops = outfield[0];
    for (i = 1; i < Num; i++)
            tops = tops.topbat(outfield[i]);
    cout << "Tops:\n";
    tops.show_stats();

    cout << "bye\n";
    return 0;
}
```

Here is the output:

```
Player: Agatha Christie
At bats: 240      Hits: 82      RBIS: 34      BA: 0.341667
```

```
Player: Jack Vance
At bats: 202      Hits: 77      RBIS: 36      BA: 0.381188

Player: Martha Grimes
At bats: 234      Hits: 75      RBIS: 28      BA: 0.320513

Player: Robert Heinlein
At bats: 222      Hits: 80      RBIS: 49      BA: 0.36036

Tops:
Player: Jack Vance
At bats: 202      Hits: 77      RBIS: 36      BA: 0.381188

bye
```

Class Scope

Chapter 8 discussed file scope and block scope. A variable with file scope, recall, can be used anywhere in the file containing its definition, while a variable with block scope is local to the block containing its definition. Function names, too, can have file scope, but they never have block scope. C++ classes introduce a new kind of scope: class scope. Class scope applies to names defined in a class, such as the names of class data members and of class member functions. Items having class scope are known within the class but not outside the class. Thus you can use the same class member names in different classes without conflict: the `ave` member of the `softball` class would be a distinct variable from the `ave` member of a *bowling* class. Also, class scope means you can't access members of a class directly from the outside world. This is true even for `public` members. For example, to invoke a public member function, you have to use an object:

```
softball catcher;     // create object
catcher.setall();     // use object to invoke a member function
setall();             // invalid -- can't call method directly
```

Similarly, we had to use the scope resolution operator when defining member functions:

```
void softball::setall(void)
{
    ...
}
```

In short, within a class definition or within a member function definition you can use an unadorned member name, as when `setall()` calls the `calc_ave()` member function. Otherwise, you must use the direct membership operator (.) or the indirect membership operator (->) or the scope resolution operator (::), depending on the context, when using a class member name.

Summary

Object-oriented programming emphasizes how a program represents data. The first step to solving a programming problem using the OOP approach is to describe the data in terms of its interface with the program, specifying how the data is used. Next, design a class that implements the interface. Typically, private data members store the information, while public member functions, also called methods, provide the only access to the data. The class encapsulates data and methods into one unit, and the private aspect accomplishes data hiding.

Usually, you separate the class definition into two parts, typically kept in separate files. The class definition proper goes into a header file, with the methods being represented by function prototypes. The source code that defines the member functions goes into a methods file. This approach separates the description of the interface from the details of the implementation. In principle, you need know only the public class interface to use the class. Of course, you can look at the implementation (unless it's been supplied to you in compiled form only), but your program shouldn't rely upon details of the implementation, such as knowing a particular value is stored as an `int`. As long as a program and a class communicate only through methods defining the interface, you are free to improve either part separately without worrying about unforeseen interactions.

A class is a user-defined type, and an object is an instance of a class. That means an object is a variable of that type or the equivalent of a variable, such as memory allocated by *new* according to the class specification. C++ tries to make user-defined types as similar as possible to standard types, so you can declare objects, pointers to objects, and arrays of objects. You can pass objects as arguments, return them as function return values, and assign one object to another of the same type. If you provide a constructor method, you can initialize objects when they are created. If you provide a destructor method, the program will execute that method when the object expires.

Each object holds its own copies of the data portion of a class definition, but they share the class methods. If *mr_object* is the name of a particular object and `try_me()` is a member function, you invoke the member function by using the dot membership operator: `mr_object.try_me()`. OOP terminology describes this function call as sending a `try_me` message to the *mr_object* object. Any reference to class data members in the `try_me()` method will then apply to the data members of the *mr_object* object. Similarly, the function call `i_object.try_me()` would access the data members of the *i_object* object.

If you want a member function to act upon more than one object, you can pass additional objects to the method as arguments. If a method needs to refer explicitly to the object that evoked it, it can use the `this` pointer. The `this` pointer is set to the address of the evoking object, so `*this` is an alias for the object itself.

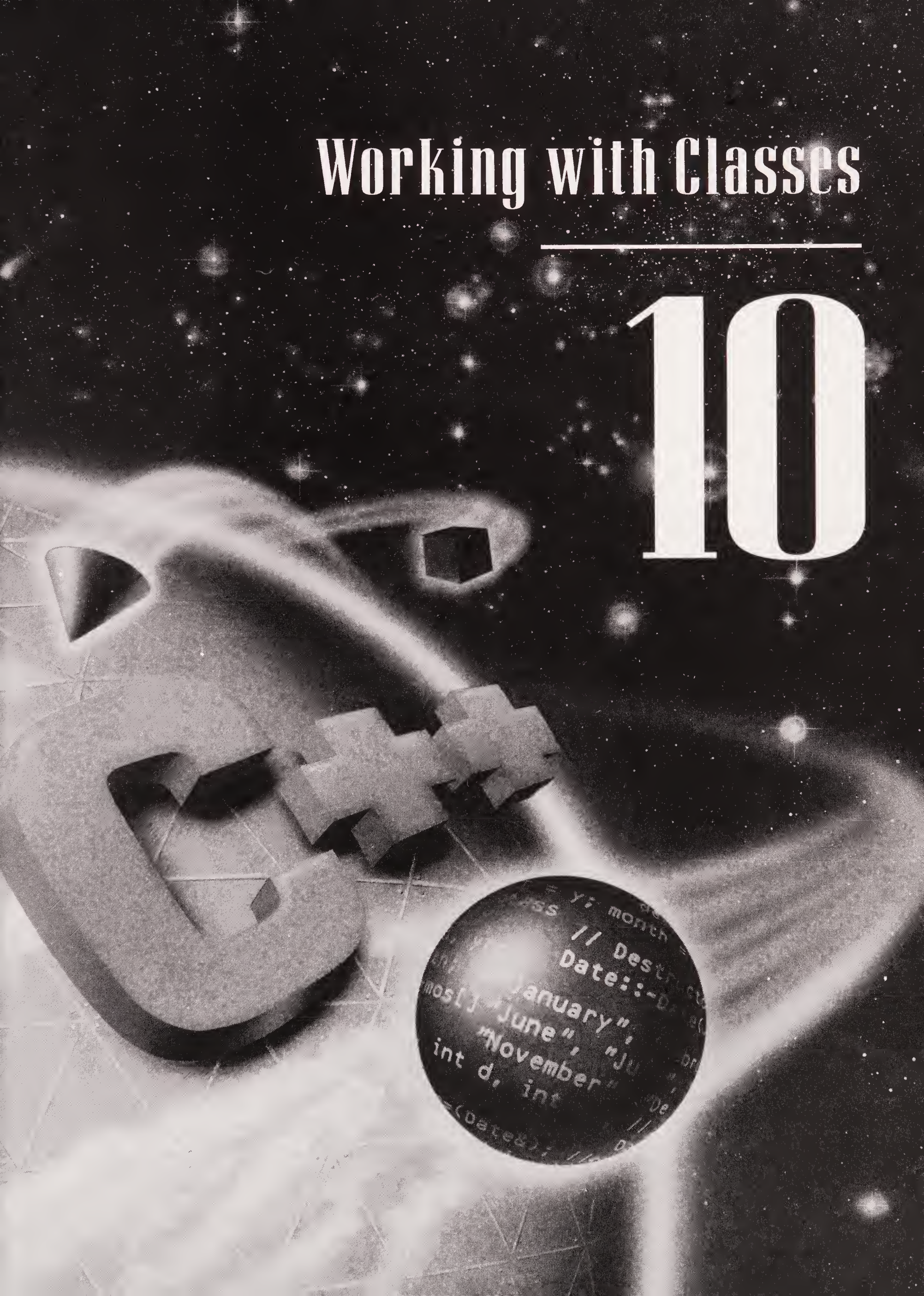
Review Questions

1. What is a class?
2. How does a class accomplish encapsulation and data hiding?
3. What is the relationship between an object and a class?
4. In what way, aside from being functions, are class function members different from class data members?
5. Define a class to represent a bank account. Data members should include the depositor's name, the account number (use a string), and the balance. Member functions should allow the following:
   - Assigning starting values to the data members
   - Displaying the depositor's name, account number, and balance
   - Depositing an amount of money given by an argument
   - Withdrawing an amount of money given by an argument

   Write a short program that uses the class.
6. When are class constructors called? When are class destructors called?
7. Provide a constructor for the bank account class of question 5.
8. What is a default constructor and what's the advantage of having one?
9. A slugging average is calculated by dividing the total bases by the times at bat. To calculate total bases, count a single as one base, a double as two bases, a triple as three bases, and a home run as four bases. Modify `softbal3.h` and `softbal3.cpp` so the class accepts input of, calculates, and displays the new statistics described here.
10. What are `this` and `*this`?

Working with Classes

10



C++ classes are feature-rich, complex, and powerful. In Chapter 9 you began a journey toward becoming an OOPerator by learning to define and use a simple class. You saw how a class encapsulates a data type by defining the type of data to be used to represent an object and by also defining, through member functions, the operations that can be performed with that data. And you learned about two special member functions, the constructor and the destructor, that manage creating and discarding objects made to a class specification. This chapter will take you a few steps further in the exploration of class properties. You'll probably find some of these features straightforward, some a bit more subtle. To best understand these new features, you should try the examples and experiment with them. What happens if I use a regular argument instead of a reference argument for this function? What happens if I leave something out of a destructor? Don't be afraid to make mistakes, for usually you can learn more from unraveling an error than by doing something correctly, but by rote. (On the other hand, don't assume that a maelstrom of mistakes inevitably leads to incredible insight.) In the end, you'll be rewarded with a fuller understanding of how C++ works and of what C++ can do for you.

This chapter starts with friends, the C++ mechanism for letting nonmember functions access private data. Then it examines operator overloading, which lets you use standard C++ operators such as = and + with class objects. Next, it looks at how you can instruct C++ to perform automatic type conversions with classes. Finally, the chapter looks at how to use `new` and `delete` with classes and how to deal with some of the subtle problems using dynamic memory can cause. This may sound like a short list of topics, but these topics have several ramifications. In particular, as you go through this chapter, you'll gain a greater appreciation for the roles class constructors and class destructors play. Also, you'll see some of the stages you may go through as you develop and improve a class design.

One difficulty with learning C++, at least by the time you've gotten this far into the subject, is that there is an awful lot to remember. And it's unreasonable to expect to remember it all until you've logged enough experience on which to hang your memories. Learning C++, in this respect, is like learning a feature-laden word processor or spreadsheet program. No one feature is that daunting, but, in practice, most people really know well only those features they use regularly, such as searching for text or italicizing . You may recall having read somewhere how to generate alternative characters or create a table of contents, but those skills probably won't be part of your daily repertoire until you find yourself in a situation in which you need them frequently. Probably the best approach to absorbing the wealth of material in this chapter is to begin incorporating just some of these new features into your own C++ programming. As your experiences enhance your understanding and appreciation of these features, begin adding other C++ features. As Bjarne Stroustrop, the creator of C++, suggested at a C++ conference for professional programmers: "Ease yourself into the language. Don't feel you have to use all of the features, and don't try to use them all on the first day."[1]

Let's look at a specific example of how C++ can add to your memory load. Suppose you want to create a class with a member representing someone's last name. The simplest way is to use a character array member to hold the name. But this has some drawbacks. You might use a 14-character array, then run into Hugh Smeadsbury-Crafthovingham. Or, to be safer, you may use a 40-character array. But, if you then create an array of 2000 such objects, you'll be wasting a lot of memory with character arrays that are only partly filled. (At this point we're adding to the *computer's* memory load.) There is an alternative. OOP-thought holds that it is much better to decide many matters, such as how much storage to use, when a program runs than when it's compiled. The usual C++ approach to storing a name in an object is to use the `new` operator in a class constructor to allocate the correct amount of memory while the program is running. But introducing `new` to a class constructor raises several new problems unless you remember to take a series of additional steps, such as expanding the class destructor, bringing all constructors into harmony with the new destructor, and writing additional class methods to facilitate correct initialization and assignment! (This chapter, of course, will explain all these steps.) If you're just learning C++, you may be better off sticking initially to the simple, if inferior, character array approach. Then, when a class design works well, you can return to your OOP workbench and enhance the class definition by using `new`. In short, grow gradually into C++.

[1] *Dr. Dobb's Journal*, December 1990, Volume 15, Issue 12, page 6.

FRIENDS

One of the main features of C++ classes is that they support the great OOP goal of data hiding. A class, recall, encapsulates data and methods into a single unit and allows you to change private data only by working through member functions. This controlled access makes it impossible to abuse, misuse, or lose the data, providing you design the member functions properly. It's a bit like a library at which professional librarians manage the checking out, returning, and reshelving of books. Such a library tends to be better organized than one in which users reshelve books upon returning them, perhaps filing *Gone with the Wind* under meteorology or *An American Life* under nonfiction. However, C++ is pragmatic enough to allow some room for finagling, and that is what C++ friends are for. A *friend function* is a nonmember function that's granted access to a class's *private* members, much as you might allow a trusted and properly briefed friend access to your Jerry Lewis recordings or to lunch with you at your private club. Similarly, a *friend class* is a class whose member functions can access another class's *private* members.

One of the most common reasons for using a friend function is when you need a function to access two separate classes. Logically, such a function should be a member of both classes. That isn't possible, but you can make the function a friend to both classes, belonging to neither, and able to access both.

To see how this works, let's begin with a possibly somewhat artificial example. Suppose you've developed a specialty concern that handles business affairs for marriages in which the wife sings opera and the husband dives undersea. (Both live by their lungs, so the match is a natural one.) You might then establish a class for each type. Listing 10-1 shows two such class definitions.

LISTING 10-1 jobs.h

```
// jobs.h -- class definitions for diva and diver

// forward declaration tells compiler of subsequent definition
class diver;

// diva class definition
class diva
{
private:
    char firstname[15];
    char lastname[15];
```

```
    char range[20];
    double annual_take;
public:
    diva(void);
    diva(char * fn, char * ln, char * rng = "",
           double inc = 0);
    ~diva(void);
    void show_diva();
    friend double total_income(const diva & d1, const diver & d2);
};

// diver class definition
class diver
{
private:
    char firstname[15];
    char lastname[15];
    double max_depth;
    double annual_take;
public:
    diver(void);
    diver(char * fn, char * ln, double md = 0.0,
           double inc = 0);
    ~diver(void);
    void show_diver();
    friend double total_income(const diva & d1, const diver & d2);
};
```

The file defines two classes: *diva* and *diver*. As usual, each has a `private` section holding data and a `public` section holding prototypes for the class member functions. Additionally, the last statement in each class definition declares the friend function:

```
friend double total_income(const diva & d1, const diver & d2);
```

The keyword `friend` identifies `total_income()` as a `friend` function, meaning that it is not a class member function but that it can access `private` class members. See Figure 10-1. Because `total_income()` is a friend to both the *diva* and the *diver* classes, it can access data from both.

Notice the short declaration near the beginning of the header file:

```
class diver;
```

This *forward declaration* prepares the way for the friend declaration in the *diva* class definition. In the *diva* definition, the argument list for `total_income()` uses the *diva* and *diver* types. At that point in the file, the *diva* name has already been used to identify one class definition, but the *diver* definition still lies ahead. If we were to omit the forward definition, then the compiler would not have heard of the *diver* type when it

reaches the first friend declaration, hence the compiler would generate an error message. And if we move the entire *diver* definition ahead of the *diva* definition, the compiler wouldn't know what the *diva* type was when reaching the friend declaration in the *diver* definition. The forward declaration solves this problem by announcing that *diver* is a class name. That's enough to alert the compiler to withhold complaining about unknown types until it's seen the rest of the story. See Figure 10-2.

DESIGNING FRIENDS

You have to design a friend function differently from a member function. First, a friend function uses a different method of determining which object to use. Then, because of that, the friend function uses a different way to refer to members of that object. Let's look at these two points.

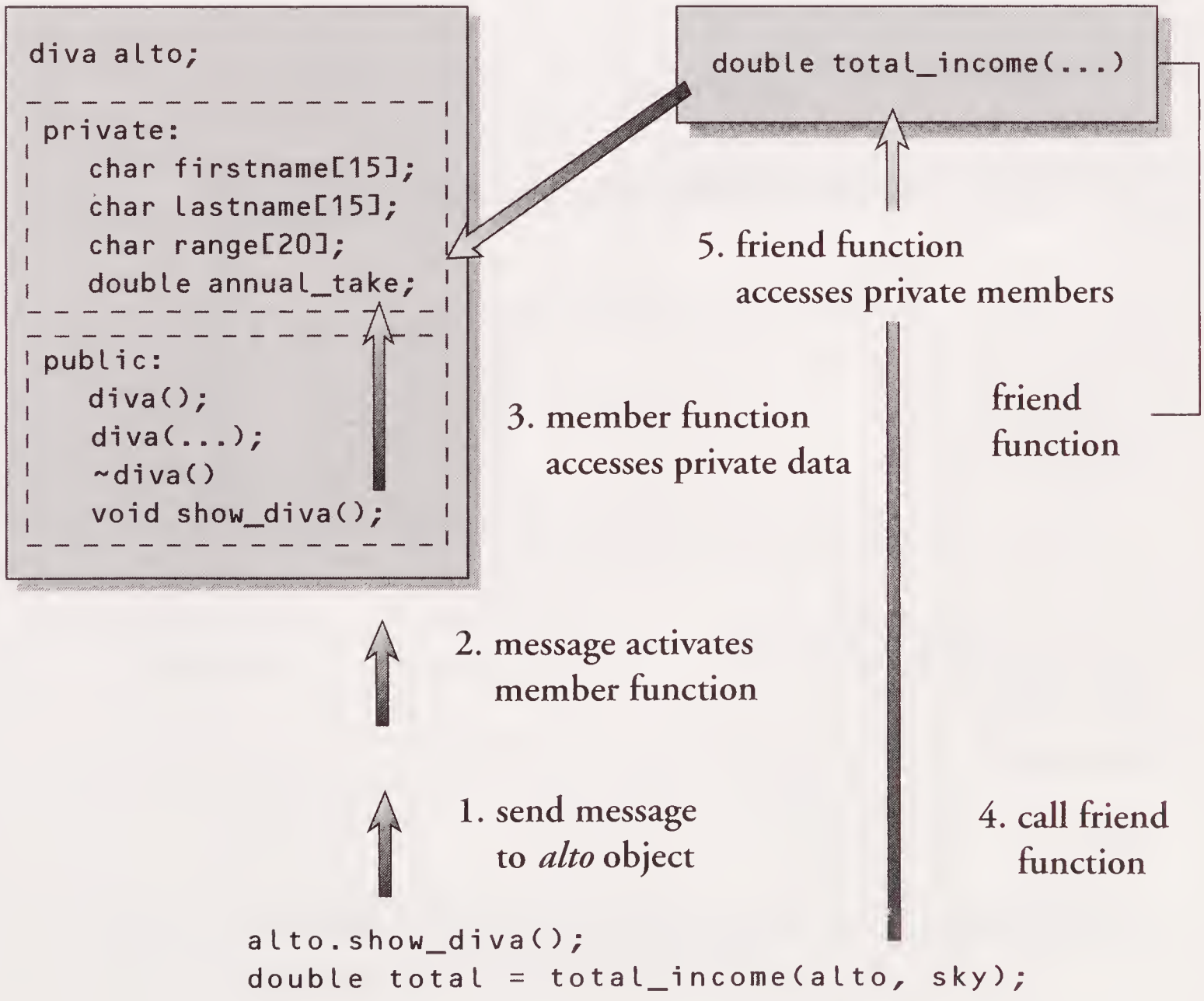


Figure 10-1 *Member function and friend function*

First, let's compare how a member function establishes which object to use with how a friend function does it. A member function, remember, is invoked by a specific object. For instance, if *maria* is a *diva* object, then `maria.show_diva()` invokes the `show_diva()` member function. Then that member function automatically accesses the *maria* object that invoked it. A friend function, however, can't use an object to invoke the function, because the friend function is not a class member. Instead, you must pass the object to the friend function as an argument. For that reason, `total_income()` has two arguments; one is a reference to a *diva* class object (type *diva* &), and the other is a reference to a *diver* class object (type *diver* &). So if *maria* is a *diva* object and *marty* is a *diver* object, then `total_income(maria, marty)` can access both the *maria* object and the *marty* object. The function signature uses `const` to prevent the `total_income()` function from accidentally changing the original values. Incidentally,

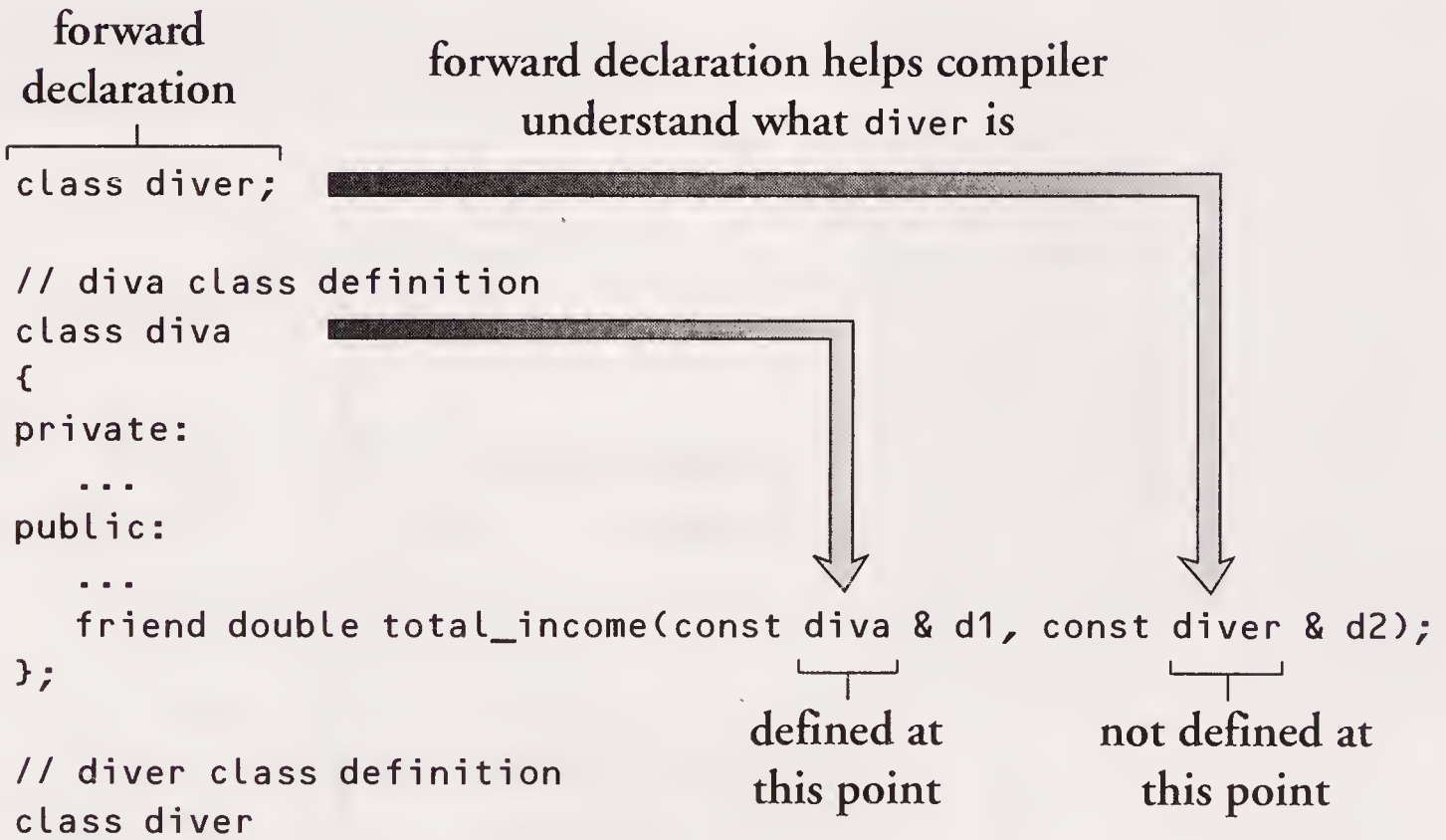
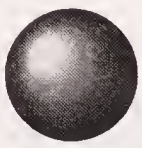


Figure 10-2 *A forward declaration*

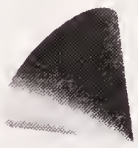
you can pass the objects by value (types *diva* and *diver*) to `total_income()` instead of by reference, but the usual practice is to pass objects by reference to save time and conserve memory. (Passing by reference, recall, allows a function to work with the original variables instead of making copies and working with them.)



A member function operates upon the object used to invoke it, while a friend function operates upon the object or objects passed to it as arguments.

The second difference between member functions and friend functions is how they refer to class members within the function definition. With a member function, a class member is understood to refer to the invoking object. So the `maria.show_diva()` call results in *annual_take* being interpreted as *maria.annual_take*. But a friend function like `total_income()` can't use a member name like *annual_take* by itself. Instead, the friend function must explicitly use an object name and a membership operator with each member name. For instance, the `total_income()` friend function will use *d1.annual_take* and *d2.annual_take* to identify the proper data members for the *diva* class object and the *diver* class object passed as arguments.

Now that we have the class definitions, let's take the usual second step in specifying classes and implement the methods and the friend function. Listing 10-2 does that.



LISTING 10-2 jobs.cpp

```
// jobs.cpp -- diva and diver methods
#include <iostream.h>
#include <string.h>
#include "jobs.h"

// diva methods
diva::diva()                        // default constructor
{
}

diva::diva(char * fn, char * ln, char * rng,
           double inc)              // initializing constructor
{
    strcpy(firstname, fn);
    strcpy(lastname, ln);
    strcpy(range, rng);
    annual_take = inc;
}
```

```
diva::~diva(void)                    // destructor
{
}

void diva::show_diva()
{
    cout << "The fabulous " << range << " " << firstname;
    cout << " " << lastname << " makes $" << annual_take;
    cout << " per annum.\n";
}

// diver methods
diver::diver()
{
}

diver::diver(char * fn, char * ln, double md,
             double inc)
{
    strcpy(firstname, fn);
    strcpy(lastname, ln);
    max_depth = md;
    annual_take = inc;
}

diver::~diver(void)
{
}

void diver::show_diver()
{
    cout << "The reliable " << firstname << " " << lastname;
    cout << " makes $" << annual_take << " per annum\nand can";
    cout << " dive to a depth of " << max_depth << " feet.\n";
}

// friend function to diva and diver

double total_income(const diva & d1, const diver & d2)
{
    return d1.annual_take + d2.annual_take;
}
```

You should note two things about the friend function definition. First, the definition for `total_income()` doesn't use the keyword `friend`. That word is used only for the prototype appearing in a class using the function. Second, because this friend func-

tion belongs to no class, the function declaration doesn't use the scope operator (`::`). The member functions have class scope (discussed in Chapter 9), meaning you can use the same function names in other class definitions, but the friend function has file scope (see Chapter 8), meaning you can have only one friend function with a particular name and signature per file.

When creating a friend function, use the keyword `friend` in the prototype found in the class definition, but don't use the keyword `friend` in the actual function definition, which should be placed outside the class definition.

Now let's try these definitions in a short program. Listing 10-3 creates and initializes a *diva* object and a *diver* object, uses member functions to display the information, then uses the `total_income()` friend function to calculate the family income. Note that `total_income()` takes both `wife` and `husband` as arguments and uses data from both objects to find the total income. The program follows the usual C++ pattern for organization: a header file containing the class definition (Listing 10-1), a methods file containing the member and friend function definitions (Listing 10-2), and a program that uses the class (Listing 10-3). You should use whatever technique your implementation requires for a multiple-file program. Chapters 1 and 8 provided some general guidelines.

LISTING 10-3 friends.cpp

```
// friends.cpp -- two classes using a friend function
// compile this along with jobs.cpp
#include <iostream.h>
#include "jobs.h"
int main(void)
{
    diva wife("Godiva", "Jones", "soprano", 48500.0);
    diver husband("Sealy", "Leadbelly", 150, 24250.0);

    wife.show_diva();                    // member function
    husband.show_diver();                // member function

    cout << "The family income is $";
    // use friend function to access two classes
    cout << total_income(wife, husband) << ".\n";
    return 0;
}
```

Here is the program's output:

```
The fabulous soprano Godiva Jones makes $48500 per annum.
The reliable Sealy Leadbelly makes $24250 per annum
and can dive to a depth of 150 feet.
The family income is $72750.
```

By the way, a function doesn't have to be a member function or a friend function in order to use an object as an argument. For instance, this is a valid function definition:

```
void report(diva & d1, diver & d2)
{
    d1.show_diva();
    d2.show_diver();
}
```

Only if the function has to access `private` members directly does it have to be a member function or a friend. The `report()` function only accesses `public` member functions directly, so it needs no special privileges. The `total_income()` function, however, accessed private data members, so it had to be a friend.

In some cases, such as the *diva-diver* example, you can avoid using friend functions by designing classes differently and by using inheritance (see Chapter 11). But friends are very handy for operator overloading, so you'll be encountering more friends as we explore that topic later in this chapter.

Other Friends

In addition to regular friend functions, you can have friend functions that are members of other classes. You can declare such friends individually, or, by declaring an entire class as a friend, you make all the member functions of that class friend functions. We won't be using these friend varieties in this primer, but the following partial listing shows how to declare a friend class and friend member function from another class:

```
class lead_guitar;       // lead_guitar class forward declaration
class rhythm_guitar;     // rhythm_guitar class forward declaration

class drummer
{
private:
    char fullname[50];
    ...
public:
    drummer();
    ~drummer();
    ...
```

```
// friend function from  lead_guitar class -- use scope operator
    friend char * lead_guitar::riffs(drummer & dr);

// friend class
    friend class rhythm_guitar;
}
```

As the listing shows, to declare as friend a function from another class, you use the class name and the scope resolution operator to identify the function. Because `riffs()` is a member of the *lead_guitar* class, it doesn't need a *lead_guitar* argument, for it will be invoked by a *lead_guitar* object. For instance, if *george* is a *lead_guitar* object and if *dan* is a *drummer* object, you could use code like the following:

```
cout << george.riffs(dan);
```

Prefixing a class declaration with `friend` makes all the member functions of that class into friend functions. Thus any member function from the *rhythm_guitar* class can access `private` members of the *drummer* class.

Operator Overloading

Now let's look at a technique for giving object operations a prettier look. *Operator overloading* is another example of C++ polymorphism. In Chapter 8 you saw how C++ enables you to define several functions having the same name as long as they have different signatures (argument lists). That was function overloading, or functional polymorphism. Its purpose is to let you use the same function name for the same basic operation even though you are applying the operation to different data types. (Imagine how awkward English would be if you had to use a different verb form for each different type of object—lift_lft your left foot, but lift_sp your spoon.) Operator overloading extends the overloading concept to operators, letting you assign multiple meanings to C++ operators. Actually, many C++ (and C) operators already are overloaded. For instance, the * operator, when applied to an address, yields the value stored at that address. But applying * to two numbers yields the product of the values. C++ uses the number and type of operands to decide which action to take.

C++ lets you extend operator overloading to user-defined types, permitting you, say, to use the + symbol to add two objects. Again, the compiler will use the number and type of operands to determine which definition of addition to use. Overloaded operators often can make code look more natural. For instance, a common computing task is adding two arrays. Usually this winds up looking like the following `for` loop:

```
for (int i = 0; i < 20; i++)
    evening[i] = sam[i] + janet[i];    // add element by element
```

But in C++ you can define a class that represents arrays and that overloads the + operator so that you can do this:

```
evening = sam + janet;                    // add two array objects
```

We'll do just that in Chapter 11. (Why not now? Because you also have to overload the `[]` operator, and that's a bit more involved than overloading the + operator.) This simple addition notation conceals the mechanics and emphasizes what is essential, and that is another OOP goal.

To overload an operator, you use a special function form called an *operator function*. An operator function has the form:

```
operatorop(argument-list)
```

where *op* is the symbol for the operator being overloaded. The *op* has to be a valid C++ operator; you can't just make up a new symbol. For instance, you can't have an `operator@()` function because C++ has no `@` operator. But the `operator+()` function would overload the + operator. Suppose, for example, that you have a *salesperson* class for which you define an `operator+()` member function to overload the + operator so that it adds sales figures of one *salesperson* object to another. Then, if *district2*, *sid*, and *chuck* are objects of the *salesperson* class, you would be able to write this equation:

```
district2 = sid + chuck;
```

The compiler, recognizing the operands as belonging to the *salesperson* class, will replace the operator with the corresponding operator function:

```
district2 = sid.operator+(chuck);
```

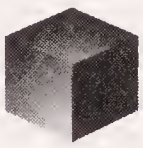
The function will then use the *sid* object implicitly (because it invoked the method) and the *chuck* object explicitly (because it's passed as an argument) to calculate the sum, which it then returns. Of course, the nice part is that you can use the nifty + operator notation instead of the clunky function notation.

C++ imposes some restrictions on operator overloading, but they're easier to understand after you've seen how overloading works. So let's develop a few examples first to clarify the process, then discuss the limitations.

A VECTOR CLASS WITH OPERATOR OVERLOADING

To show how overloading is useful in real applications, we'll develop an example based on vectors. Not only will the example present operator overloading, it will also illustrate further aspects of class design, such as incorporating two different ways of describing the same thing into an object. Even if you don't care for vectors, you'll be able to use many

of the new techniques in other contexts. A vector, as the term is used in engineering and physics, is a quantity having both a magnitude (size) and a direction. For instance, if you push something, the effect depends on how hard you push (the magnitude) and in what direction you push. A push in one direction could save a tottering vase, while a push in the opposite direction could hasten its rush to doom. To fully describe the motion of your car, you should give both the speed (the magnitude) and the direction; arguing with the highway patrol that you were driving under the speed limit carries little weight if you were traveling in the wrong direction. (Immunologists and computer scientists may use the term vector differently; ignore them.) The discussion on Vectors will tell you more about vectors, but understanding them completely isn't necessary for following the C++ aspects of the examples.



VECTORS

You're a worker bee that's discovered a marvelous nectar cache. You rush back to the hive and announce you've found nectar 120 yards away. "Not enough information," buzz the other bees. "You have to tell us the direction, too!" You answer, "It's 30 degrees north of the sun direction." Knowing both the distance (magnitude) and the direction, the other bees rush to the sweet site. Bees know vectors.

Many quantities involve both a magnitude and a direction. The effect of a push, for instance, depends on both its strength and direction. Moving an object on a computer screen involves a distance and a direction. You can describe such things using vectors. For instance, you can describe moving (displacing) an object on the screen with a vector, which you can visualize as an arrow drawn from the starting position to the final position. The length of the vector is its magnitude, and that describes how far the point has been displaced. The orientation of the arrow describes the direction. See Figure 10-3. A vector representing such a change in position is called a *displacement vector.*

Now you're Lhanappa, the great mammoth hunter. Scouts report a mammoth herd 14.1 kilometers to the northwest. But, because of a southeast wind, you don't want to approach from the southeast. So you go 10 kilometers west, then 10 kilometers north, approaching the herd from the south. You know these two displacement vectors bring you to the same location as the single 14.1-kilometer vector pointing northwest. Lhanappa, the great mammoth hunter, also knows how to add vectors.

Adding two vectors has a simple geometric interpretation. First, draw one vector. Then draw the second vector starting from the arrow-end of the first vector. Finally, draw a vector from the starting point of the first vector to the

end point of the second vector. This third vector represents the sum of the first two. See Figure 10-4. Note that the length of the sum can be less than the sum of the individual lengths.

Vectors make a natural choice for operator overloading. First, you can't represent a vector with a single number, so it makes sense to create a class to represent vectors. Second, vectors have analogs to ordinary arithmetic operations such as addition and subtraction. This parallel suggests overloading the corresponding operators so you can use them with vectors.

To keep things simple, we'll implement a two-dimensional vector, such as a screen displacement, instead of a three-dimensional vector, such as might represent movement of a helicopter or a gymnast. You need just two numbers to describe a two-dimensional vector, but you have a choice of what set of two numbers:

- You can describe a vector by its magnitude (length) and direction (an angle).
- You can represent the vector by its *x* and *y components.*

The components are a horizontal vector (the *x* component) and a vertical vector (the *y* component) which add up to the final vector. For instance, you can describe a

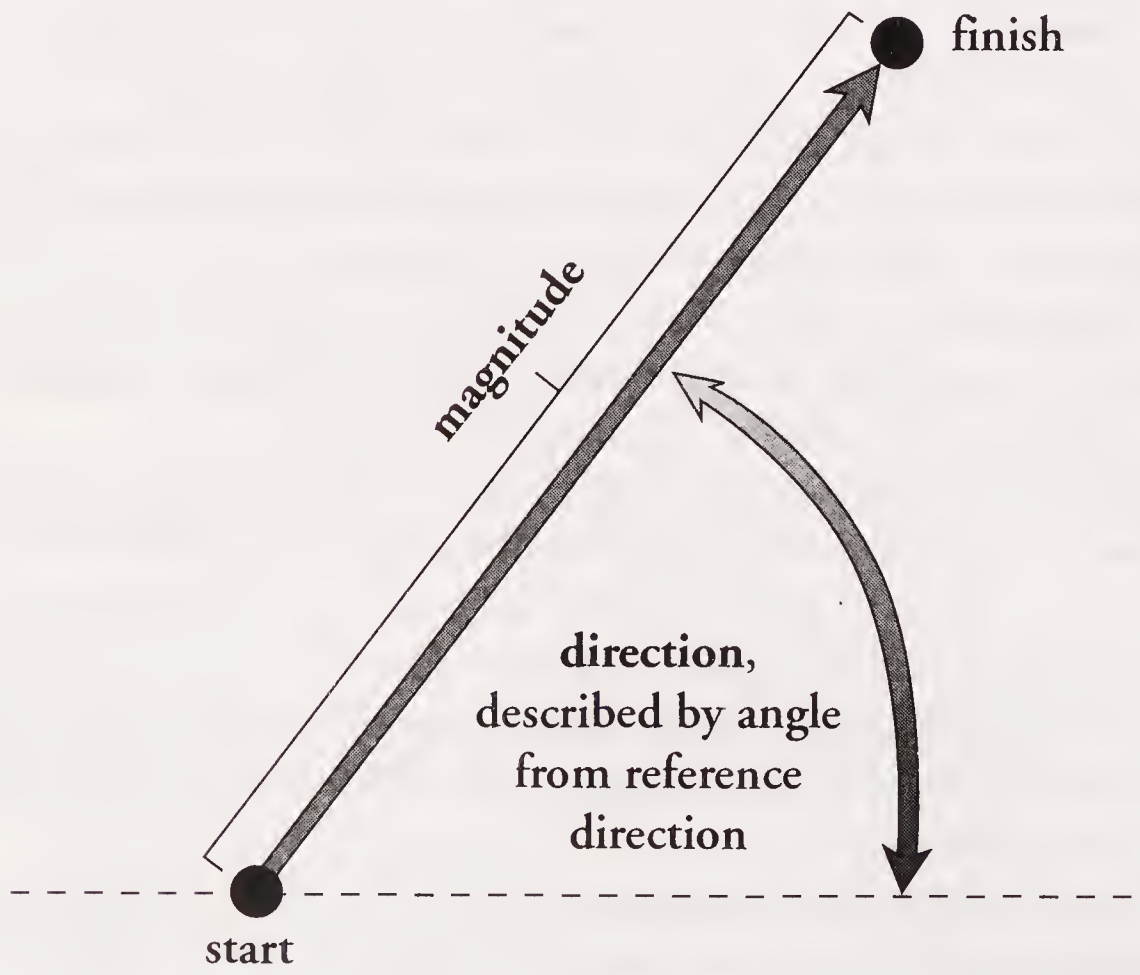


Figure 10-3 *Describing a displacement with a vector*

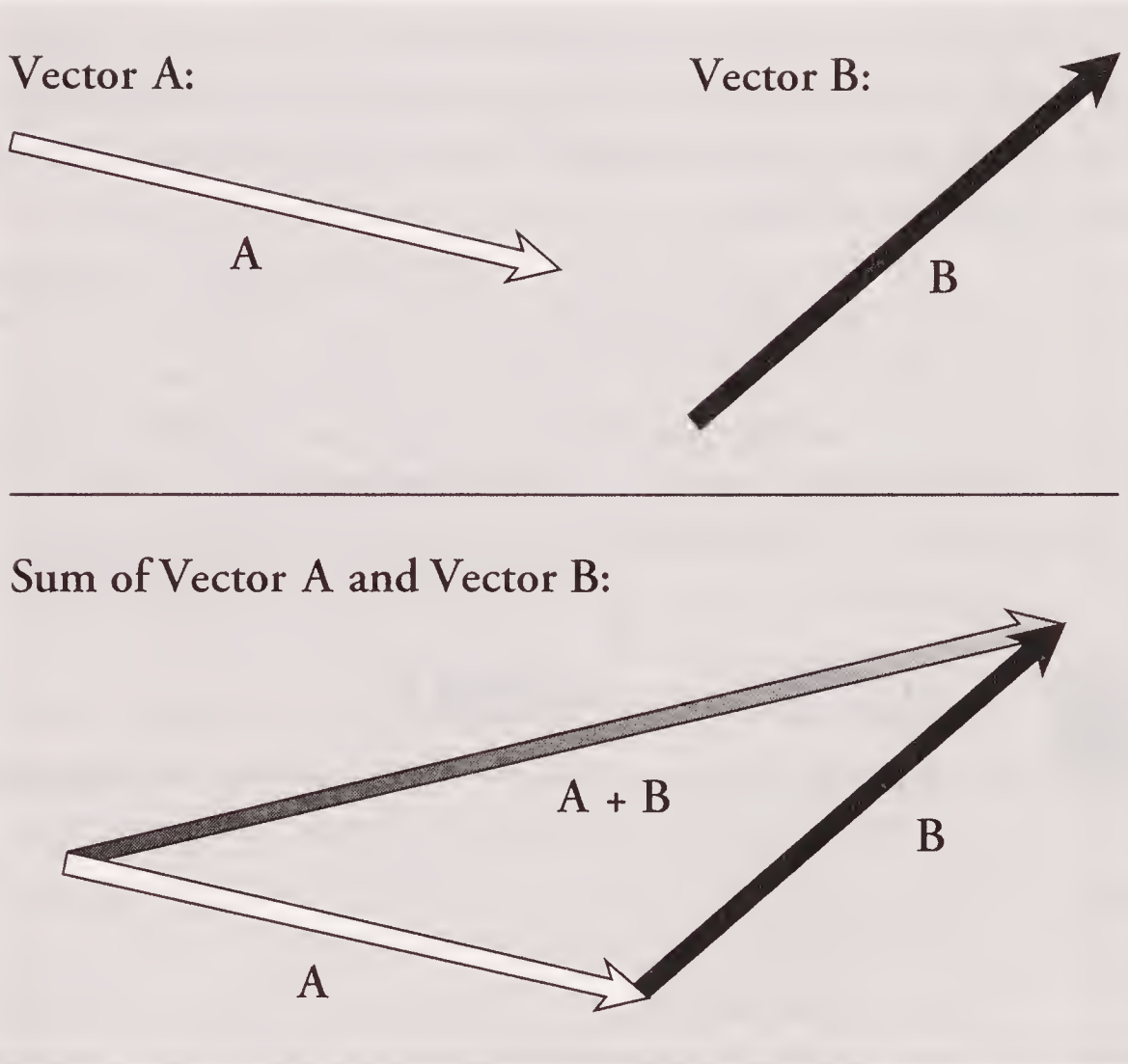


Figure 10-4 *Adding two vectors*

motion as moving a point 30 units to the right and 40 units up. See Figure 10-5. That motion puts the point at the same spot as moving 50 units at an angle of 53.1° from the horizontal. Therefore a vector with a magnitude of 50 and an angle of 53.1° is equivalent to a vector having a horizontal component of 30 and a vertical component of

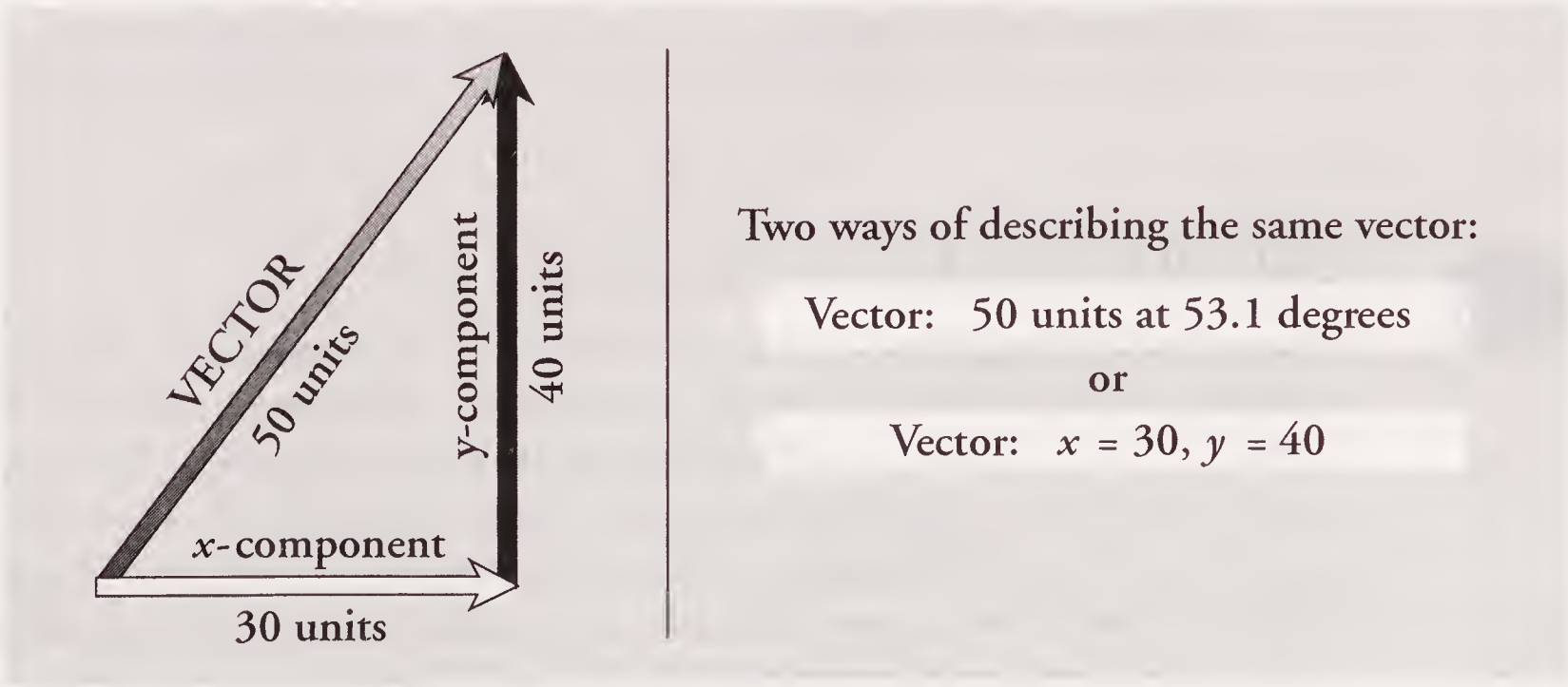
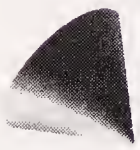


Figure 10-5 *x and y components of a vector*

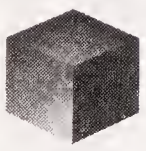
40. What counts with displacement vectors is where you start and where you end up, not the exact route taken to get there. This choice of representation is basically the same thing we covered with the Chapter 7 program that converted between rectangular and polar coordinates.

Sometimes one form is more convenient, sometimes the other, so we'll incorporate both representations into the class description. See the discussion on multiple representations. Also, we'll design the class so that if you alter one representation of a vector, the object automatically updates the other representation. The ability to build such intelligence into an object is another C++ class virtue. Listing 10-4 presents a preliminary class definition that doesn't yet include operator overloading.



LISTING 10-4 vector0.h

```
// vector0.h -- vector class before operator overloading
class vector
{
private:
    double x;                                   // horizontal value
    double y;                                   // vertical value
    double mag;                                 // length of vector
    double ang;                                 // direction of vector
    void set_mag(void);
    void set_ang(void);
public:
    vector(void);
    vector(double h, double v);                 // set x, y values
   ~vector(void);
    void set_by_polar(double m, double a);
    void show_polar(void);                      // show polar values
    void show_vector(void);                     // show rectangular values
};
```



MULTIPLE REPRESENTATIONS AND CLASSES

Quantities having different, but equivalent, representations are common. For example, you can measure gasoline consumption in miles per gallon, as they do in the U.S., or in liters per 100 kilometers, as they do in Europe. You can represent a number in string form or numeric form, and you can represent intelligence as an IQ or in kiloturkeys. Classes lend themselves nicely to encompassing different aspects and representations of an entity in a single object. First, you can store multiple representations in one object. Second, you can write the class functions so that assigning values for one representation automatically assigns

values for the other representation(s). For instance, the `set_by_polar()` method for the *vector* class sets the magnitude and angle members to the function arguments, but it also sets the *x* and *y* members. By handling conversions internally, a class can help you think of a quantity in terms of its essential nature rather than in terms of its representation.

This particular class definition emphasizes the rectangular coordinate approach, with a class constructor that takes arguments representing the horizontal and vertical values. The constructor implementation will call the private functions `set_mag()` and `set_ang()` to set the corresponding magnitude and angle values. However, if you want to specify a vector by its polar coordinates (magnitude and angle), you can use the `set_by_polar()` method, magnitude and angle as well as *x* and *y* horizontal and vertical values and the definition provides methods for displaying both vector representations.

You may have wondered why we didn't provide a third constructor, one that initializes a vector using the magnitude and angle values. The problem is that magnitude and angle require the same function signature (`double, double`) as *x* and *y*, but C++ requires distinct function signatures for overloading. So we have a choice of providing a constructor for one or the other representation, but not both. Later, we'll refine this example and show a way around that problem.

Overloading the + Operator

Meanwhile, let's add operator overloading. Adding two vectors is simple in rectangular coordinates. Just add the two *x* components together to get the new *x* component and add the two *y* components together to get the new *y* component. Here is a sample prototype for a class operator overloading method (it goes into the class definition) and its implementation (it goes into the class member function file):

```
vector operator+(const vector & b);          // prototype
vector vector::operator+(const vector & b)   // definition
{
    double sx, sy;
    sx = x + b.x;                  // x component of sum
    sy = y + b.y;                  // y component of sum
    vector sum = vector(sx, sy);
    return sum;
}
```

The prototype defines a function named `operator+()`. The function returns a *vector* and it takes a *vector* argument. To save time and memory, we pass the vector by reference (type *vector &*), making *b* an alias for the vector passed to the function. Because the

function should not alter the value of the vectors it is adding, we declare the argument as a `const` type. Because `operator+()` is a class method, it uses the invoking object implicitly. For instance, suppose `move1` and `move2` are two vectors and that we use the `operator+()` function to add them:

```
vector q = move1.operator+(move2);
```

The `operator+()` function definition contains the following statement:

```
sx = x + b.x;                    // x component of sum
```

In this statement, the term `b.x` refers to the object passed as an argument (`move2`, in this case), and `x` refers to the object invoking the member function (`move1`, in this case), making the statement equivalent to the following:

```
sx = move1.x + move2.x;          // x component of sum
```

After calculating the *x* and *y* components of the sum, the method uses the `vector(sx,sy)` constructor to initialize the vector `sum` to the proper values:

```
vector sum = vector(sx, sy);
```

As you'll see soon, this constructor also sets the `mag` and `ang` members. Thus `sum` is a vector object having both its rectangular representation and its polar representation set to the correct values. Then the `operator+()` function returns the `sum` vector to the calling program.

So far, `operator+()` looks like a regular class member function with a funny name. Here's where C++ gets cute. As we indicated earlier, C++ offers an alternative syntax for the function call that makes it look as if you were using an operator. The alternative syntax for

```
vector q = move1.operator+(move2);  // function call syntax
```

is this:

```
vector q = move1 + move2;           // alternative syntax
```

Both mean the same thing, but the second form is easier on the eyes and on the mind, and it is the one we'll use.

Let's take one more look at the planning steps in overloading the `+` operator for the *vector* class:

- Create a member function named `operator+()`.
- Addition requires two operands. Invoking the function with an object provides one operand, so use an argument to provide the second operand. This operand can be type *vector*, or, in the interests of efficiency, type `const vector &`.

- The sum of two *vector* objects is also a *vector* object, so give the function a *vector* return type.
- Put the following prototype in the class definition:

```
vector operator+(const vector & b);          // prototype
```

- Use the following function heading in the methods definition file:

```
vector vector::operator+(const vector & b) // definition
```

- Write code to handle the intended operation.

At this point, if you're programming a Bobo the Space Barbarian game, you can write code like the following:

```
#include <iostream.h>
#include "vector.h"                                  // class definition
int main (void)
{
    ...
    vector movebobo1(20,30);
    vector movebobo2(50, 10);
    vector bobo = movebobo1 + movebobo2;             // add 2 vectors
    bobo.showpolar();                                // display result
    ...
}
```

But, having gone this far, it's worth the extra effort to add a few more features to the class definition. In particular, we'll investigate how to overload an overloaded operator and when friends are useful for overloading.

More Refinement: Overloading an Overloaded Operator

Let's start by overloading the - operator. In ordinary C++, this operator already has two meanings. First, when used with two operands, it's the subtraction operator. The subtraction operator is termed a *binary* operator because it has exactly two operands. Second, when used with one operand, as in `-x`, it's a minus sign. This form is termed a *unary* operator, meaning it has exactly one operand. Both operations—subtraction and sign reversal—make sense for vectors, too, so we'll extend them to the *vector* class, beginning with subtraction. To subtract vector B from vector A, you just subtract components, so the definition for overloading subtraction is quite similar to the one for addition:

```
vector operator-(const vector & b);          // prototype
vector vector::operator-(const vector & b)   // definition
{
```

```
    double dx, dy;
    dx = x - b.x;
    dy = y - b.y;
    vector diff = vector(dx, dy);
    return diff;
}
```

Next, let's overload the unary minus operator, which takes just one operand. Applying this operator to a regular number, as in `-x`, changes the sign of the value. We'll want to define it so that applying it to a vector reverses the sign of each component. (In polar terms, negation leaves the magnitude unchanged, but reverses the direction. Many politicians with little or no mathematical training nonetheless have an intuitive mastery of this operation.) Here are the prototype and definition for overloading negation:

```
vector operator-();
vector vector::operator-()
{
    double nx, ny;
    nx = -x;
    ny = -y;
    vector neg = vector(nx, ny);
    return neg;
}
```

Note that now there are two separate definitions for `operator-()`. That's fine, for the two definitions have different signatures. You can define both binary and unary versions of the - operator because C++ provides both binary and unary versions of that operator to begin with. An operator having only a binary form, such as division (`/`), can only be overloaded as a binary operator.



Because operator overloading is implemented with functions, you can overload the same operator many times as long as each operator function has a distinct signature and as long as each operator function has the same number of operands as the corresponding built-in C++ operator.

As with overloading functions, you should use sensible restraint in overloading operators. For instance, don't overload the * operator so that it swaps the components of a *vector* object. Nothing in the notation would suggest what the operator did, so it would be better to define a class method with an explanatory name like `swap()`.

Operator Overloading and Friends

Sometimes you need to use friend functions when overloading operators. Multiplying a vector by a real number provides just such a situation. In visual terms, multiplying a vector makes the vector longer or shorter by that factor. So multiplying a vector by 3 produces a vector with three times the length, but still pointed in the same direction. It's easy to translate that image into how the class represents a vector. In polar terms, multiply the magnitude and leave the angle alone. In rectangular terms, you multiply a vector by a number by multiplying its *x* and *y* components separately by the number. That is, if a vector has components of 5 and 12, multiplying by 3 makes the components 15 and 36. Here are the prototype (goes in the class definition) and the implementation (goes in the methods file) for multiplication:

```
vector operator*(double n);              // prototype
vector vector::operator*(double n)       // definition
{
    double mx, my;
    mx = n * x;
    my = n * y;
    vector mult = vector(mx, my);
    return mult;
}
```

This code works fine for multiplying a vector times a number, but it fails for multiplying a number times a vector! To see this, first consider multiplying a vector times a number:

```
vector borge = piano * 2.0;                  // valid code
```

This translates to the following function call:

```
vector borge = piano.operator*(2.0);
```

That statement requires that `operator*()` be a member function of the *vector* class, for it is invoked by a *vector* object. It also requires that the function take a type `double` argument. These two requirements match the function just defined, so the program automatically uses that definition.

But suppose you reverse the order of the operands while writing a program:

```
vector borge = 2.0 * piano;                  // not supported
```

That would correspond to a meaningless function call:

```
vector borge = 2.0.operator*(piano);         // nonsense
```

That is, this demands a function that is a member of the `double` class (which doesn't even exist) and that takes a type *vector* argument. But our definition of multiplication requires that a *vector* object (*piano*, in this case) be the *first* operand and the `double` value (*2.0*) be the *second* operand.

One way around this difficulty is to tell everyone (and to remember yourself) that you can only write `piano * 2.0` but never write `2.0 * piano`. This is a programmer-friendly, user-beware solution, and that's not what OOP is about.

Can you write a second member function to handle this reverse-order problem? No, you can't, because the invoking object is always the *first* operand; the `piano` object *must* come before the `2.0` if you use a member function. But, recall, you don't use an object to invoke a friend function; instead, you pass the object as an argument. Thus you can use a friend function with the number as the *first* operand and the vector as the *second* argument:

```
friend vector operator*(double n, vector & a);
```

The simplest way to implement this friend function is to have it reverse the order of its arguments and then call the original function:

```
vector operator*(double n, vector & a)
{
    return a * n;          // calls original operator function
}
```

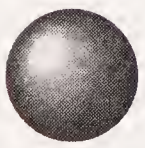
Thus the friend function translates

```
vector borge = 2.0 * piano;              // now supported
```

to the following:

```
vector borge = piano * 2.0;
```

Using a friend function to reverse the operand order is a common and useful technique for overloading an operator that takes two different types of operands. Oh, yes, keep in mind that you use the keyword `friend` with the prototype in the class definition but not when actually defining the function.



If you want to overload an operator for a class and you want to use the operator with a nonclass term as the first operand, you have to use a friend function to reverse the operand order.

By the way, we didn't have this kind of problem when defining the `operator+()` member function because, in that case, *both* arguments were type *vector*, so either could come first.

Friends versus Class Member Functions

Incidentally, you can define all the operator functions as friend functions. In that case, each binary operator function would require two *vector* arguments, and the unary `operator-()` function would take a single *vector* argument. (A friend function always requires one more argument than the equivalent member function; the extra argument explicitly represents the object that a member function uses implicitly.) For instance, you can define subtraction this way:

```
// prototype
friend vector operator-(const vector & a, const vector & b);
// definition
vector operator-(const vector & a, const vector & b)
{
    double dx, dy;
    dx = a.x - b.x;
    dy = a.y - b.y;
    vector diff = vector(dx, dy);
    return diff;
}
```

Calling this version would look different from calling the first version if you used function notation:

```
a = b.operator-(c);      // - defined as class method
a = operator-(b, c);     // - defined as friend function
```

But either definition would let you use the same operator notation:

```
a = b - c;               // permitted by either definition
```

Defining the vector Class: Nearly the Final Version

Now that we have the operator functions ready, we can put together the full *vector* class definition. Listing 10-5 presents the complete definition for the *vector* class.

LISTING 10-5 vector1.h

```
// vector1.h -- vector class, introduce operator overloading
class vector
{
private:
    double x;                    // horizontal value
    double y;                    // vertical value
    double mag;                  // length of vector
    double ang;                  // direction of vector
    void set_mag(void);
```

```
    void set_ang(void);
public:
    vector(void);
    vector(double h, double v);
    ~vector(void);
    void set_by_polar(double m, double a);
    void show_polar(void);              // show polar values
    void show_vector(void);             // show rectilinear values
// operator overloading
    vector operator+(const vector & b);
    vector operator-(const vector & b);
    vector operator-();
    vector operator*(double n);
// friend function
    friend vector operator*(double n, vector & a);
};
```

Listing 10-6 defines the class member functions declared in Listing 10-5. Note how the constructor functions and the `set_by_polar()` function each set both the rectangular and the polar representations of the vector. Thus either set of values is available immediately without further calculation should you need them. Also, as mentioned in Chapters 4 and 7, C++'s built-in math functions use angles in radians, so we've built conversion to and from degrees into the methods. The implementation hides such things as converting from polar coordinates to rectangular coordinates or converting radians to degrees from the user. All the user needs to know is that the class uses angles in degrees and that it makes a vector available in two equivalent representations.

These design decisions follow the OOP tradition of having the class interface concentrate on the essentials while hiding the details. Thus when you use the *vector* class, you can think about a vector's general features, such as that they can represent displacements and that you can add two vectors. Whether you express a vector in component notation or in magnitude, direction notation becomes secondary, for you can set a vector's values and display them in whichever format is most convenient at the time.

By the way, these definitions use math functions, and some C++ systems don't automatically search the math library. For instance, some UNIX systems require that you do the following:

```
$ CC source_file(s) -lm
```

The *-lm* option instructs the linker to search the math library. So, when you eventually compile programs using the vector class, if you get a message about undefined externals, try the *-lm* option or check to see if your system requires something similar.

LISTING 10-6 vector1.cpp

```
// vector1.cpp -- methods for vector class
#include <iostream.h>
#include <math.h>
#include "vector1.h"
const double Rad_to_deg = 57.2957795130823;

// private methods
void vector::set_mag(void)
{
    mag = sqrt(x * x + y * y);
}

void vector::set_ang(void)
{
    ang = atan2(y, x);
}

// public methods
vector::vector(void)
{
    x = y = mag = ang = 0.0;
}

vector::vector(double h, double v)
{
    x = h;
    y = v;
    set_mag();
    set_ang();
}

vector::~vector(void)
{
}

void vector::set_by_polar(double m, double a)
{
    mag = m;
    ang = a / Rad_to_deg;
    x = m * cos (a);
    y = m * sin (a);
}

void vector::show_polar(void)
{
```

```
    cout << "(" << mag << ", ";
    cout << ang * Rad_to_deg << ")\n";
}

void vector::show_vector(void)
{
    cout << "(" << x << ", " << y << ")\n";
}

// operator overloading
vector vector::operator+(const vector & b)
{
    double sx, sy;
    sx = x + b.x;
    sy = y + b.y;
    vector sum = vector(sx, sy);
    return sum;
}

vector vector::operator-(const vector & b)
{
    double dx, dy;
    dx = x - b.x;
    dy = y - b.y;
    vector diff = vector(dx, dy);
    return diff;
}

vector vector::operator-()
{
    double nx, ny;
    nx = -x;
    ny = -y;
    vector neg = vector(nx, ny);
    return neg;
}

vector vector::operator*(double n)
{
    double mx, my;
    mx = n * x;
    my = n * y;
    vector mult = vector(mx, my);
    return mult;
}

// friend method
```

```
vector operator*(double n, vector & a)
{
    return a * n;
}
```

The next step, of course, is to use these definitions in a program. Listing 10-7 does that using vectors representing movements. It demonstrates using both the rectangular and polar representations, and it shows addition and multiplication. Remember to compile Listing 10-6 along with Listing 10-7.

LISTING 10-7 use_vect.cpp

```
// use_vect.cpp -- use the vector class
// compile with vector1.cpp
#include <iostream.h>
#include "vector1.h"
int main(void)
{
    vector first_move(120, 50);
    vector second_move(50, 120);
    vector result;

    result = first_move + second_move; // adding objects!
    cout << "First move: ";
    first_move.show_vector();          // display first_move
    cout << "Magnitude, angle = ";
    first_move.show_polar();
    cout << "Second move: ";
    second_move.show_vector();
    cout << "Magnitude, angle = ";
    second_move.show_polar();
    cout << "Result: ";
    result.show_vector();
    cout << "Magnitude, angle = ";
    result.show_polar();

    vector twotimes = result * 2.0;    // member function
    cout << "Doubled result: ";
    twotimes.show_vector();

    result = 0.5 * result;             // friend function
    cout << "Halved result: ";
    result.show_vector();

    return 0;
}
```

Here is the program's output:

```
First move: (120, 50)
Magnitude, angle = (130, 22.619865)
Second move: (50, 120)
Magnitude, angle = (130, 67.380135)
Result: (170, 170)
Magnitude, angle = (240.416306, 45)
Doubled result: (340, 340)
Halved result: (85, 85)
```

Program Notes

The `first_move` vector represents moving 120 units to the right and 50 units up. The net result is the same as moving 130 units in a direction 22.62 degrees counterclockwise from the *x*-axis. The first two output lines display these two descriptions of the *first_move* vector. Similarly, the second move of 50 units to the right and 120 units up is the same as a move of 130 units in a direction 67.38 degrees from the *x*-axis. The combined effect of the two moves is moving 170 units to the right (120 + 50) and 170 units up, just as the output for the *result* vector shows. And that is equivalent to moving 240.42 units at an angle of 45 degrees from the *x*-axis, as the sixth line of output shows. Note that you can't simply add the magnitudes of two vectors to get the magnitude of the sum. That's why we used rectangular coordinates rather than polar coordinates to calculate the sum. Finally, the program illustrates that both overloadings of the * function work.

Overloading Restrictions

C++ does impose some limits on user-defined operator overloading as follows:

1. The overloaded operator must have at least one operand that is a user-defined type. This prevents you from overloading operators for the standard types. Thus you can't redefine the minus operator (-) so that it yields the sum of two `double` values instead of their difference. This restriction preserves program sanity, although it may hinder creative accounting.

2. You can't use an operator in a manner that violates the syntax rules for the original operator. For instance, you can't overload the modulus operator (%) so that it can be used with a single operand:

```
int x;
diva shiva;
% x;       // invalid for modulus operator
% shiva;   // invalid for overloaded operator
```

Similarly, you can't alter operator precedence. So if you overload the addition operator to let you add two classes, the new operator has the same precedence as ordinary addition.

3. You can't create new operator symbols. For instance, you can't use the combination ** to denote exponentiation.

4. You cannot overload the following five operators:

sizeof	the sizeof operator
.	membership operator
.*	pointer-to-member operator
::	scope resolution operator
?:	conditional operator

This still leaves all the operators in Table 10-1 available for overloading.

5. Most of the operators in Table 10-1 can be overloaded by using either member or nonmember functions. However, you can use *only* member functions to overload the the following operators:

=	assignment operator
()	function call operator
[]	subscripting operator
->	class member access by pointer operator

Note: We have not covered, nor will we cover, every operator mentioned in the list of restrictions or in Table 10-1. However, Appendix E does summarize those operators not covered in the main body of this text.

OVERLOADING THE << OPERATOR

One of the more useful features of classes is that you can overload the << operator so that you can use it with `cout` to display an object's contents. In some ways, this overloading is a bit trickier than the earlier examples, so we'll develop it in two steps instead of in one.

TABLE 10-1 Operators that can be overloaded

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

Suppose *hector* is a *vector* object. To display *vector* values, we've been using `show_vector()` and `show_polar()`. Wouldn't it be nice, however, if you could do the following?

```
cout << hector;  // make cout recognize vector class?
```

You can, because << is one of the C++ operators that can be overloaded. In fact, it already is heavily overloaded. In its most basic incarnation, the << operator is one of C and C++'s bit manipulation operators; it shifts bits left in a value (see Appendix E). But the *ostream* class overloads the operator, converting it into an output tool. Recall that `cout` is an *ostream* object and that it is smart enough to recognize all the basic C++ types. That's because the *ostream* class definition includes an overloaded `operator<<()` definition for each of the basic types. That is, one definition uses an `int` argument, one uses a `double` argument, and so on. So one way to teach `cout` to recognize a *vector* object is to add a new function operator definition to the *ostream* class definition. But it's not a good idea to alter the `iostream.h` file and mess around with a standard interface. It's better to use the *vector* class definition to teach the *vector* class how to use `cout`.

First Version of Overloading <<

To teach the *vector* class to use `cout`, you'll have to use a friend function. Why? Because a statement like

```
cout << hector;
```

uses two objects, with the *ostream* class object (`cout`) first. If you use a *vector* member function to overload <<, the *vector* object would come first, as it did when we overloaded the *operator with a member function. That means you would have to use the << operator this way:

```
hector << cout;   // if operator<<() were a member function
```

That would be confusing. But by using a friend function, you can overload the operator this way:

```
void operator<<(ostream & os, const vector & v)
{
    os << "(x,y) = (" << v.x << "," << v.y <<")";
}
```

This lets you use

```
cout << hector;
```

to print data in the following format:

```
(x,y) = (120,50)
```

Note that the new `operator << ()` definition uses an *ostream* reference `os` as its first argment. Normally, `os` will refer to the `cout` object, as it does in the expression `cout << hector`. But you could use the operator with other *ostream* objects, in which case *os* would refer to those objects. (What? You don't know of any other *ostream* objects? In Chapter 12, you'll learn how to create new objects to manage output to files, and these objects will be able to use *ostream* methods. You then will be able to use our `operator<<()` definition to write to files as well as to the screen.) Furthermore, the call `cout << hector` should use the `cout` object, not a copy, so the function passes the object as a reference instead of by value. Thus the expression `cout << hector` causes `os` to be an alias for `cout`. The *vector* object can be passed by value or by reference, since either form makes the object values available to the function. Again, passing by reference uses less memory and time than passing by value.

Second Version of Overloading <<

The implementation we just presented has a problem. Statements such as

```
cout << hector;
```

work fine, but the implementation doesn't allow you to combine the redefined << operator with the ones `cout` normally uses:

```
cout << "hector value: " << hector << "\n"; // can't do
```

To understand why this doesn't work and what must be done to make it work, you first need to know a bit more about how `cout` operates. Consider the following statements:

```
int x = 5;
int y = 8;
cout << x << y;
```

C++ reads the output statement from left to right, meaning it is equivalent to the following:

```
(cout << x) << y;
```

The << operator, as defined in `iostream.h`, takes an *ostream* object to its left. Clearly, the expression `cout << x` satisfies that requirement because `cout` is an *ostream* object. But the output statement also requires that the whole expression (`cout << x`) be a type *ostream* object, because that expression is to the left of `<< y`. Therefore, the *ostream* class implements the `operator<<()` function so that it returns an *ostream* object. In particular, it returns the invoking object, `cout` in this case. Thus the expression `cout << x` is itself an *ostream* object, and it can be used to the left of the << operator.

You can take the same approach with the friend function. Just revise the `operator<<()` function so that it returns a reference to an *ostream* object:

```
ostream & operator<<(ostream & os, const vector & v)
{
    os << "(x,y) = (", << v.x << "," << v.y <<")";
    return os;
}
```

Note that the return type is *ostream* &. That means, recall, that the function returns a reference to an *ostream* object. Because a program passes an object reference to the function to begin with, the net effect is that the function's return value is just the object passed to it. That is, the statement

```
cout << hector;
```

becomes the following function call:

```
operator<<(cout, hector);
```

And that call returns the *cout* object. So now the following statement does work:

```
cout << "hector value: " << hector << "\n"; // can do
```

Let's break this into separate steps to see how it works. First,

```
cout << "hector value: "
```

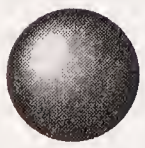
invokes the particular *ostream* definition of << that displays a string and returns the *cout* object, so the expression `cout << "hector value: "` displays the string, then is replaced by its return value, `cout`. This reduces the original statement to the following one:

```
cout << hector << "\n";
```

Next, the program uses the *vector* definition of << to display the *hector* values and to return the *cout* object again. This reduces the statement to the following:

```
cout << "\n";
```

The program now finishes up by using the *ostream* definition of << for strings to display the final string.



In general, to overload the << operator to display an object of class *c_name*, use a friend function with a definition of this form:

```
ostream & operator<<(ostream & os, const c_name & obj)
{
```

```
    os << ... ;  // display object contents
    return os;
}
```

IMPROVING THE CLASS BY ADDING A STATE MEMBER

Once you start playing with class definitions, it can be hard to stop, so let's add one more improvement! Our definition of the << operator displays just the rectangular representation of a vector. Can you overload the definition so that the operator can also display the polar representation? No. Separate definitions require separate function signatures. The operator takes two operands, so the friend operator function takes two arguments, and they have to be an *ostream* object and a *vector* object. So you have no freedom to select another function signature.

But you can change the class definition. The idea is to add a *state member*. This is a variable that affects how the object behaves, sort of like a switch on a printer that lets you set which printer emulation to use. For instance, you can add a type `char` member that can be set to `'p'` or `'r'`. The object will display in polar mode if the state member is `'p'` and in rectangular mode if the state member is `'r'`. (The *ostream* class, incidentally, uses the same state member technique to control the various formatting options discussed in Chapter 12.) Suppose that you call this state member `mode`. Then you can rewrite `operator<<()` this way:

```
ostream & operator<<(ostream & os, const vector & v)
{
    if (v.mode == 'r')
            os << "(x,y) = (" << v.x << ", " << v.y << ")";
    else if (v.mode == 'p')
    {
            os << "(m,a) = (" << v.mag << ", "
               << v.ang * Rad_to_deg << ")";
    }
    else
            os << "Vector object mode is invalid";
    return os;
}
```

The program labels its output with `(x,y)` or `(m,a)` so that you can readily see if the output is in *x*- and *y*- coordinates or in magnitude and angle values. That way, you don't have to check the program code in order to interpret the program output. (Keep that output user friendly!)

To use this approach, you need to alter the *vector* class definitions and rewrite several member functions. First, of course, you have to add the *mode* member to private

class members. Then you have to add methods to set the mode to the desired value. Also, you should make it impossible to set an invalid mode. True, the `operator<<()` definition prints an error message if the mode isn't a `'p'` or an `'r'`, but that's more a debugging tool. If a program generates that error message, you know you should work more on the class design. The most direct way to prevent invalid modes is to modify the constructors so that they always initialize a new object to a correct mode. For instance, you can define the default constructor this way:

```
vector::vector(void)            // default constructor
{
    x = y = mag = ang = 0.0;
    mode = 'r';                 // default mode
}
```

YET ANOTHER TRICK

Remember how the other constructor is biased towards the *x-y* view of vectors? The prohibition on duplicate signatures prevented us from making a constructor using the magnitude-angle view. But now, you can use the new mode member to rewrite the other constructor so that it can initialize a vector either to *x-y* values or to magnitude-angle values. The trick is to add the mode as the third argument to the constructor:

```
vector::vector(double n1, double n2, char form)
{
    mode = form;          // set mode member to form argument
    if (form == 'r')
    {
            x = n1;
            y = n2;
            set_mag();
            set_ang();
    }
    else if (form == 'p')
    {
            mag = n1;
            ang = n2 / Rad_to_deg;
            set_x();      // add to class definition
            set_y();      // add to class definition
    }
    else
    {
            cout << "Incorrect 3rd argument to vector() -- ";
            cout << "vector set to 0\n";
```

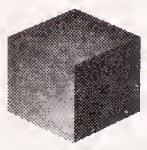
```
            x = y = mag = ang = 0.0;
            mode = 'r';
        }
    }
```

Further, you can use `'r'` as the default value for the third argument. This lets you continue to use the two-argument form to initialize vectors in the *x-y* mode.

Listing 10-8 shows the revised class definition. It adds a `mode` member and private functions for setting the *x* and *y* values from the polar values. It provides a new prototype for one of the constructors. Note that the prototype provides a default value of `'r'` for `mode`. Because the new constructor can set either polar or rectangular values, we drop the old `set_to_polar()` function. Next, the `operator<<()` method replaces the old `show_vector()` and `show_polar()` methods.

LISTING 10-8 vector2.h

```
// vector2.h -- vector class with <<, mode state
class vector
{
private:
    double x;               // horizontal value
    double y;               // vertical value
    double mag;             // length of vector
    double ang;             // direction of vector
    char mode;              // 'r' = rectangular, 'p' = polar
    void set_mag(void);
    void set_ang(void);
    void set_x(void);
    void set_y(void);
public:
    vector(void);
    vector(double n1, double n2, char form = 'r');
    ~vector(void);
    void polar_mode();
    void rect_mode();
// operator overloading
    vector operator+(const vector & b);
    vector operator-(const vector & b);
    vector operator-();
    vector operator*(double n);
// friends
    friend vector operator*(double n, vector & a);
    friend ostream & operator<<(ostream & os, const vector & v);
};
```



FRIEND OR NO FRIEND?

The new *vector* class definition makes the `operator<<()` function a friend function to the *vector* class. But this function, although not inimical to the *ostream* class, is not a friend to it. The `operator<<()` function takes an *ostream* argument and a *vector* argument, so it might seem this function has to be friends to both classes. If you look at the code for the function, however, you'll notice that the function accesses individual members of the *vector* object but only uses the *ostream* object as a whole. Because `operator<<()` accesses `private` *vector* object members directly, it has to be a friend to the `private` *vector* class. But because it does not directly access `private` *ostream* object members, the function does not have to be a friend to the *ostream* class. That's nice, for it means you don't have to tinker with the *ostream* definition.

Listing 10-9 lists the revised methods for the *vector* class. We've already discussed most of these changes individually. Keep in mind that default values appear just in the prototypes in Listing 10-8, not in the methods file.



LISTING 10-9 vector2.cpp

```
// vector2.cpp -- methods for vector class
#include <iostream.h>
#include <math.h>
#include "vector2.h"
const double Rad_to_deg = 57.2957795130823;

// private methods
void vector::set_mag(void)
{
    mag = sqrt(x * x + y * y);
}

void vector::set_ang(void)
{
    ang = atan2(y, x);
}

void vector::set_x(void)
{
    x = mag * cos(ang);
}

void vector::set_y(void)
```

```
{
    y = mag * sin(ang);
}

// public methods
vector::vector(void)            // default constructor
{
    x = y = mag = ang = 0.0;
    mode = 'r';
}

vector::vector(double n1, double n2, char form)
{
    mode = form;
    if (form == 'r')
    {
            x = n1;
            y = n2;
            set_mag();
            set_ang();
    }
    else if (form == 'p')
    {
            mag = n1;
            ang = n2 / Rad_to_deg;
            set_x();
            set_y();
    }
    else
    {
            cout << "Incorrect 3rd argument to vector() -- ";
            cout << "vector set to 0\n";
            x = y = mag = ang = 0.0;
            mode = 'r';
    }
}

vector::~vector(void)
{
}

void vector::polar_mode()
{
    mode = 'p';
}

void vector::rect_mode()
```

```
{
    mode = 'r';
}

// operator overloading
vector vector::operator+(const vector & b)
{
    double sx, sy;
    sx = x + b.x;
    sy = y + b.y;
    vector sum = vector(sx, sy);
    return sum;
}

vector vector::operator-(const vector & b)
{
    double dx, dy;
    dx = x - b.x;
    dy = y - b.y;
    vector diff = vector(dx, dy);
    return diff;
}

vector vector::operator-()
{
    double nx, ny;
    nx = -x;
    ny = -y;
    vector neg = vector(nx, ny);
    return neg;
}

vector vector::operator*(double n)
{
    double mx, my;
    mx = n * x;
    my = n * y;
    vector mult = vector(mx, my);
    return mult;
}

// friend methods
vector operator*(double n, vector & a)
{
    return a * n;
}
```

```
ostream & operator<<(ostream & os, const vector & v)
{
    if (v.mode == 'r')
            os << "(x,y) = (" << v.x << ", " << v.y << ")";
    else if (v.mode == 'p')
    {
            os << "(m,a) = (" << v.mag << ", "
              << v.ang * Rad_to_deg << ")";
    }
    else
            os << "Vector object mode is invalid";
    return os;
}
```

Listing 10-10 provides a short program testing the revised class. (All the work that goes into the class design makes the actual program short and easy to write. Well, that's part of the OOP intent.) Remember to compile Listing 10-9 along with Listing 10-10.

LISTING 10-10 use_cout.cpp

```
// use_cout.cpp -- use the revised vector class
// compile with the vector2.cpp file
#include <iostream.h>
#include "vector2.h"
int main(void)
{
    double x, y, mag, ang;
    cout << "Enter x and y components of first move: ";
    cin >> x >> y;
    vector first_move(x, y);
    cout << "Enter magnitude and angle of second move: ";
    cin >> mag >> ang;
    vector second_move(mag, ang, 'p');
    cout << "First move: ";
    cout << first_move;
    first_move.polar_mode();
    cout << ", " << first_move << "\n";
    cout << "Second move: " << second_move << "\n";
    cout << "Result: " << (first_move + second_move) << "\n";

    return 0;
}
```

Here is a sample run:

```
Enter x and y components of first move: 10 10
Enter magnitude and angle of second move: 20 90
First move: (x,y) = (10, 10), (m,a) = (14.142136, 45)
Second move: (m,a) = (20, 90)
Result: (x,y) = (10, 30)                    ← Default constructor sets 'r' mode
```

Program Notes

The declaration

```
vector first_move(x, y);
```

omits the third argument, so the program uses the default mode of `'r'`, that is, *x* and *y* components. Note that you can use variables to initialize a object. Of course, you should assign values to those variables prior to using them this way. Listing 10-9 uses `cin` to provide the values.

The declaration

```
vector second_move(mag, ang, 'p');
```

explicitly instructs the constructor to use the polar mode. Again, the program uses values obtained from `cin` to initialize the vector.

The statements

```
cout << first_move;
first_move.polar_mode();
cout << ", " << first_move << "\n";
```

use `cout <<` to display the `first_move` vector. Because the constructor earlier set the `first_move` mode to *'r'*, the program displays the *x*-*y* values. Then the program uses the `polar_mode()` function to change the mode. Thus the next time it displays `first_move` with the `<<` operator, the program shows the polar representation.

Notice the final `cout` statement:

```
cout << "Result: " << (first_move + second_move) << "\n";
```

Our definition of `<<` requires a *vector* object operand, but this statement provides the sum of two *vector* objects. This works because *vector* addition uses an operator function that returns a *vector* object, so the sum is replaced by a single object. Furthermore, our implementation of `operator+()` uses a constructor to create the vector that is returned, and the constructor automatically sets the mode to `'r'`. Thus the program displays the sum in rectangular coordinates.

We've used the *vector* class to illustrate operator overloading with member functions and with friend functions. We've used it to illustrate overloading the `<<` operator to display *vector* values, and we've used it to illustrate how a state variable can add

flexibility to a design. This entire sequence of examples shows how a class definition can grow in flexibility and power as you work with it. When developing a new class, it's usually a good idea to proceed as these examples did, starting with a basic implementation, getting it to work, then expanding the class. By now, however, the *vector* class is getting too involved to add new features without risking the new ideas being buried in the clutter. So we'll retire the *vector* class and return to simpler classes as we explore still more class concepts and techniques. Before proceeding to new material, you may wish to pause to take a deep, relaxing breath and reward yourself with a treat, perhaps a cookie, perhaps a good movie, perhaps a short trip to Tahiti.

Automatic Conversions and Type Casts for Classes

The next topic on the class menu is the conversion of another type to a class type. When, in C++, you make a statement assigning a value of one standard type to a variable of another standard type, C++ automatically will convert the value to the same type as the receiving variable, providing the two types are compatible. For instance, the following statements all generate numeric type conversions:

```
long count = 8;      // int value 8 converted to type long
double time = 11;    // int value 11 converted to type double
int side = 3.33;     // double value 3.33 converted to type int 3
```

These assignments work because C++ recognizes that the diverse numeric types all represent the same basic thing, a number, and because C++ incorporates built-in rules for making the conversions. Recall, however (from your reading of Chapter 3), that you can lose some precision in these conversions. For instance, assigning `3.33` to the `int` variable *side* results in *side* getting the value `3`, losing the `0.33` part.

The C++ language does not automatically convert types that are not compatible. For instance, the statement

```
int * p = 10;      // type clash
```

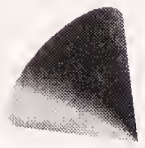
fails because the left-hand side is a pointer-type, while the right-hand side is a number. And even though a computer may represent an address internally with an integer, integers and pointers conceptually are quite different. For instance, you wouldn't square a pointer. However, when automatic conversions fail, you may be able to use a type cast:

```
int * p = (int *) 10;    // ok, p and (int *) 10 both pointers
```

This sets a pointer to the address `10` by type casting `10` to type pointer-to-`int` (that is, type `int *`).

You may define a class sufficiently related to a basic type or to another class that it makes sense to convert from one form to another. In that case, you can instruct C++

how to make such conversions automatically or, perhaps, via a type cast. To show how that works, let's recast the pounds-to-stone program from Chapter 3 into class form. First, design an appropriate type. Fundamentally, we're representing two ways (pounds and stone) to represent one thing (a weight). A class provides an excellent way to incorporate two representations of one concept into a single entity. Therefore, it makes sense to place both representations of weight into the same class and then to provide class methods for expressing the weight in different forms. Listing 10-11 provides the class header.

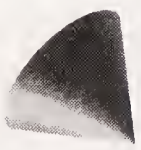


LISTING 10-11 stonewt.h

```
// stonewt.h -- definition for stonewt class
const int Lbs_per_stn = 14;         // pounds per stone
class stonewt
{
private:
    int stone;                      // whole stones
    double pds_left;                // fractional pounds
    double pounds;                  // entire weight in pounds
public:
    stonewt(int lbs);               // constructor for int pounds
    stonewt(double lbs);            // constructor for double pounds
    stonewt(int stn, double lbs);   // constructor for stone, lbs
    stonewt();                      // default constructor
    ~stonewt();
    void show_lbs();                // show weight in pounds format
    void show_stn();                // show weight in stone format
};
```

Note that the class has four constructors. This allows you to initialize a *stonewt* object to an integer number of pounds, to a floating-point number of pounds, or to a stone and pound combination. Or you can create a *stonewt* object without initializing it.

Also, the class provides two display functions. One displays the weight in pounds, and the other displays the weight in stone and pounds. Listing 10-12 shows the class methods implementation. Note that each constructor assigns values to all three private members. Thus creating a *stonewt* object automatically sets both representations of weight.



LISTING 10-12 stonewt.cpp

```
// stonewt.cpp -- stonewt class methods
#include <iostream.h>
#include "stonewt.h"
```

```
stonewt::stonewt(int lbs)
{
    stone = lbs / Lbs_per_stn;          // integer division
    pds_left = lbs % Lbs_per_stn;
    pounds = lbs;
}

stonewt::stonewt(double lbs)
{
    stone = int (lbs) / Lbs_per_stn;    // integer division
    pds_left = int (lbs) % Lbs_per_stn + lbs - int(lbs);
    pounds = lbs;
}

stonewt::stonewt(int stn, double lbs)
{
    stone = stn;
    pds_left = lbs;
    pounds =  stn * Lbs_per_stn +lbs;
}

stonewt::stonewt()
{
    stone = pounds = pds_left = 0;
}

stonewt::~stonewt()
{
}

void stonewt::show_stn()
{
    cout << stone << " stone, " << pds_left << " pounds\n";
}

void stonewt::show_lbs()
{
    cout << pounds << " pounds\n";
}
```

Because a *stonewt* object represents a single weight, it makes sense to provide ways to convert an integer or a floating-point value to a *stonewt* object. And we have already done so! In C++, any constructor taking a *single* argument acts as a blueprint for converting a value of that argument type to the class type. Thus the constructors

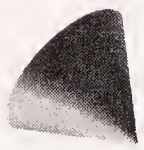
```
stonewt(int lbs);      // template for int-to-stonewt conversion
stonewt(double lbs);   // template for double-to-stonewt conversion
```

serve as instructions for converting type `int` values and type `double` values to a type *stonewt* value. However, the constructor

```
stonewt(int stn, double lbs);
```

has two arguments, so it cannot be used to convert types.

Listing 10-13 uses the class constructors for initializing three *stonewt* objects, then it uses two of them to handle type conversions. Remember to compile Listing 10-12 along with Listing 10-13.



LISTING 10-13 stone.cpp

```
// stone.cpp -- user-defined conversions
// compile with stonewt.cpp
#include <iostream.h>
#include "stonewt.h"

int main(void)
{
    stonewt pavarotti = 260;    // uses constructor to initialize
    stonewt wolfe(285.7);       // same as stonewt wolfe = 285.7;
    stonewt taft(21, 8);

    cout << "The tenor weighed ";
    pavarotti.show_stn();
    cout << "The detective weighed ";
    wolfe.show_stn();
    cout << "The President weighed ";
    taft.show_lbs();
    pavarotti = 265.8;          // uses constructor for conversion
    taft = 325;                 // same as taft = stonewt(325);
    cout << "After dinner, the tenor weighed ";
    pavarotti.show_stn();
    cout << "After dinner, the President weighed ";
    taft.show_lbs();
    cout << "No stone left unearned\n";
    return 0;
}
```

Here is the output:

```
The tenor weighed 18 stone, 8 pounds
The detective weighed 20 stone, 5.7 pounds
```

```
The President weighed 302 pounds
After dinner, the tenor weighed 18 stone, 13.8 pounds
After dinner, the President weighed 325 pounds
No stone left unearned
```

Program Notes

First, note that when a constructor has a single argument, you can use the following form when initializing a class object:

```
// a syntax for initializing a class object when
// using a constructor with one argument
stonewt pavarotti = 260;
```

This is equivalent to the other two forms we've used:

```
// standard syntax forms for initializing class objects
stonewt pavarotti(260);
stonewt pavarotti = stonewt(260);
```

However, the last two forms can also be used with constructors having multiple arguments.

Next, note the following two assignments from Listing 10-13:

```
pavarotti = 265.8;
taft = 325;
```

The first assignment uses the constructor with a type `double` argument to convert `265.8` to a type *stonewt* value. This sets the `pounds` member of `pavarotti` to *265.8*. Because it uses the constructor, this assignment also sets the `stone` and `pds_left` members of the class. Similarly, the second assignment converts a type `int` value to a type *stonewt* value, setting all three member values in the process.

CONVERSION FUNCTIONS

Listing 10-13 converts a number to a *stonewt* object. Can you do the reverse? That is, can you convert a *stonewt* object to a `double` value, as in the following?

```
stonewt wolfe(285.7);
double host = wolfe;     // ?? possible ??
```

The answer is that you can do this, but not by using constructors. Constructors only provide for converting another type *to* the class type. To do the reverse, you have to use a special form of C++ operator function called a *conversion function.*

Conversion functions resemble user-defined type casts, and you can use them the way you would use a type cast. For instance, if you define a *stonewt*-to-`double` conversion function, you can use the following conversions:

```
stonewt wolfe(285.7);
double host = double (wolfe);          // syntax #1
double thinker = (double) wolfe;       // syntax #2
```

Or you can let the compiler figure out what to do:

```
stonewt wells(20, 3);
double star = wells;     // implicit use of conversion function
```

The computer, noting that the right-hand side is type *stonewt* and that the left-hand side is type `double`, looks to see if you've defined a conversion function matching that description.

So how do you create a conversion function? To convert to type *typename*, use a conversion function of this form:

```
operator typename();
```

Note the following points:

- The conversion function must be a class method.
- The conversion function must *not* specify a return type.
- The conversion function must have *no* arguments.

For instance, a function to convert to type `double` would have this prototype:

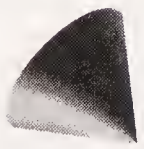
```
operator double();
```

The *typename* part tells the conversion the type to which to convert, so no return type is needed. The fact that the function is a class method means it has to be invoked by a particular class object, and that tells the function which value to convert. Thus the function doesn't need arguments.

To add functions converting *stone_wt* objects to type `int` and to type `double`, then, requires adding the following prototypes to the class definition:

```
operator int();
operator double();
```

Listing 10-14 shows the modified class definition.



LISTING 10-14 stonewt1.h

```
// stonewt1.h -- revised definition for stonewt class
const int Lbs_per_stn = 14;     // pounds per stone
class stonewt
{
private:
    int stone;                  // whole stones
    double pds_left;            // fractional pounds
    double pounds;              // entire weight in pounds
```

```
public:
    stonewt(int lbs);          // construct from int pounds
    stonewt(double lbs);       // construct from double pounds
    stonewt(int stn, double lbs); // construct from stone, lbs
    stonewt();                 // default constructor
    ~stonewt();
    void show_lbs();           // show weight in pounds format
    void show_stn();           // show weight in stone format
// conversion functions
    operator int();
    operator double();
};
```

Next, Listing 10-15 shows the definitions for these two conversion functions; these definitions should be added to the class member function file. Note that each function does return the desired value even though there is no declared return type. Also note the `int` conversion definition rounds to the nearest integer rather than truncating.

LISTING 10-15 stonewt1.cpp

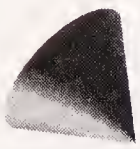
```
// stonewt1.cpp -- stonewt class methods + conversion functions
#include <iostream.h>
#include "stonewt1.h"

// previous definitions go here

// conversion functions
stonewt::operator int()
{
    if (pounds - (int) pounds < 0.5)
            return pounds;
    else
            return pounds + 1;
}

stonewt::operator double()
{
    return pounds;
}
```

Listing 10-16 tests the new conversion functions. The assignment statement in the program uses an implicit conversion, while the final `cout` statement uses an explicit type cast. Remember to compile Listing 10-15 along with Listing 10-16.



LISTING 10-16 stone1.cpp

```
// stone1.cpp -- user-defined conversion functions
// compile with stonewt1.cpp
#include <iostream.h>
#include "stonewt1.h"

int main(void)
{
    stonewt poppins(9,2.8);              // 9 stone, 2.8 pounds
    double p_wt = poppins;               // implicit conversion
    cout << "Convert to double => ";
    cout << "Poppins: " << p_wt << " pounds.\n";
    cout << "Convert to int => ";
    cout << "Poppins: " << int (poppins) << " pounds.\n";
    return 0;
}
```

Here's the program output; it shows the result of converting the type *stonewt* object to type `double` and to type `int`:

```
Convert to double => Poppins: 128.8 pounds.
Convert to int => Poppins: 129 pounds.
```

Applying Type Conversions Automatically

The last example used `int (poppins)` with `cout`. Suppose, instead, it omitted the explicit type cast:

```
cout << "Poppins: " << poppins << " pounds.\n";
```

Would the program use an implicit conversion, as it did in the following statement?

```
double p_wt = poppins;
```

The answer is *no*. In the `p_wt` example, the context indicates that `poppins` should be converted to type `double`. But in the `cout` example, nothing indicates whether the conversion should be to `int` or to `double`. Facing this lack of information, the compiler would complain that you were using an ambiguous conversion. Nothing in the statement indicates what type to use.

Interestingly, if the class had defined only the `double` conversion function, the compiler would accept our errant statement. That's because with only one conversion possible, there is no ambiguity.

You can have a similar situation with assignment. With the current class definitions, the compiler rejects the statement

```
long gone = poppins;   // ambiguous
```

as ambiguous. In C++ you can assign both `int` and `double` values to a `long` variable, so the compiler legitimately could use either conversion function. The compiler doesn't want the responsibility of choosing which. But if you eliminate one of the two conversion functions, the compiler accepts the statement. For example, suppose you omit the `double` definition. Then the compiler will use the `int` conversion to convert `poppins` to a type `int` value. Then it converts the `int` value to type `long` when assigning it to `gone`.

When the class defines two or more conversions, you can still use an explicit type cast to indicate which conversion function to use. You can use either type cast notation:

```
long gone = (double) poppins;   // use double conversion
long gone = int (poppins);      // use int conversion
```

The first statement converts `poppins` weight to a `double` value, then assignment converts the `double` value to type `long`. Similarly, the second statement converts `poppins` first to type `int`, then to `long`.

In summary, then, C++ provides the following type conversions for classes:

- A class constructor that has but a single argument serves as an instruction for converting a value of the argument type to the class type. For example, the *stonewt* class constructor with a type `int` argument is invoked automatically when you assign a type `int` value to a *stonewt* object.
- A special class member operator function called a conversion function serves as an instruction for converting a class object to some other type. The conversion function is a class member, has no declared return type, has no arguments, and is called *operator typename()*, where *typename* is the type to which the object is to be converted. This conversion function is invoked automatically when you assign a class object to a variable of that type or use the type cast operator to that type.

Dynamic Memory and Classes

What would you like for breakfast, lunch, and dinner for the next month? How many ounces of milk for dinner on the third day? How many raisins in your cereal for breakfast on the fifteenth day? If you're like most people, you'd rather postpone some of those decisions until the actual mealtimes. Part of OOP strategy is to take the same attitude towards memory allocation, letting the program decide about memory during run time rather than during compile time. That way, memory use can depend on the needs of a program instead of upon a rigid set of storage-class rules. To gain dynamic control of

memory, you remember, C++ utilizes the `new` and `delete` operators. Unhappily, using these operators with classes can pose some new programming problems. As you'll see, destructors can become necessary instead of merely ornamental. And sometimes you have to overload the assignment operator to get programs to behave properly. We'll look into these matters now.

REVIEW EXAMPLE AND STATIC CLASS MEMBERS

We haven't used `new` and `delete` for a while, so let's review them with a short program. While we're at it, we'll introduce a new storage class: the `static` class member. The vehicle will be a *string* class. The class will hold a pointer to a string and a value representing the string length. We'll use the *string* class primarily to give an inside look at how `new`, `delete`, and `static` class members operate. For that reason, the constructors and destructors will display messages when called so that you can follow the action. Also, we'll omit several useful member and friend functions, such as an overloaded << operator and a conversion function, in order to simplify the class interface. (But rejoice! The review questions for this chapter give you the opportunity to add those useful support functions.) Listing 10-17 shows the class definition. We've called the file `strngs.h` instead of `string.h` to avoid conflicting with the standard library file `string.h`.

LISTING 10-17 strngs.h

```
// strngs.h -- string class definition
class string
{
private:
    char * str;                         // pointer to string
    int len;                            // length of string
    static int num_strings;             // number of objects
public:
    void says();
    string(char * s);                   // constructor
    string();                           // default constructor
    ~string();                          // destructor
};
```

You should note two points about this definition. First, it uses a pointer-to-`char` instead of a `char` array to represent a name. This means that the class definition does not allocate storage space for the string itself. Instead, it will use `new` in the constructors

to allocate space for the string. This arrangement avoids straitjacketing the class definition with a predefined limit to the string size.

Second, the definition declares the `num_strings` member as belonging to the *static* storage class. A static class member has a special property—a program creates only one copy of a static class variable regardless of the number of objects created. That is, a

```
class string
{
private:
    char * str;
    int len;
    static int num_strings;
public:
     ...
};
```

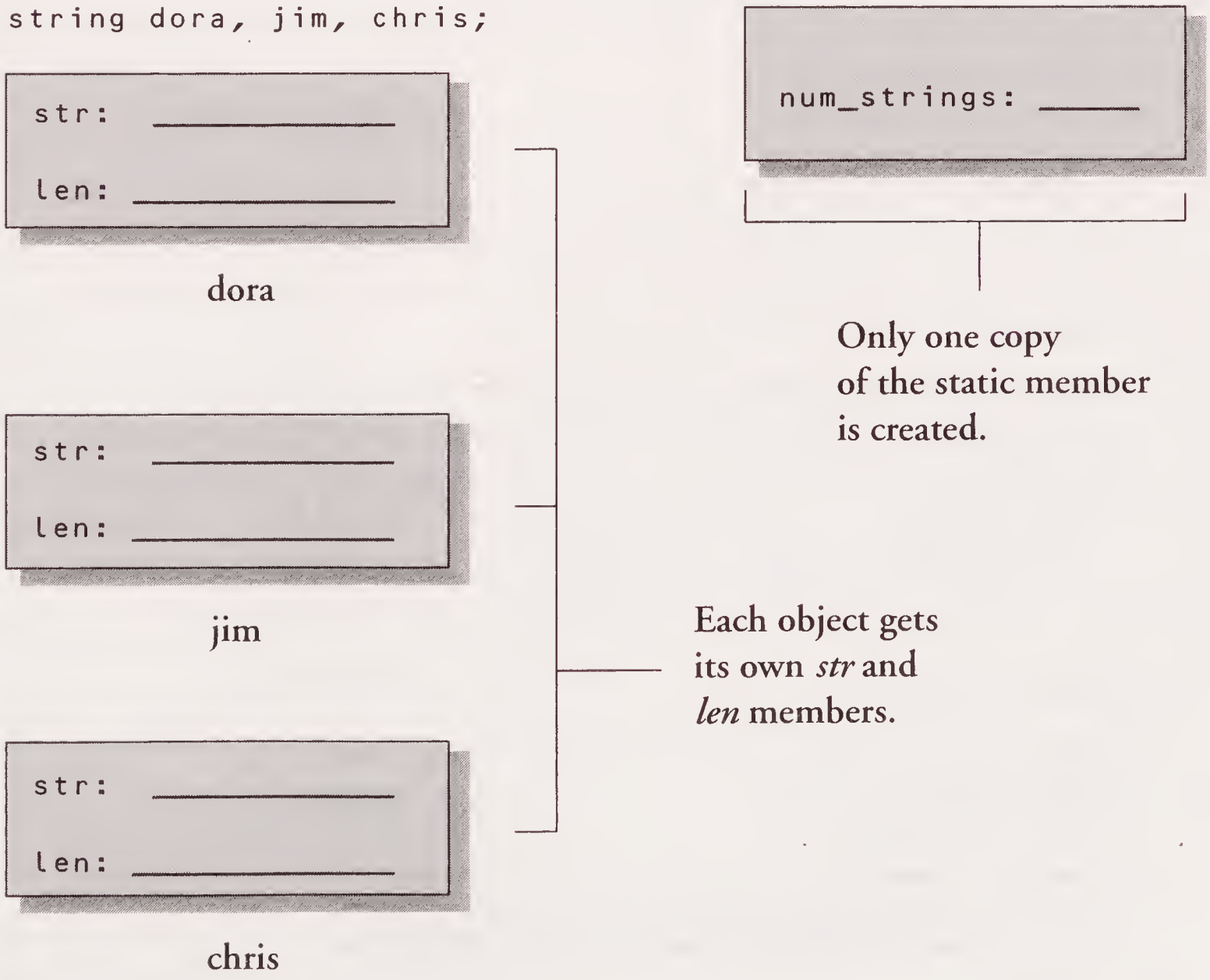


Figure 10-6 *The static class member*

static member is *shared* among all objects of that class, much as a phone number might be shared among all members of a family. If, say, you create ten *string* objects, there would be ten *str* members and ten *len* members, but just one shared *num_strings* member. See Figure 10-6. This is convenient for data that should be private to a class but that should have the same value for all class objects. The *num_strings* member, for example, is intended to keep track of the number of objects created.

Let's look at the implementation of the class methods in Listing 10-18. There you'll see how these two points (using a pointer and using a static member) are handled.

LISTING 10-18 strngs.cpp

```
// strngs.cpp -- string class methods
#include <iostream.h>
#include <string.h>
#include "strngs.h"

// initializing static class member
int string::num_strings = 0;

// class methods
void string::says()                          // display method
{
    cout << str << "\n";

}

string::string(char * s)
{
    str = new char[strlen(s) + 1];           // allot storage
    strcpy(str, s);                          // initialize pointer
    len = strlen(str);
    num_strings++;                           // set object count
    cout << num_strings << ": \"" << str
         << "\" object created\n";           // For Your Information
}

string::string()
{
    str = new char[4];
    strcpy(str, "C++");                      // default string
    len = strlen(str);
    num_strings++;
    cout << num_strings << ": \"" << str
          << "\" object created\n";          // FYI
}
```

```
string::~string()
{
    cout << "\"" << str << "\" object deleted, ";   // FYI
    cout << --num_strings << " left\n";             // FYI
    delete str;                                     // required
}
```

First, notice the following statement from Listing 10-18:

```
int string::num_strings = 0;
```

This statement initializes the static *num_strings* member to zero. Note that you cannot initialize a static member variable inside the class definition. That's because the definition is a template and doesn't allocate memory. Usually, you allocate and initialize memory by creating an object using that template. In the case of a static class member, you initialize the static member independently with a separate statement outside the class definition. That's because the static class member is stored separately rather than as part of an object. Note that the initialization statement gives the type and uses the scope operator.

```
int string::num_strings = 0;
```

This initialization goes in the methods file, not in the class definition file. That's because the class definition is in a header file, and a program may include a header file in several other files. That would result in multiple copies of the initialization statement, which is an error.

Next, notice that each constructor contains the expression `num_strings++`. This ensures that each time a program creates a new object, the shared variable *num_strings* increases by one, keeping track of the total number of string objects. Also, the destructor contains the expression `--num_strings`. Thus the *string* class also keeps track of deleted objects, keeping the value of the *num_strings* member current.

Now look at the first constructor, which initializes a *string* object to a string:

```
string::string(char * s)
{
    str = new char[strlen(s) + 1];      // allot storage
    strcpy(str, s);                     // initialize pointer
   len = strlen(str);
    num_strings++;                      // set object count
    cout << num_strings << ": \"" << str
         << "\" object created\n";      // For Your Information
}
```

The class *str* member, recall, is just a pointer, so the constructor has to provide the memory for holding a string. You can pass a string pointer to the constructor when you initialize an object:

```
string boston("Boston");
```

The constructor uses the `strlen()` function to compute the length of the string, uses `new` to allocate sufficient space to hold the string, then assigns the address of the new memory to the *str* member. (Recall that `strlen()` returns the length of a string not counting the terminating null character, so the constructor adds 1 to the `strlen()` return value to allow space for the string including the null character.)

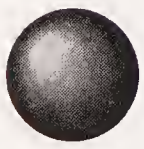
Next, the constructor uses `strcpy()` to copy the passed string into the new memory. Then it initializes the *len* member and updates the object count. Finally, to help us monitor what's going on, the constructor displays the current number of objects and the string stored in the object.

The default constructor behaves similarly, except that it provides a default string of C++.

The destructor contains the example's most important addition to our handling of classes:

```
string::~string()
{
    cout << "\"" << str << "\" object deleted, ";     // FYI
    cout << --num_strings << " left\n";               // FYI
    delete str;                                       // required
}
```

The destructor begins by announcing when the destructor gets called. This part is informative, but not essential. The `delete` statement, however, is vital. Recall that the *str* member points to memory allocated with `new`. When a *string* object expires, the *str* pointer expires. But the memory *str* pointed to remains allocated unless we use `delete` to free it. Freeing memory *used* by an object's members does not automatically free memory *pointed to* by an object's members. By placing the `delete` statement in the destructor, we ensure that the memory allocated with `new` is freed when the object expires.



Whenever you use `new` in a constructor to allocate memory, you should use `delete` in the corresponding destructor to free that memory.

Listing 10-19, taken from a program under development at *The Daily Vegetable*, illustrates when and how the *string* constructors and destructor work. Remember to compile Listing 10-18 along with Listing 10-19.

LISTING 10-19 vegnews.cpp

```
// vegnews.cpp -- using new and delete with classes
// compile with strngs.cpp
#include <iostream.h>
#include "strngs.h"

string sports("Spinach Leaves Bowl for Dollars");
                                    // sports an external object
void callme1(void);                 // creates local object
string * callme2(void);             // creates dynamic object

int main(void)
{
    cout << "Top of main()\n";
    string headlines[2] =           // local object array
    {
            string("Celery Stalks at Midnight"),
            string("Lettuce Prey")
    };
    headlines[0].says();
    headlines[1].says();
    callme1();
    cout << "Middle of main()\n";
    string *pr = callme2();         // set pointer to object
    sports.says();
    pr->says();                     // invoke class method
    delete pr;                      // delete object
    cout << "End of main()\n";
    return 0;
}

void callme1(void)
{
    cout << "Top of callme1()\n";
    string grub;                  // local object
    grub.says();
    cout << "End of callme1()\n";
}

string * callme2(void)
{
    cout << "Top of callme2()\n";
    string *pveg = new string("Cabbage Heads Home");
                  // dynamic object uses constructor
    pveg->says();
    cout << "End of callme2()\n";
    return pveg;                    // pveg expires, object lives
}
```

Here is the output:

```
1: "Spinach Leaves Bowl for Dollars" object created
Top of main()
2: "Celery Stalks at Midnight" object created
3: "Lettuce Prey" object created
Celery Stalks at Midnight
Lettuce Prey
Top of callme1()
4: "C++" object created
C++
End of callme1()
"C++" object deleted, 3 left
Middle of main()
Top of callme2()
4: "Cabbage Heads Home" object created
Cabbage Heads Home
End of callme2()
Spinach Leaves Bowl for Dollars
Cabbage Heads Home
"Cabbage Heads Home" object deleted, 3 left
End of main()
"Celery Stalks at Midnight" object deleted, 2 left
"Lettuce Prey" object deleted, 1 left
"Spinach Leaves Bowl for Dollars" object deleted, 0 left
```

Program Notes

You should make sure you understand the sequence of events in this example program, so let's run through them now. The *sports* object is an external variable, so it's created before `main()` begins execution. The next objects created are the two elements of the *headlines* array. The program calls the constructor twice, once to initialize each array element. Each element is a class object, so the calls

```
headlines[0].says();
headlines[1].says();
```

evoke the class method `says()` for the two objects *headlines[0]* and *headlines[1]*.

Next, the program calls the `callme1()` function. This function uses the default constructor to create a local object called *grub*. The default constructor initializes the *str* member to "C++". This object expires when the `callme1()` function finishes execution, as is shown by the following output lines:

```
Top of callme1()
4: "C++" object created
C++
```

```
End of callme1()
"C++" object deleted, 3 left
```

In addition to printing its farewell message, the destructor also frees the memory that held the string *"C++"*.

Now we come to the thorniest part of the example. The program calls `callme2()`, and this function uses `new` to create and initialize a *string* object:

```
string *pveg = new string("Cabbage Heads Home");
```

The function assigns the address of this new object to the `pveg` pointer. Because we provide *string* with a string argument, the program calls the corresponding constructor to initialize the object. Figure 10-7 summarizes the statement.

Just as with structures, you can use the `->` operator to access class members. In this case, `vegnews.cpp` uses the operator to invoke a class method:

```
pveg->says();
```

This causes the program to display the `"Cabbage Heads Home"` string. Then the function terminates, freeing the memory used by its variables. This means the memory used to hold the pointer *pveg* is freed. But because `callme2()` doesn't use `delete pveg`, the memory holding the object to which *pveg* pointed is still allocated. Note that there is no destructor message when `callme2()` terminates; that shows the object still is present—the string `"Cabbage Heads Home"` lives on! But because *pveg* has expired, the program no longer can access that object with *pveg*. However, the program passes the value of *pveg* back to the calling program and assigns it to the pointer *pr*. In short, at first *pveg* pointed to the string. Then *pveg* expired, but meanwhile the program set *pr* to point to the string. Thus the program now can use *pr* to access the dynamic object. And it does so after first displaying the *sports* object:

```
Top of callme2()
4: "Cabbage Heads Home" object created
Cabbage Heads Home                    ← uses pveg pointer in callme2()
End of callme2()
Spinach Leaves Bowl for Dollars
Cabbage Heads Home                    ← uses pr pointer in main()
```

Now the program moves to the sad tasks of deleting the remaining objects. Because it created the cabbage object with `new`, it can remove it with `delete`:

```
delete pr;
```

Recall that this frees the memory *pr* points to and not *pr* itself. Deleting the pointed-to object activates the destructor:

```
"Cabbage Heads Home" object deleted, 3 left
```

```
string *pyh = new string("Misty Dynamic");
```

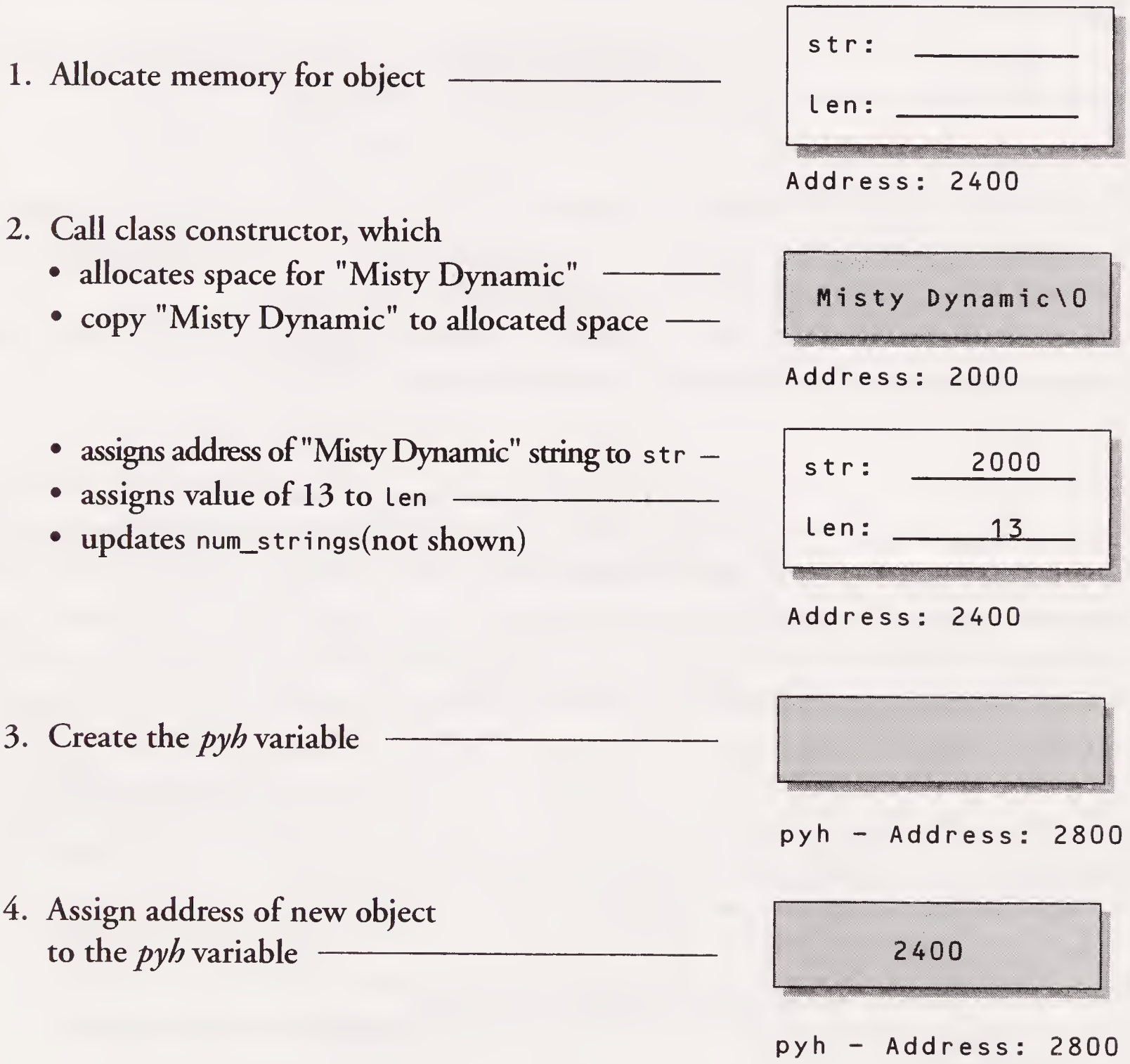


Figure 10-7 new *and the class constructor*

Scoping rules control the existence of the remaining objects. The two array elements are automatic variables, so they expire when execution leaves the block in which they are defined. In this case, the block is the body of the `main()` function, so these two objects are freed when `main()` terminates. Last of all, the external object expires when the program terminates:

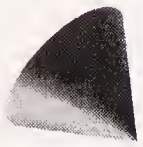
```
End of main()
"Celery Stalks at Midnight" object deleted, 2 left
"Lettuce Prey" object deleted, 1 left
"Spinach Leaves Bowl for Dollars" object deleted, 0 left
```

Note the program uses `new` and `delete` on two levels. First, it uses `new` to allocate storage space for the name strings for each object that is created. This happens in the constructor functions, so the destructor function uses `delete` to free that memory. Second, the program uses `new` to allocate an entire object. This allocates space not for the name string but for the object, which includes the *str* pointer that holds the address of the string. The `new` operator also allocates space for the *len* member. It does not allocate space for the *num_strings* member because that is a static member stored separately from the objects. The program then used `delete` to delete this object when it was finished with it. Again, this frees only the space used to hold the *str* pointer and the *len* member. It doesn't free the memory used to hold the string *str* points to, but the destructor takes care of that final task.

Once you've tucked this example under your belt, you're ready for further adventures in memory management. Meanwhile, it's time for another deep, relaxing breath and for another treat to reward yourself. Not only do you deserve it, but you'll need the break, because the next item on our OOP menu is a substantial serving of several related matters.

BACK TO SOFTBALL: IMPROVED MEMORY MANAGEMENT

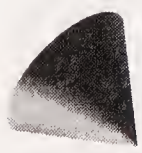
You've learned enough now to improve the *softball* class definition (see Chapter 9). (If you're tired of softball, you can modify the program to represent, say, baseball, soccer, or bowling.) For instance, you can now replace the `show_stats()` function with an overloaded `<<` operator so that you can use `cout` to print object contents. And, as we did with the *string* class, you can use `new` instead of a fixed array to allocate space for names. Making these changes is more than practicing new skills. (And don't forget, you do have to *practice* and *experiment* with these new skills to really understand them.) It also leads you to a new and subtle problem—just what you must have been hoping for! But you have to make the changes first before you can uncover that problem. Listing 10-20 provides the new class definition. The new elements here are the redefinition of *firstname* and *lastname* as pointers instead of arrays, the new friend function for overloading the `<<` operator, and the disappearance of the now unneeded `show_stats()` prototype.



LISTING 10-20 softbal4.h

```
// softbal4.h -- revised header file for softball class
class softball
{
private:
    char * firstname;           // replaces char array
    char * lastname;            // replaces char array
    unsigned atbats;
    unsigned hits;
    unsigned rbis;
    float ave;
    float calc_ave();
public:
    softball(char * fn, char * ln, unsigned ab = 0,
            unsigned h = 0, unsigned rbi = 0);
    softball(void);
    ~softball();
    void setall();
    void update();
    softball& topbat(softball& player);
// friend function
    friend ostream & operator<<(ostream & os, softball & player);
};
```

Next, Listing 10-21 provides the methods. The revised constructor follows the lead of the *string* class, using `new` and `strcpy()` to create and initialize string storage, just as we did with the *string* class. The `setall()` method uses the same technique. Because the constructor uses `new`, the destructor must use `delete`. Having a hunch about the kind of problem that's going to appear, we've put an output statement in the destructor so you can watch it at work. Finally, the `operator<<()` function for displaying an object follows the same form we used for the *vector* class, returning a reference to an *ostream* object.



LISTING 10-21 softbal4.cpp

```
// softbal4.cpp -- revised methods for softball class
#include <iostream.h>
#include <string.h>
#include "softbal4.h"

softball:: softball(void)
{
}
```

```
// revised constructor allocates string space
softball:: softball(char * fn, char * ln, unsigned ab,
                          unsigned h, unsigned rbi)
{
    firstname = new char[strlen(fn) + 1];
    strcpy(firstname, fn);
    lastname = new char[strlen(ln) + 1];
    strcpy(lastname, ln);
    atbats = ab;
    hits = h;
    rbis = rbi;
    ave = calc_ave();
}

softball::~softball()
{
    cout << "Bye, " << firstname << "\n";
    delete firstname;
    delete lastname;
}

float softball::calc_ave()
{
    if (atbats == 0)
          return 0;
    else
          return float(hits) / atbats;
}

void softball::setall()
{
    char temp[80];
    cout << "Enter player's first name: ";
    cin >> temp;
    firstname = new char[strlen(temp) + 1];
    strcpy(firstname, temp);
    cout << "Enter player's last name: ";
    cin >> temp;
    lastname = new char[strlen(temp) + 1];
    strcpy(lastname, temp);

    cout << "Enter number of at bats: ";
    cin >> atbats;
    cout << "Enter number of hits: ";
    cin >> hits;
    cout << "Enter number of RBIs: ";
    cin >> rbis;
```

```
    ave = calc_ave();
    cout << "\n";
}

void softball::update()
{
    cout << "Enter stats for " << firstname << " ";
   cout << lastname << ":\n";
    cout << "Enter additional number of at bats: ";
    int temp;
    cin >> temp;
    atbats = atbats + temp;
    cout << "Enter additional number of hits: ";
    cin >> temp;
    hits = hits + temp;
    cout << "Enter additional number of RBIs: ";
    cin >> temp;
    rbis = rbis + temp;
    ave = calc_ave();
    cout << "\n";
}

// replace show_stats() with overloaded << operator
ostream& operator<<(ostream & os, softball & player)
{
    os << "Player: " << player.firstname << " "
      << player.lastname << "\n" << "At bats: "
      << player.atbats << "   Hits: " << player.hits
      << "   RBIS: " << player.rbis << "   BA: "
      << player.ave << "\n\n";
    return os;
}

softball& softball::topbat(softball& player)
{
    if (player.ave > ave)
            return player;
    else
            return *this;
}
```

Listing 10-22 presents a program that exercises the new class definition. It uses the standard library `exit()` function, which has a prototype in the `stdlib.h` header file. This function, when called, terminates the program and returns an integer value to the operating system, much like `return` does when used in `main()`. One difference be-

tween `return 0` and `exit(0)` is that `return` is a keyword, while `exit()` is a function call. A second difference is that `exit()` terminates a program even if called from a function other than `main()`. What the operating system does with the return value is a UNIX topic or a DOS topic. In general terms, you can use a UNIX script or a DOS batch file to run a C++ program, and certain script commands and batch file commands can examine a program's return value and use it to decide what to do next. The usual convention is that a zero return value means all went well, while a nonzero return value indicates some sort of problem. Listing 10-22, for example, uses `exit()` to return a value of `1` if the user fails to enter enough players. The program also illustrates how you can use `new` to create dynamic arrays of objects and how to delete a dynamic array. Finally, the program uncovers a problem. Remember to compile Listing 10-21 with Listing 10-22.

LISTING 10-22 dynamic1.cpp

```
// dynamic1.cpp -- using new two different ways
// compile with softbal4.cpp
#include <iostream.h>
#include <stdlib.h>              // prototype for exit()
#include "softbal4.h"
int main(void)
{
    int players;

    cout << "Enter number of players: ";
    cin >> players;
    if (players < 1)
    {
          cout << "You need at least one to play! Bye.\n";
          exit(1);               // terminate program
    }
    softball * team = new softball[players];  // dynamic array

    for (int i = 0; i < players; i++)
          team[i].setall();

    for (i = 0; i < players; i++)
          cout << team[i];       // overloaded <<

    softball tops = team[0];     // initialize tops to team[0]
    for (i = 1; i < players; i++)
          tops = tops.topbat(team[i]);
    cout << "Tops:\n";
    cout << tops;
```

```
    delete [players] team;      // delete all members of array
    cout << "bye\n";            // delete syntax explained later
    return 0;
}
```

Here is the program output:

```
Enter number of players: 3
Enter player's first name: Jane
Enter player's last name: Austin
Enter number of at bats: 6
Enter number of hits: 4
Enter number of RBIs: 3

Enter player's first name: Mary
Enter player's last name: Shelley
Enter number of at bats: 5
Enter number of hits: 2
Enter number of RBIs: 1

Enter player's first name: Emily
Enter player's last name: Bronte
Enter number of at bats: 5
Enter number of hits: 3
Enter number of RBIs: 2

Player: Jane Austin
At bats: 6   Hits: 4   RBIS: 3   BA: 0.666667

Player: Mary Shelley
At bats: 5   Hits: 2   RBIS: 1   BA: 0.4

Player: Emily Bronte
At bats: 5   Hits: 3   RBIS: 2   BA: 0.6

Tops:
Player: Jane Austin
At bats: 6   Hits: 4   RBIS: 3   BA: 0.666667

Bye, Jane
Bye, Mary
Bye, Emily
bye
Bye, &!!&!!
Null pointer assignment
```

Note the troubled lines at the end. They're a sign of new problems. But before tackling them, let's analyze the problem-free part of the program.

Program Notes

By using `new` to create objects, the program avoids being committed to a particular number of players. Instead, you can enter the number of players, then create a dynamic array of that size:

```
softball * team = new softball[players];
```

Creating a dynamic array of objects follows the same form as creating a dynamic array of a standard type—follow the keyword `new` with the type name, then with brackets containing the number of elements desired. The pointer *team* then points to the first object in the array. Recall that if *team* points to an element of array, you can use the notation `team[0]` to indicate the first element, `team[1]` to indicate the second element, and so on. Because each element is an object, each element can invoke a class method:

```
for (int i = 0; i < players; i++)
        team[i].setall();
```

Similarly, you can use the << operator to display the contents of each object:

```
for (i = 0; i < players; i++)
    cout << team[i];
```

As the program output shows, this part of the program worked just as it should.

Next, the program creates a new *softball* object and initializes it to `team[0]`:

```
softball tops = team[0];
```

This copies the contents of the `team[0]` object into the `tops` object.

Then the program uses a loop and the `topbat()` method to find the object with the highest batting average. The `topbat()` method, recall, compares two objects. One is the object that invokes the method, and the second is the object provided as an argument.

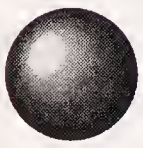
```
for (i = 1; i < players; i++)
    tops = tops.topbat(team[i]);
```

Thus the expression `tops.topbat(team[i])` compares the *tops* object with the `team[i]` object. Then the one with the higher batting average is assigned to *tops*. Again, the program output demonstrates the superb design of the program.

Finally, the program deletes those objects no longer needed. Note this line:

```
delete [players] team;
```

In creating the *team* array, `new` called the default *softball* constructor three times, once for each object in the array. Therefore, to free that memory, the program also has to call the *softball* destructor three times. In general, if the array has *players* elements, then the program should call the destructor *players* times. That's what the expression `delete [players] team` accomplishes. The value in the brackets indicates how many times to call the destructor. The *team* pointer points to the first element, and the first destructor call frees the memory used for that element. The program then uses pointer arithmetic to figure out the address of the next element, then frees that block of memory, and so on. If we were to omit the `[players]` part, the program would delete only the first element of the array. As the `Byes` in the program show, the program successfully frees the memory for each of the three objects in the array.



If you use the form

```
softball * team = new softball[players];
```

to create a dynamic array of objects whose constructors use `new`, then you need to use the form

```
delete [players] team;
```

in order to summon the class destructor for each array element.

After the program prints `Bye`, it frees the variable *tops*. That's because *tops* is a local automatic variable, and automatic variables expire when the program finishes executing the block containing them (see Chapter 8). And here something goes awry. First, there's some peculiar output:

```
Bye, &!!&!!
```

Then, after the program terminates, there's this cryptic and unsettling message:

```
Null pointer assignment
```

The blame for these unpleasantries lies with what happens when you copy one object containing pointers to another.

POINTER PROBLEMS IN OBJECT INITIALIZATION AND ASSIGNMENT

Copying objects containing pointers can be troublesome. The `dynamic1.cpp` program potentially copies such objects in two places. First, it initializes one object to another:

```
softball tops = team[0];
```

This copies the contents of the `team[0]` object to the newly created `tops` object. Second, depending on how the batting average comparison goes, the program assigns one object to another:

```
for (i = 1; i < players; i++)
    tops = tops.topbat(team[i]);
```

Although initialization and assignment both assign values to an object, they differ conceptually. Initialization takes place only when an object is created, while assignment takes place with previously created objects. We'll look at the problem that arises when you initialize an object, but that same situation occurs in assignment.

Consider what happens when you initialize one *softball* object to another. Listing 10-22 uses the following statement:

```
softball tops = team[0];
```

By default, C++ uses *memberwise initialization.* That means each data member of *tops* is a copy of the corresponding data member in `team[0]`. For instance, `team[0].hits` has the value `4`, so `tops.hits` is assigned the value `4`. So far, so good. But now consider *firstname.* The value of `team[0].firstname` is not the string *Jane.* It is the *address* of the string *Jane.* That means that initialization copies the *address* of the string to `tops->firstname`. In short, we now have two members holding the same address, and we have only one string. Initialization copied only the address, not the string. See Figure 10-8.

Having two pointers to the same string causes a problem when the program reaches the deletion stage. The statement

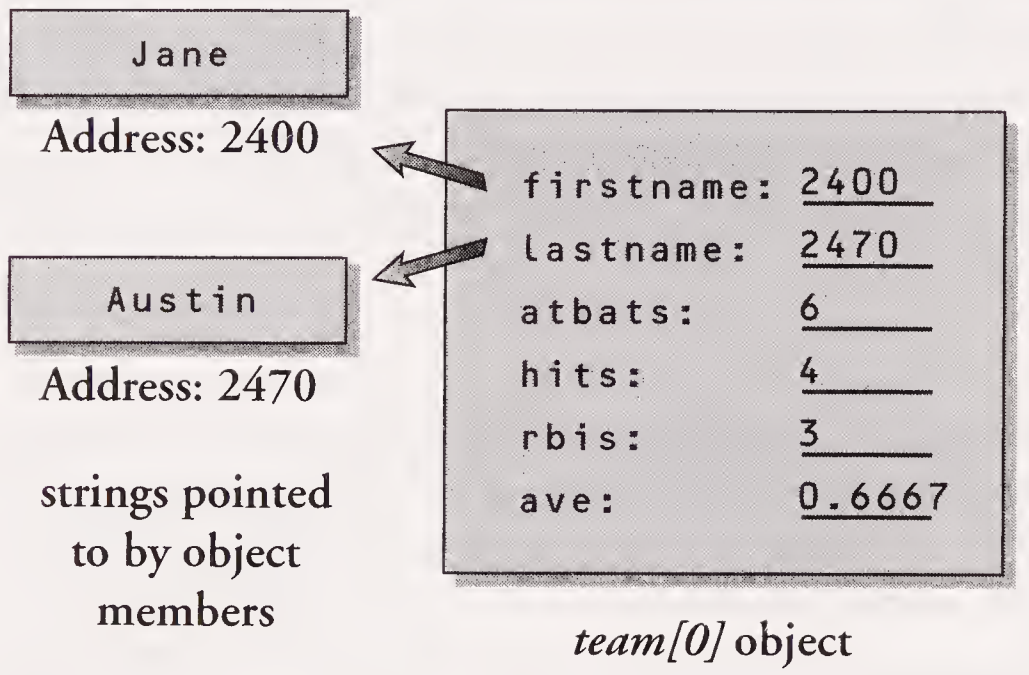
```
delete [players] team;
```

frees the memory used to hold the `team[0]` object, along with the memory holding the remaining *team* objects. At this time, the pointer `team[0].firstname` passes from the realm of the known. That would have left the pointed-to string of *"Jane"* still in memory except that deleting the `team[0]` object invokes the class destructor function. And that function, recall, deletes the string to which the *firstname* pointer pointed. The *lastname* pointer is handled similarly. Thus between `delete` and the destructor function, the program frees the memory used by the object *and* by the strings pointed to by the object members. It's a clean sweep.

But then the program automatically deletes the `tops` object when `main()` terminates. This frees the memory used to hold the object members, including the `tops.firstname` pointer. But in doing so, the program calls the destructor function one more time, and the destructor function attempts to free the memory to which `tops.firstname` pointed. This is the same memory to which `team.[0].firstname` pointed, and that memory already has been freed. Apparently, the program already has

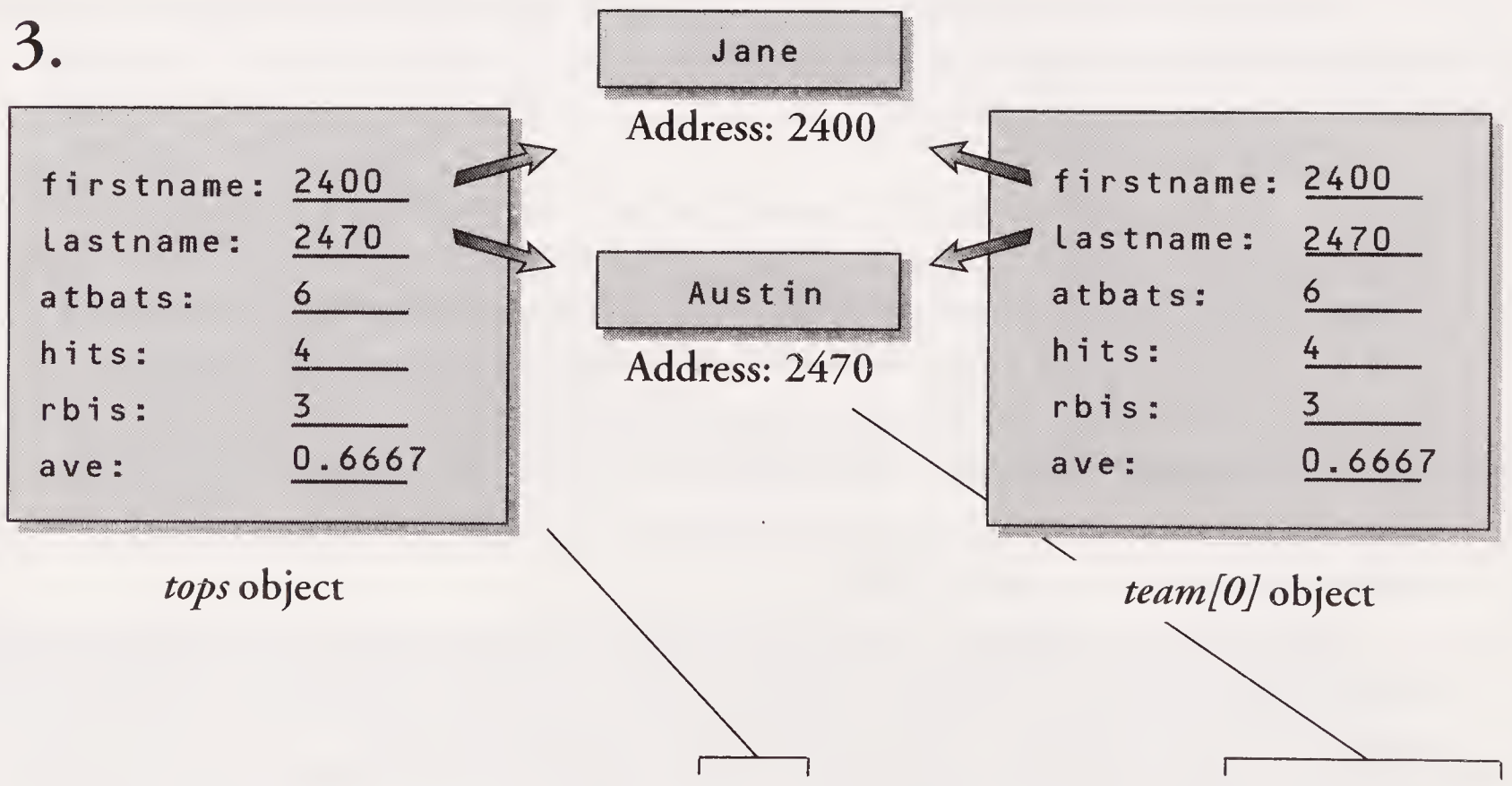
used that memory for something else, which is why the destructor prints the string as something other than *"Jane"*. And the attempt to free memory that's already been freed can lead to the *Null pointer assignment* message. In other words, we wound up with two pointers to the same memory location and wound up trying to free that memory twice. Programs don't like that, and that's good reason for you not to like it.

1.



2. `softball tops = team[0];  // initialize tops to team[0]`

3.



Default initialization duplicates member values but does not duplicate pointed-to data.

Figure 10-8 *Memberwise initialization*

By default, C++ assignment works the same way as initialization, using memberwise copying. So assignment, too, can lead to two pointers pointing to the same memory, raising the specter of double deletion.

FIXING THE OBJECT INITIALIZATION PROBLEM

You've now seen that initializing one object to another causes problems *if* the class includes members pointing to memory allocated by `new`. The way to fix the problem is to change the way initialization works. Default initialization, as you've seen, copies one pointer member to the other, leaving both pointers pointing to the same thing. But you can change that so that initialization instead copies the *item* to which the pointer points. That way, you wind up with two pointers pointing to two different data items. One data item is an exact copy of the other, but at a different location. That makes the values of the two pointers different. The program gets *two* pointers and *two* strings. Thus when it deletes the two objects, the program doesn't wind up trying to delete the same block of memory twice.

C++ provides a mechanism for changing how object initialization works. It consists of defining a special constructor, called a *copy constructor*, with this general form:

```
classname(const classname &)
```

Here the ampersand means the argument is a reference argument. If defined, this constructor is used in three situations:

- When you initialize one class object to another, the constructor sets values for the initialized object.
- When you pass a object by value as an argument, the constructor initializes the corresponding formal argument in the called function.
- When you have a function return an object, the constructor initializes a temporary object used to convey the object's values to the calling program.

If you don't define this special constructor, C++ uses memberwise initialization in these three cases. This is a nice break—by writing a copy constructor you can head off three separate problem areas!

Let's apply this constructor to the *softball* class. It should create copies of the pointed-to strings in addition to copying built-in data. So you can do the following:

```
softball::softball(const softball & player)
{
// allocate space to hold the name
    firstname = new char[strlen(player.firstname) + 1];

// copy name from player to new object
```

```
    strcpy(firstname, player.firstname);

    lastname = new char[strlen(player.lastname) + 1];
    strcpy(lastname, player.lastname);
    atbats = player.atbats;
    hits = player.hits;
    rbis = player.rbis;
    ave = calc_ave();
}
```

This new constructor determines the length of the string that `player.firstname` points to, uses `new` to allocate enough space to hold a copy, and assigns the address of that space to *firstname*, which is a pointer member in the new object. Then it copies the original string to that location. Note that the two *firstname* members (*firstname* and `player.firstname`) have different values because they point to different locations. The *lastname* member gets the same treatment. In short, memberwise initialization copies the pointer, while this definition copies the string. See Figure 10-9 and compare it to Figure 10-8.

FIXING THE ASSIGNMENT PROBLEM

Assigning one object to another poses the same problem as initialization, at least if the class has members pointing to memory allocated by `new`. The problem has a similar solution: redefine how assignment works. You can do this by overloading the assignment operator. The assignment operator is one of the few that can be overloaded only with a member function, not a friend function, so that's the route we'll take.

Redefining assignment has two more twists to it beyond what you had to do for initialization. First, because you assign an object to another object that's been defined previously, the receiving object's member pointers may already point to other data. If you reassign the object's member pointers to point to new data, you should first free the memory used for the old data.

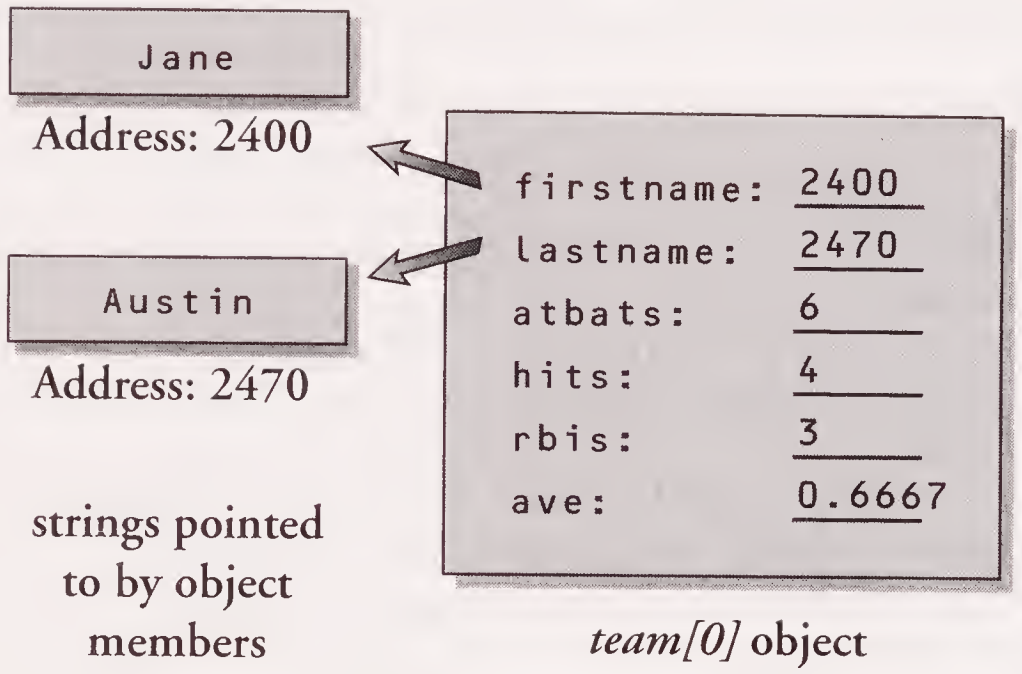
Second, the operator function should return a reference to an object. That way, you can chain overloaded assignment the same as you do regular assignment:

```
int x, y, z;
x = y = z = 3;                          // chaining assignment
softball tops, worst;
softball catcher("Pickles", "Malarky", 8, 3, 1);
tops = worst = catcher;                 // chaining assignment
```

In function notation, the last statement becomes the following:

```
tops.operator=(worst.operator=(catcher));
```

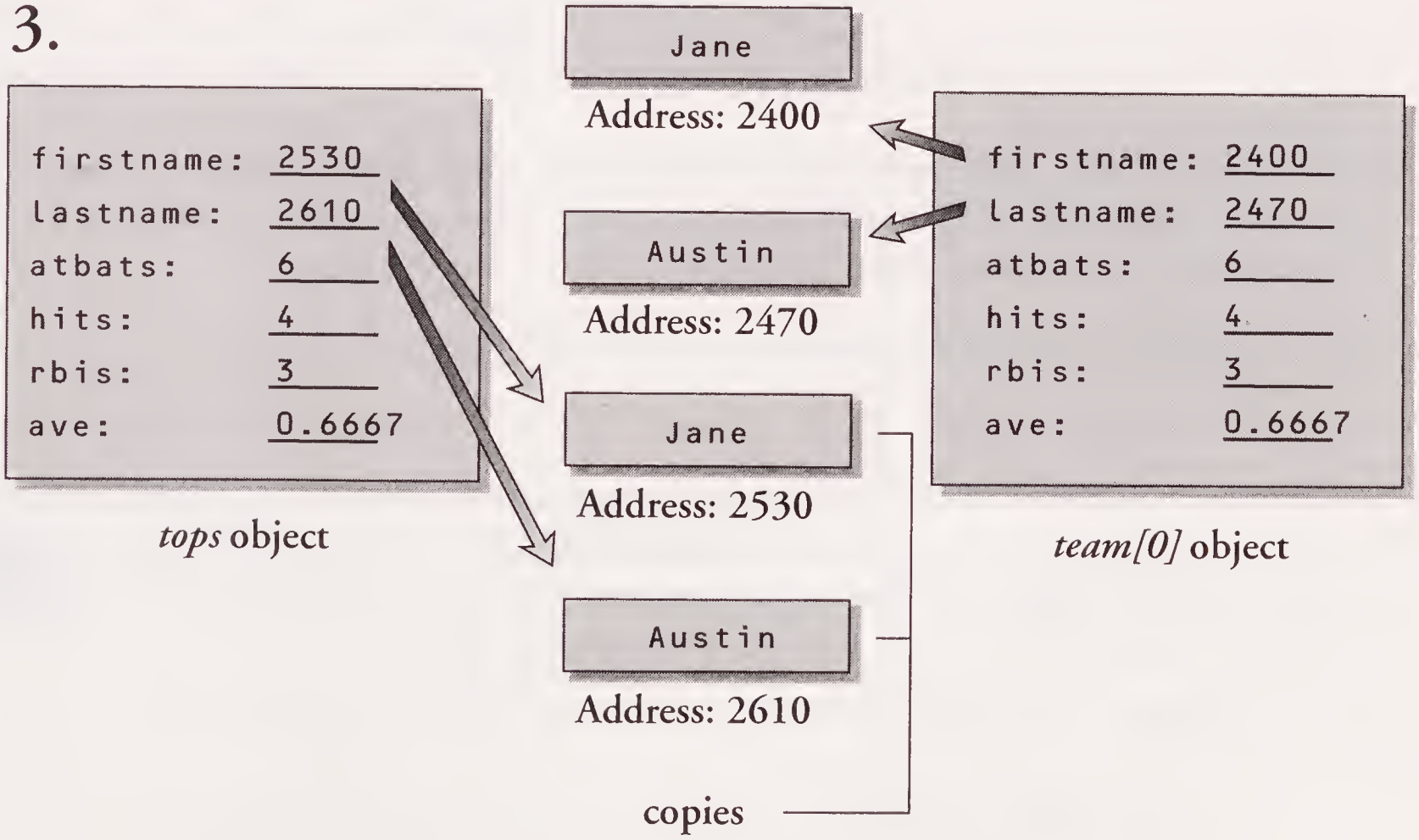
1.



2.

```
softball tops = team[0];  // initialize tops to team[0]
```

3.



The *softball(softball & player)* constructor was defined so that it copies the pointed-to strings and sets the member pointers accordingly, giving two objects and two separate sets of data.

Figure 10-9 *Using redefined initialization*

In other words, the return value of the *worst* call becomes the argument of the *tops* call. This works if the return value is of type *softball &*.

The following definition follows these guidelines:

```
softball & softball::operator=(softball & player)
{
    if (this == &player)
       return *this;              // no change if assigned to self
    delete firstname;
    firstname = new char[strlen(player.firstname) + 1];
    strcpy(firstname, player.firstname);
    delete lastname;
    lastname = new char[strlen(player.lastname) + 1];
    strcpy(lastname, player.lastname);
    atbats = player.atbats;
    hits = player.hits;
    rbis = player.rbis;
    ave = calc_ave();
    return *this;
}
```

As Chapter 9 revealed, `this` is a pointer to the invoking object, so `*this` is the invoking object. Thus the function returns the invoking object after it has been set to the new values.

There's one more precaution to take. This assignment function assumes that *firstname* and *lastname* were initialized previously. That assumption is not true for objects created with the old default constructor, for it does nothing:

```
softball::softball(void)
{
}
```

Therefore you have to redefine the default constructor so that it does initialize the two pointer members. The simplest route is to initialize the two pointers to the null pointer (the pointer having the value zero and represented by `NULL` in `iostream.h`), for it does no harm to apply the `delete` operator to the null pointer:

```
softball::softball(void)
{
    firstname = NULL;
    lastname = NULL;
}
```

Listing 10-23 presents the new class definition; it now declares the new methods for initialization and assignment.

LISTING 10-23 softbal5.h

```
// softbal5.h -- revised header file for softball class
class softball
{
private:
    char * firstname;
    char * lastname;
    unsigned atbats;
    unsigned hits;
    unsigned rbis;
    float ave;
    float calc_ave();
public:
    softball(const softball &player);          // new initializer
    softball(char * fn, char * ln, unsigned ab = 0,
           unsigned h = 0, unsigned rbi = 0);
    softball(void);
    ~softball();
    void setall();
    void update();
    softball& topbat(softball& player);
// overloaded operators
    softball & operator=(softball & player); // new =
// friend function
    friend ostream & operator<<(ostream & os, softball & player);
};
```

Listing 10-24 provides the two new class methods and the revised default constructor.

LISTING 10-24 softbal5.cpp

```
// softbal5.cpp -- revised methods for softball class
#include <iostream.h>
#include <string.h>
#include "softbal5.h"

softball:: softball(void)
{
    firstname = NULL;
    lastname = NULL;
}
```

```
// special constructor initializes object to an object
softball:: softball(const softball & player)
{
    firstname = new char[strlen(player.firstname) + 1];
    strcpy(firstname, player.firstname);
    lastname = new char[strlen(player.lastname) + 1];
    strcpy(lastname, player.lastname);
    atbats = player.atbats;
    hits = player.hits;
    rbis = player.rbis;
    ave = calc_ave();
}

// revised constructor allocates string space
softball:: softball(char * fn, char * ln, unsigned ab,
                    unsigned h, unsigned rbi)
{
    firstname = new char[strlen(fn) + 1];
    strcpy(firstname, fn);
    lastname = new char[strlen(ln) + 1];
    strcpy(lastname, ln);
    atbats = ab;
    hits = h;
    rbis = rbi;
    ave = calc_ave();
}

softball::~softball()
{
    cout << "Bye, " << firstname << "\n";
    delete firstname;
    delete lastname;
}

float softball::calc_ave()
{
    if (atbats == 0)
            return 0;
    else
            return float(hits) / atbats;
}

void softball::setall()
{
    char temp[80];
    cout << "Enter player's first name: ";
    cin >> temp;
```

```
    firstname = new char[strlen(temp) + 1];
    strcpy(firstname, temp);
    cout << "Enter player's last name: ";
    cin >> temp;
    lastname = new char[strlen(temp) + 1];
    strcpy(lastname, temp);

    cout << "Enter number of at bats: ";
    cin >> atbats;
    cout << "Enter number of hits: ";
    cin >> hits;
    cout << "Enter number of RBIs: ";
    cin >> rbis;
    ave = calc_ave();
    cout << "\n";
}

void softball::update()
{
    cout << "Enter stats for " << firstname << " ";
    cout << lastname << ":\n";
    cout << "Enter additional number of at bats: ";
    int temp;
    cin >> temp;
    atbats = atbats + temp;
    cout << "Enter additional number of hits: ";
    cin >> temp;
    hits = hits + temp;
    cout << "Enter additional number of RBIs: ";
    cin >> temp;
    rbis = rbis + temp;
    ave = calc_ave();
    cout << "\n";
}

// replace show_stats() with overloaded << operator
ostream& operator<<(ostream & os, softball & player)
{
    os << "Player: " << player.firstname << " "
    << player.lastname << "\n" << "At bats: "
    << player.atbats << "   Hits: " << player.hits
    << "   RBIS: " << player.rbis << "   BA: "
    << player.ave << "\n\n";
    return os;
}

softball & softball::topbat(softball& player)
{
```

```
    if (player.ave > ave)
            return player;
    else
            return *this;
}

// overload assignment
softball & softball::operator=(softball & player)
{
    if (this == &player)
        return *this;           // no change if assigned to self
    delete firstname;
    firstname = new char[strlen(player.firstname) + 1];
    strcpy(firstname, player.firstname);
    delete lastname;
    lastname = new char[strlen(player.lastname) + 1];
    strcpy(lastname, player.lastname);
    atbats = player.atbats;
    hits = player.hits;
    rbis = player.rbis;
    ave = calc_ave();
    return *this;
}
```

You can recompile the program in Listing 10-22 using the new definitions and run the result. The only change needed in the listing is to replace

```
#include "softbal4.h"
```

with

```
#include "softbal5.h"
```

And, of course, you should compile Listing 10-22 along with Listing 10-24 instead of with Listing 10-21. As you can see in the following sample run, the program now works correctly.

```
Enter number of players: 3
Enter player's first name: Linda
Enter player's last name: Ronstadt
Enter number of at bats: 9
Enter number of hits: 5
Enter number of RBIs: 1

Enter player's first name: Elvis
Enter player's last name: Presley
Enter number of at bats: 6
```

```
Enter number of hits: 0
Enter number of RBIs: 0

Enter player's first name: Bruce
Enter player's last name: Springsteen
Enter number of at bats: 10
Enter number of hits: 6
Enter number of RBIs: 2

Player: Linda Ronstadt
At bats: 9        Hits: 5        RBIS: 1        BA: 0.555556

Player: Elvis Presley
At bats: 6        Hits: 0        RBIS: 0        BA: 0

Player: Bruce Springsteen
At bats: 10       Hits: 6        RBIS: 2        BA: 0.6

Tops:
Player: Bruce Springsteen
At bats: 10       Hits: 6        RBIS: 2        BA: 0.6

Bye, Linda
Bye, Elvis
Bye, Bruce
bye
Bye, Bruce
```

REVIEWING TECHNIQUES

By now you've encountered several programming techniques for dealing with various class-related problems, and you may be having trouble keeping track of all of them. So let's summarize several techniques and when they are used.

OVERLOADING THE << OPERATOR

To redefine the `<<` operator so that you use it with `cout` to display an object's contents, define a friend operator function of the following form:

```
ostream & operator<<(ostream & os, const c_name & obj)
{
    os << ... ;  // display object contents
    return os;
}
```

Here *c_name* represents the name of the class.

Conversion Functions

To convert a single value to class type, create a class constructor of the following form of prototype:

```
c_name(type_name value);
```

Here *c_name* represents the class name, and *type_name* represents the name of the type to which you wish to convert.

To convert a class type to some other type, create a class member function having the following prototype:

```
operator typename();
```

Although the function has no declared return type, it should return a value of the desired type.

Classes Whose Constructors Use new

Classes that use the `new` operator to allocate memory pointed to by a class member require several precautions in the design.

1. Any class member pointing to memory allocated by `new` should have the `delete` operator applied to it in the class destructor. This frees the allocated memory.

2. If a destructor frees memory by applying `delete` to a pointer that is a class member, then every constructor for that class should initialize that pointer either by using `new` or by setting the pointer to the `NULL` pointer.

3. You should define a copy constructor that allocates new memory rather than copying a pointer to existing memory. This enables a program to initialize one class object to another. The constructor should have the following form of prototype:

   ```
   classname(const classname&)
   ```

4. You should define a class member function overloading the assignment operator and having the following form of function definition (here *c_pointer* is a member of the *c_name* class and has the type pointer-to-`type_name`):

   ```
   c_name & c_name::operator=(c_name & cn)
   {
           delete c_pointer;
           c_pointer = new type_name[size];
           // then copy data pointed to by cn.c_pointer to
   ```

```
        // location pointed to by c_pointer
        ...
        return *this;
}
```

USING POINTERS TO OBJECTS

Now that we've fixed initialization and assignment, let's look further at how to use pointers to objects. When you create an object using `new`, `new` returns a pointer to an object. The last two examples used `new` to create an array of objects, and you could use the pointer as if it were an array name. That simplified the notation, but you should be comfortable using pointer notation, too. As we mentioned earlier, you can use the -> operator with object pointer variables just as you use it with structure pointer variables. For instance, suppose `left` is a pointer to a *softball* class object. Then you can invoke the `update()` class method this way:

```
left->update();  // left must point to a softball object
```

As with structures, you use the dot membership operator with the *name* of an object, and you use the arrow membership operator with the *address* of an object.

Also, if `left` is a pointer to a *softball* object, then `*left` is a softball object. For example, to display the contents of the object to which `left` points, use the following statement:

```
cout << *left;  // uses overloaded << for softball objects
```

Listing 10-25 illustrates these points along with how to initialize objects when you use `new` to create them. It uses the same `softbal5.h` header file as the preceding example and should be compiled with the same `softbal5.cpp` methods file.

LISTING 10-25 dynamic3.cpp

```
// dynamic3.cpp -- using the redefined class
// compile with softbal5.cpp
#include <iostream.h>
#include <stdlib.h>
#include "softbal5.h"
int main(void)
{
// use default constructor
    softball * right = new softball;

// use initializing constructor
```

```
    softball * left = new softball("Anton","Bruckner",9,5,3);

// assign anonymous object to *right
    *right = softball("Johannes","Brahms",4,4,2);

// use object-initializing constructor
    softball * tops = new softball(*left);

    cout << *left;
    operator<<(cout,*right);
    cout << "Most Hits " << *tops;

// accessing class methods with pointer to object
    left->update();
    *tops = right->topbat(*left);

    cout << "Top BA " << *tops;

// farewell, team
    delete right;
    delete left;
    delete tops;
    cout << "Well played and farewell!\n";
    return 0;
}
```

Here is a sample run:

```
Player: Anton Bruckner
At bats: 9   Hits: 5   RBIS: 3   BA: 0.555556

Player: Johannes Brahms
At bats: 4   Hits: 4   RBIS: 2   BA: 1

Most Hits Player: Anton Bruckner
At bats: 9   Hits: 5   RBIS: 3   BA: 0.555556

Enter stats for Anton Bruckner:
Enter additional number of at bats: 1
Enter additional number of hits: 0
Enter additional number of RBIs: 0

Top BA Player: Johannes Brahms
At bats: 4   Hits: 4   RBIS: 2   BA: 1

Bye, Johannes
```

```
Bye, Anton
Bye, Johannes
Well played and farewell!
Bye, Johannes
```

Program Notes

This program brings many pointer-related properties into play, so let's analyze it a step at a time. The program begins by allocating space for a *softball* object and assigning its address to the pointer *right*:

```
softball * right = new softball;
```

Because this doesn't explicitly initialize the new object, the program uses the default constructor, which, remember, now initializes the *firstname* and *lastname* members of the new structure to the null pointer.

The next statement shows how to initialize an object when you use `new` instead of a variable definition to create it:

```
softball * left = new softball("Anton","Bruckner",9,5,3);
```

The argument list on the right-hand side matches the signature for one of the *softball* class constructors, so the program uses that constructor to create and initialize the object to which *left* points.

The next program statement is more involved than the first two:

```
*right = softball("Johannes","Brahms",4,4,2);
```

The expression `*right` represents the object to which *right* points. Note that because the `*right` object already exists, the statement is an assignment statement, not an initialization statement. This causes the statement to execute in two steps. First, the function call on the right-hand side evokes the corresponding constructor function, which constructs an object and initializes it using the argument values. This object is a temporary, anonymous object. Second, the program uses the *softball* class overloaded assignment operator to copy the anonymous object to the object pointed to by `right`. That assignment definition, remember, copies the strings themselves rather than just copying pointers.

The next statement in the program is initialization, not assignment, because it creates a new object:

```
softball * tops = new softball(*left);
```

Therefore, this statement invokes the class constructor defined with a *softball* class argument. Note that `*left` is an object, not a pointer.

The next few statements remind you how the overloaded *softball* friend version of << works:

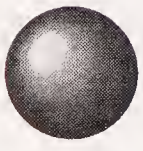
```
cout << *left;
operator<<(cout,*right);
cout << "Most Hits " << *tops;
```

First, `*left` is a *softball* object, so it forms a correct operand for the overloaded << operator. The second statement shows the function call equivalent of using the overloaded operator; this illustrates that the operator notation is a lot nicer looking. The third statement shows how you can use the *ostream* and the *softball* friend versions of << in the same statement.

The next two statements illustrate accessing class methods with pointers:

```
left->update();
*tops = right->topbat(*left);
```

The first call updates the object to which *left* points. The second call invokes the `topbat()` method. The object to which *right* points is the invoking object, and the object to which *left* points is the argument. The return value is assigned to the object to which *tops* points.



If *po* is a pointer to a class object, use **po* in contexts that require an object, and use *po*-> to invoke class member functions.

Next, the statement

```
cout << "Top BA " << *tops;
```

displays the contents of the object **tops*.

Finally, the program deletes the objects it created with *new*. Notice the final output:

```
Bye, Johannes
Bye, Anton
Bye, Johannes
Well played and farewell!
Bye, Johannes
```

The first three *Bye*s come from applying `delete` to the *right*, *left*, and *tops* pointers. When `main()` terminates, the program sheds the temporary variable originally produced when the program assigned values to `*right`. As we mentioned in Chapter 8, implementations have some latitude about when they "forget" such temporary variables, and our implementation apparently treats them like automatic variables, deleting them at the end of the block containing them.

SUMMARY

This chapter covers many important aspects of defining and using classes. Several of these aspects are subtle, even difficult concepts. If some of them seem obscure or unusually complex to you, don't feel bad—they affect most newcomers to C++ that way. Often, the way you come to really appreciate concepts like, say, copy constructors, is through getting into trouble by ignoring them. So some of the material in this chapter may seem vague to you until your own experiences enrich your understanding. Meanwhile, let's summarize the chapter.

Normally, the only way you can access private class members is by using a class method. C++ alleviates that restriction with friend functions. To make a function a friend function, declare the function in the class definition and preface the declaration with the keyword `friend`. A `friend` function can be an independent function or a member function of another class. By declaring an entire class to be a friend class, you grant friend status to all the member functions of that class.

C++ extends overloading to operators by letting you define special operator functions that describe how particular operators relate to a particular class. An operator function can be a class member function or a friend function. C++ lets you invoke an operator function either by calling the function or by using the overloaded operator with its usual syntax. An operator function for the operator `op` has this form:

```
operatorop(argument-list)
```

The argument list represents operands for the operator. If the operator function is a class member function, then the first operand is the invoking object and isn't part of the argument list. For instance, we overloaded addition by defining an `operator+()` member function for the *vector* class. If *up*, *right*, and *result* are three vectors, you can use either of the following statements to invoke vector addition:

```
result = up.operator+(right);
result = up + right;
```

For the second version, the fact that the operands *up* and *right* are type *vector* tell C++ to use the *vector* definition of addition.

When an operator function is a member function, the first operand is the object invoking the function. In the example above, for example, the *up* object is the invoking object. If you want to define an operator function so that the first operand is not a class object, you must use a friend function. Then you can pass the operands to the function definition in whichever order you want.

One of the most common tasks for operator overloading is defining the << operator so that it can be used in conjunction with the `cout` object to display an object's contents. To allow an *ostream* object to be the first operand, define the operator func-

tion as a friend. To allow the redefined operator to be concatenated with itself, make the return type *ostream &*. Here's a general form satisfying those requirements:

```
ostream & operator<<(ostream & os, const c_name & obj)
{
    os << ... ;  // display object contents
    return os;
}
```

C++ lets you establish conversions to and from class types. First, any class constructor taking a single argument acts as a conversion function, converting values of the argument type to the class type. C++ invokes the constructor automatically if you assign a value of the argument type to an object. For instance, suppose you have a *string* class with a constructor that takes `char *` value as its sole argument. Then, if *bean* is a *string* object, you can use the following statement:

```
bean = "pinto";   // converts type char * to type string
```

To convert from a class to another type, you must define a conversion function providing instruction about how to make the conversion. A conversion function must be a member function. If it is to convert to type *typename*, it should have the following prototype:

```
operator typename();
```

Note that it must have no declared return type, must have no arguments, and must (despite having no declared return type) return the converted value. For instance, a function to convert type *vector* to type `double` would have this function form:

```
vector::operator double()
{
    ...
    return a_double_value;
}
```

You can use `new` and `delete` to create and delete class objects. The `new` operator returns a pointer to an object, and you can use the pointer membership operator (`->`) with the pointer to access the class methods. If `pclass` is a pointer to an object, then `*pclass` is an object which can be treated like any other object of the same class.

You can also use `new` in a class constructor to allocate memory for data and then assign the address of the memory to a class member. This enables a class, for example, to handle strings of various sizes without committing the class design in advance to a fixed array size. Using `new` in class constructors also raises potential problems when an object expires. If an object has member pointers pointing to memory allocated by `new`, freeing the memory used to hold the object does not automatically free the memory pointed to by the object member pointers. Therefore, if you use `new` in a class constructor to allo-

cate memory, you should use `delete` in the class constructor to free that memory. That way, the demise of an object automatically triggers the deletion of pointed-to memory.

Objects having members pointing to memory allocated by `new` also have problems with initializing one object to another or assigning one object to another. By default, C++ uses memberwise initialization and assignment, which means that the initialized or the assigned-to object winds up with exact copies of the original object's members. If an original member points to a block of data, the copy member points to the same block. When the program eventually deletes the two objects, the class destructor will attempt to delete the same block of memory twice, which is an error. The solution is to define a special copy constructor that redefines initialization and to overload the assignment operator. In each case, the new definition should create duplicates of any pointed-to data and have the new object point to the copies. That way, both the old and the new object refer to separate, but identical, data with no overlap.

As you may have noticed, classes require much more care and attention to detail than do simple C-style structures. In return, they do much more for you.

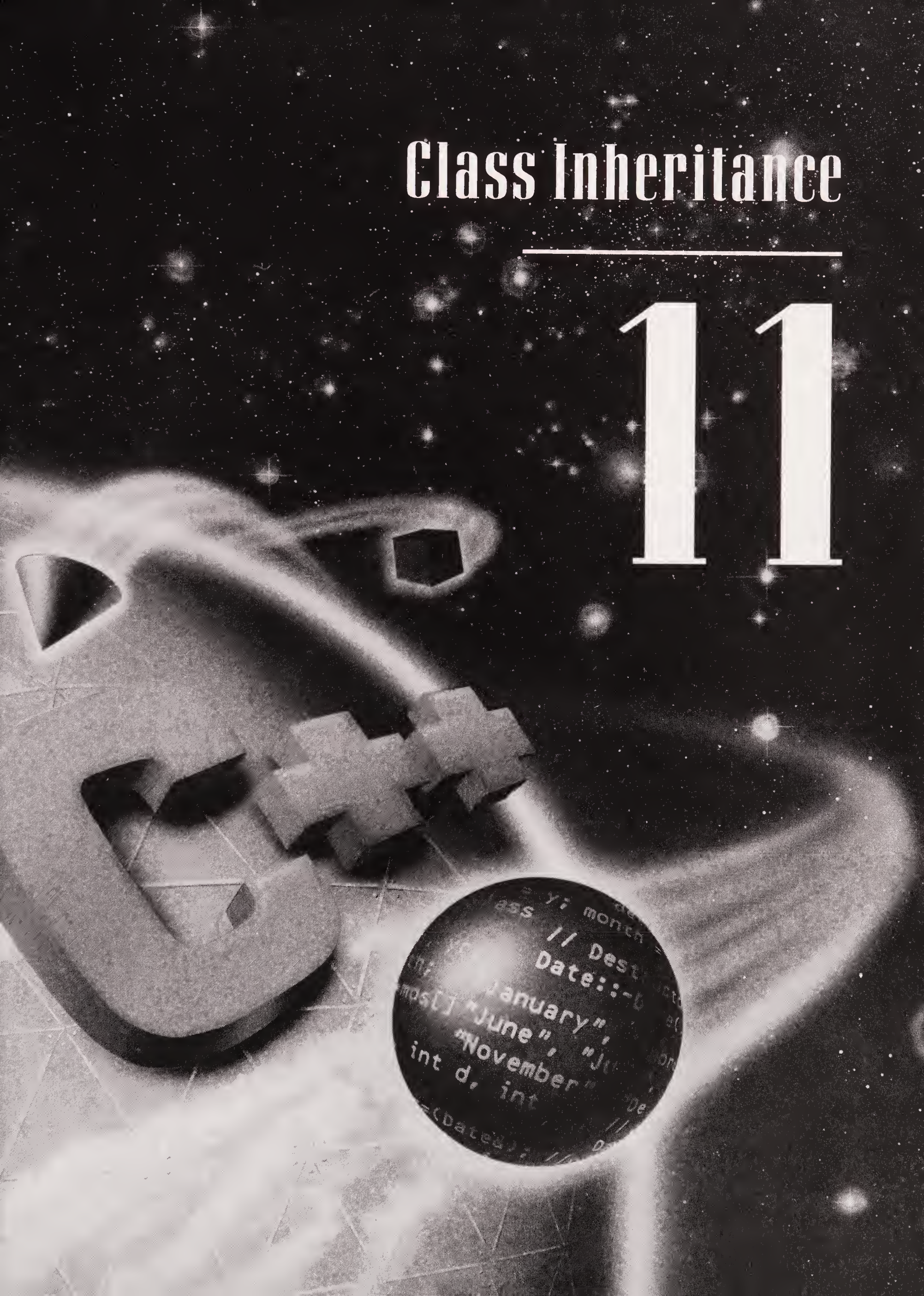
Review Questions

1. What are the differences between a friend function and a member function?
2. Does a nonmember function have to be a friend to access a class's members?
3. Use a member function to overload the multiplication operator for the *diva* class; have the operator multiply the *annual_take* member by a type `double` value.
4. Use a friend function to overload the multiplication operator for the *diva* class; have the operator multiply a type `double` value by the *annual_take* member.
5. Which operators cannot be overloaded?
6. What restriction applies to overloading the following operators?
   `=  ()  []  ->`
7. Overload the `<<` operator so that it can replace the `show_diver()` function in the *diver* class.
8. Rewrite the *stonewt* class so that it has a state member governing whether the object is interpreted in stone form, integer pounds form, or floating-point pounds form. Overload the `<<` operator to replace the `show_stn()` and `show_lbs()` methods.

9. Define a conversion function for the *vector* class that converts a vector to a type `double` value representing the vector's magnitude.
10. Revise the *softball* class definition so that it includes a static class member that keeps track of the number of softball objects.
11. Name three problems that may arise if you define a class in which a pointer member is initialized using `new` and indicate how they can be remedied.
12. Enhance the *string* class definition by removing the `cout` statements from the constructors and the destructor and by doing the following:
    a. Provide a `length()` member function that returns the size of a string.
    b. Replace the `says()` function with a redefined `<<` operator.
    c. Provide a copy constructor.
    d. Overload the assignment operator so that it copies data instead of addresses.
    e. Provide a conversion function to convert a *string* object to a regular C++ string.

Class Inheritance

11



One of the main goals of object-oriented programming is providing reusable code. When you develop a new project, particularly if the project is large, it's nice to be able to reuse proven code rather than to reinvent it. Employing old code saves time and, because it has already been used and tested, can help suppress the introduction of bugs into a program. Also, the less you have to concern yourself with details, the better you can concentrate upon overall program strategy.

Traditional C function libraries provide reusability through predefined, precompiled functions, such as `strlen()`, that you can use in your programs. Many vendors furnish specialized C libraries providing functions beyond those of the standard C library. For example, you can purchase libraries of database management functions and of screen control functions. However, function libraries have a limitation. Unless the vendor supplies the source code for its library functions (and often it doesn't), you can't extend or modify the functions to meet your particular needs. Instead, you have to shape your program to meet the workings of the library. Even if the vendor does supply the source code, you run the risk of unintentionally modifying how part of a function works as you add your changes.

C++ classes bring you a higher level of reusability. Many vendors now offer class libraries, which consist of class definitions and implementations. Because a class encapsulates data representation with class methods, it provides a more integrated package than does a function library. A single class, for example, may provide all the resources for managing a dialog box. Often, class libraries are available in source code, meaning that you can modify them to meet your needs. But C++ has a better method than code modification for extending and modifying classes. This method, called *class inheritance*, lets you derive new classes from old ones, with the derived class inheriting the properties, including the methods, of the old class, called a base class. Just as inheriting a fortune usually is easier than earning one from scratch, deriving a class through inher-



itance usually is easier than designing a new one. You can add new properties to the derived class, and you even can redefine how an inherited class member function should work. You can do all this even if you don't have the source code for the original class's member functions; you just need the class definition. Finally, the classes you derive today can become parent classes to the classes you derive tomorrow.

Inheritance is a splendid concept, and its basic implementation is quite simple. But managing inheritance so that it works properly in all situations requires some tricky adjustments. This chapter looks at both the simple and the subtle aspects of inheritance. It also looks at classes having members that themselves are classes and at multiple inheritance, in which a class can inherit properties of more than one ancestor class. Both techniques let you use existing classes as building blocks for new classes.

An Array Base Class

When one class inherits from another, the original class is called a *base* class (that's base as in foundation, and not as in lowly, vile, or depraved) and the inheriting class is called a *derived* class. So, to illustrate inheritance, we need to begin with a base class. We'll use an array class for that purpose. Perhaps you're wondering what the point is to creating an array class when C++ already has arrays. By defining an array class, you can add features missing from standard C++ arrays. For example, you can make it possible to assign the contents of one array to another with a single assignment statement instead of using a loop to assign each element individually. You can make it impossible to use an array index that's outside the array's bounds. (Nothing in C or C++ prevents you from accidentally using an out-of-range subscript, and that can lead to problems!) Also, you can allow an array's indexing to begin at some value other than zero. You can define operators for adding arrays, subtracting arrays, and so on. You can define class member functions for common operations, such as summing the elements of an array. In short, the array has great upscale potential for the judicious C++ investor.

Once you get into the spirit of array improvement, you probably will come up with more ideas. But don't overreact. It's a poor idea to define some humongous class that implements every conceivable array enhancement. Such a definition would be weighed down with features that were rarely, if ever, used. Not being part of a committee or of a government agency, we can use a better strategy. And that is to define a simple base class, then use class inheritance to add features as needed.

CLASS DEFINITION

Let's begin with a base class modeled after a type `double` array. The first point to decide is how to represent the data. The obvious choice of using a standard type `double` array as a class member works poorly because C++ requires that the number of elements in an array be a compile-time constant. That is, the class would have to use an array with a predetermined number of elements, as in the following:

```
class dumbarray
{
private:
    double arr[20];      // array element has fixed size
...
}
```

To represent arrays of two different sizes, you'd need two different classes, making this a poor design choice. A better idea is to incorporate a dynamic array allocated with `new`. Recall that you can use the `new` operator to return the address of the first element of a dynamic array. One class member can store that address, while a second class member stores the number of elements. That way a single class can represent any valid size of array:

```
class smartarray
{
private:
    double * arr;         // address of first element
    unsigned int size;    // variable number of array elements
...
}
```

Listing 11-1 shows a class definition based on this approach. The class name *darray* is short for *d*ouble *array*.

LISTING 11-1 darray.h

```
// darray.h -- define array class
class darray
{
protected:                                  // new keyword
    double * arr;                           // address of first element
    unsigned int size;                      // number of array elements
public:
    darray();                               // default constructor
    darray(unsigned int n, double val = 0.0);
    darray(const double *pn, unsigned int n);
    darray(const darray & a);               // copy constructor
```



```
    ~darray();                              // destructor
    unsigned int arsize();                  // returns array size
// overloaded operators
    double & operator[](int i);             // array indexing
    darray operator*(const darray & a);
    darray & operator=(const darray & a);
    friend ostream & operator<<(ostream & os, const darray & a);
};
```

As usual, the class definition encapsulates data and methods into a single package. The `public` section of the class definition provides prototypes for the various class member functions, including operator functions, and for a friend function overloading the << operator. And the `protected` section provides the data members—an `arr` member to hold the address of the array of data and a `size` member to indicate the number of elements. See Figure 11-1.

PROTECTED AND PRIVATE

Wait a moment—where did that `protected` come from? Up to now the examples have used `private` to hide data. But classes intended to serve as base classes often use `protected` instead of `private`. The `protected` keyword is like `private` in that a program can access class members in a `protected` section only by using class members. The difference between `private` and `protected` comes into play only with classes derived from the base class. Members of a derived class can access `protected` members of a base class, but they cannot access `private` members of the base class. Let's see what that means. Suppose you derive a class from *darray*. Then the new class will inherit the `arr` and `size` members along with class member functions like `arsize()`. Because `arr` and `size` are `protected`, any member functions added to the derived class will be able to use them. If, on the other hand, `arr` and `size` had been `private`, new member functions could not use them. But the old member functions inherited from *darray*, such as `arsize()`, could use `arr` and `size`.

Most often, you'll want the new member functions added to a derived class to be able to access the inherited members. Thus you most often will use `protected` instead of `private`. Using `protected` instead of `private` is one of the two common adjustments for making a class suitable to be a base class. The other adjustment is using virtual functions, and we'll get to that later in this chapter.



A class intended to be a base class usually should use `protected` instead of `private` members.

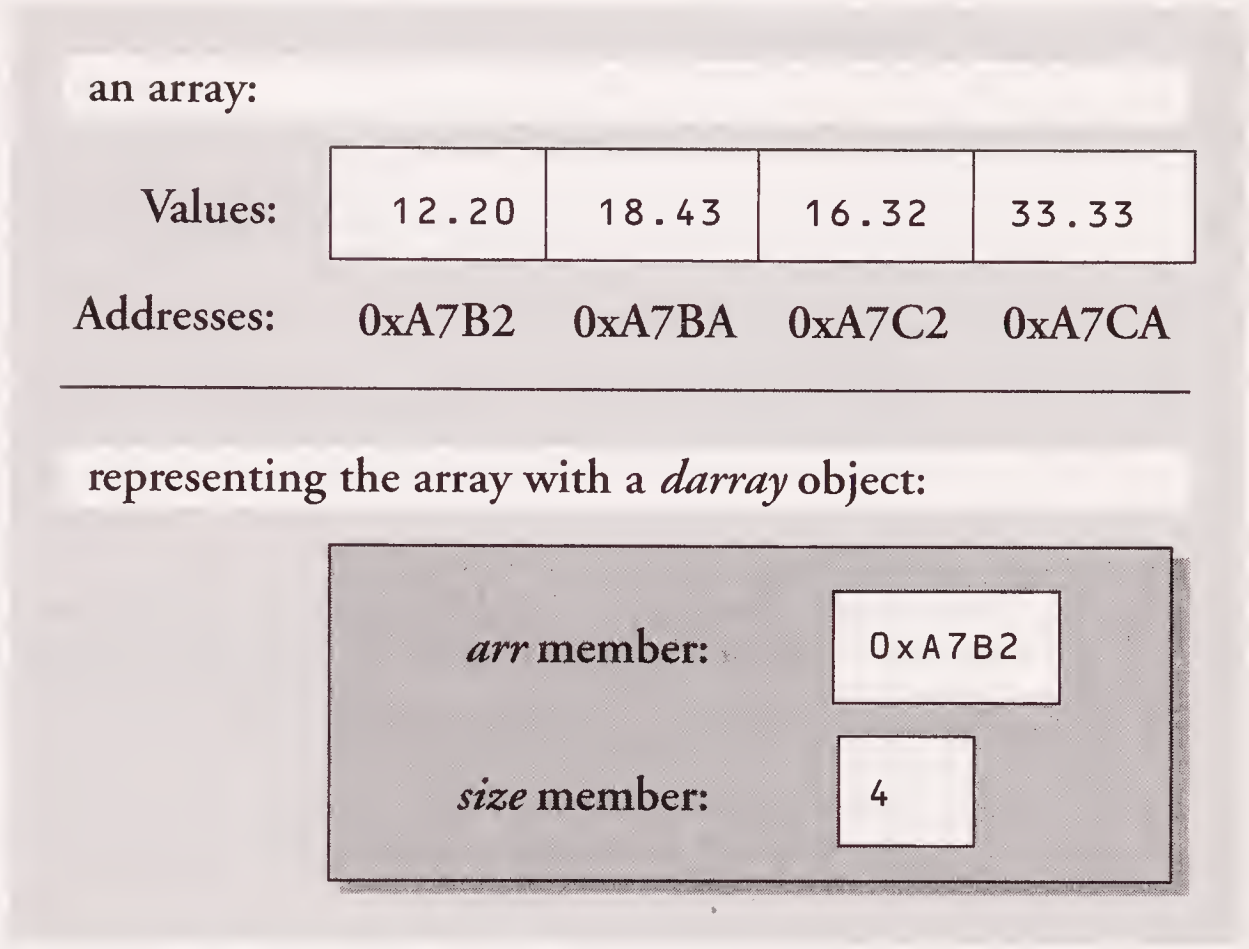


Figure 11-1 *The* darray *class data representation*

The Class Methods

The class definition follows the `protected` data elements with several constructors, a destructor, a method that returns the array size, and some overloaded operators. Note that the first operator function overloads `[]`, the index operator. Remember, the data is protected, so you can access it only through class member functions, not by using regular array notation. But overloading the `[]` operator makes using that operator function to access elements inside a *darray* object look just like using the usual array notation. We'll discuss this point more fully in a moment or two. Listing 11-2 shows the code for these methods.

LISTING 11-2 darray.cpp

```
// darray.cpp -- darray class methods
#include <iostream.h>
#include <stdlib.h>      // exit() prototype
#include "darray.h"

// default constructor -- no arguments
darray::darray()
{
    arr = NULL;
```



```
    size = 0;
}

// constructs array of n elements, each set to val
darray::darray(unsigned int n, double val)
{
    arr = new double[n];
    size = n;
    for (int i = 0; i < size; i++)
            arr[i] = val;
}

// initialize darray object to a nonclass array
darray::darray(const double *pn, unsigned int n)
{
    arr = new double[n];
    size = n;
    for (int i = 0; i < size; i++)
            arr[i] = pn[i];
}

// initialize darray object to another darray object
darray::darray(const darray & a)
{
    size = a.size;
    arr = new double[size];
    for (int i = 0; i < size; i++)
            arr[i] = a.arr[i];
}

darray::~darray()
{
    delete arr;
}

// return array size
unsigned int darray::arsize()
{
    return size;
}

// let user access elements by index
double & darray::operator[](int i)
{
    return arr[i];
}

// multiply two darry objects element by element
darray darray::operator*(const darray & a)
```

```
{
    if(size != a.size)
    {
            cout << "Array multiplication error: arrays must "
                            "be of same size. Bye.\n";
            exit(1);
    }
    darray product(size);
    for (int i = 0; i < size; i++)
            product[i] = arr[i] * a.arr[i];
    return product;
}

// define class assignment
darray & darray::operator=(const darray & a)
{
    if (this == &a)       // if object assigned to self,
        return * this;    // don't change anything
    delete arr;
    size = a.size;
    arr = new double[size];
    for (int i = 0; i < size; i++)
            arr[i] = a.arr[i];
    return *this;
}

// quick output, 5 values to a line
ostream & operator<<(ostream & os, const darray & a)
{
    for (int i = 0; i < a.size; i++)
    {
            os << a.arr[i] << " ";
            if (i % 5 == 4)
                    os << "\n";
    }
    if (i % 5 != 0)
            os << "\n";
    return os;
}
```

The Class Methods

Let's take a quick look at the class methods to see what they do. Not only will this acquaint you with the class, it will review some of the class properties you've been studying in the preceding two chapters.

First, notice that the class constructors (except for the default constructor) use `new` to allocate memory for the array. That, after all, is the basic idea that gives this class



the ability to accommodate different array sizes. But, as Chapter 9 illustrated, when you use `new` in a class constructor, you have to support that use with several other design features. It's the old story of how power brings with it the responsibility to use it correctly. In this case, using `new` requires the following features:

1. The class destructor should free memory allocated by `new` in the constructors.
2. If any one constructor uses `new` to assign an address to a class member, all constructors should assign a valid address to that member, either by using `new` or by assigning the `NULL` pointer. These are the only kinds of addresses that can be used correctly with `delete`; it's okay to apply `delete` to `NULL`, but not to an uninitialized pointer.
3. There should be a special constructor for initializing an object to an object of the same class, and there should also be an operator function for assigning one object to another. Both functions should copy data instead of merely copying addresses.

Fortunately, Chapter 9 showed how to do these very tasks. Now let's look at the methods individually.

Constructors and Destructor

The default constructor initializes the `arr` pointer to `NULL` so that the destructor will work correctly:

```
darray::darray()
{
    arr = NULL;
    size = 0;
}
```

This, recall, is the constructor used if you declare a *darray* object without explicitly initializing it.

The second constructor lets you initialize a *darray* object to a particular number of elements and initialize each element to a particular value:

```
// constructs array of n elements, each set to val
darray::darray(unsigned int n, double val)
{
    arr = new double[n];
    size = n;
    for (int i = 0; i < size; i++)
            arr[i] = val;
}
```

For instance, the declaration

```
darray ofsun(365, 1.0);
```

creates a *darray* object called *ofsun* having 365 elements, and sets each element to 1.0. The class prototype provides a default value of `0.0` for the second argument, so you can call the constructor using a single argument:

```
darray nged(40); // 40 elements, each set to 0.0
```

Don't forget that you can use a pointer as if it were an array name. Of course, you can ignore that fact and use `*(arr + i)` instead of `arr[i]`.

The third constructor lets you initialize a *darray* object to a regular array:

```
// initialize to a nonclass array
darray::darray(const double *pn, unsigned int n)
{
    arr = new double[n];
    size = n;
    for (int i = 0; i < size; i++)
            arr[i] = pn[i];
}
```

This is useful, for if other parts of a program use a regular C++ array, you can use this constructor to create a *darray* object equivalent. To use this constructor, pass it the name of an array (more generally, the address of an array) and the number of elements. The constructor creates a *darray* object with the same number of elements, then copies the original array into the object. Here's a sample use:

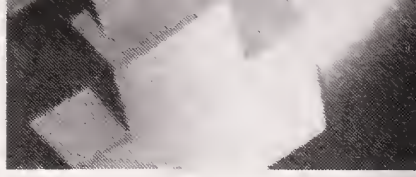
```
double gas_costs[4] = {11.84, 12.43, 18.39, 22.74};
darray gas_bills(gas_costs, 4);
```

The first declaration creates an ordinary array of four elements and initializes them to the values in the brackets. The second declaration creates a *darray* object containing a copy of this array. Incidentally, this example shows a second important use for this constructor. The *darray* class does not have a mechanism for initializing a *darray* object to a set of values enclosed in braces. That is, you can't do the following:

```
darray beads(4) = { 1.2, 2.3, 3.4, 4.5};      // won't work
```

But you can initialize a *darray* object to a set of numbers in two steps, as the `gas_bill` example shows: create and initialize a regular array, then create a *darray* object and initialize it to the array.

The final constructor is the special copy constructor used to initialize one *darray* object to another:



```
// initialize to a class array
darray::darray(const darray & a)
{
    size = a.size;
    arr = new double[size];
    for (int i = 0; i < size; i++)
            arr[i] = a.arr[i];
}
```

As you may recall, a program calls a copy constructor in three circumstances:

- when you initialize one *darray* object to another
- when you pass a *darray* object by value to a function
- when a function returns a *darray* object

If you omit a copy constructor, C++ uses memberwise copying in these three situations. If the object contains a pointer, this default method copies the pointer, not the data, and that leads to the memory-management problems detailed in Chapter 10. But this wonderful copy constructor overrides the default initialization, provides a copy of the data, and thus sidesteps those troublesome memory problems.

The last three constructors allocate memory with `new`, so the destructor must free that memory by using `delete`:

```
darray::~darray()
{
    delete arr;
}
```

Other Methods of the darray Class

The class member functions do more than let you create and destroy arrays, enjoyable though that may be in itself. The other methods, which include operator functions and a friend function, return the array size, provide a means to access individual elements of a *darray* object, let you multiply one array by another, define object assignment, and provide an output function. Let's look at each now.

The `arsize()` member returns the number of elements in the array:

```
// return array size
unsigned int darray::arsize()
{
    return size;
}
```

This brings us to the overloaded operators. The first lets you access *darray* elements using array notation:

```
// let user access elements by index
double & darray::operator[](int i)
{
    return arr[i];
}
```

Let's investigate why this function is necessary and how it works. First, why is it necessary? Suppose *funds* is a *darray* object. Then *funds.arr* is the member pointer to the array storage, and `funds.arr[i]` is the element with index `i`. But `arr` is a `protected` member, so a client program cannot access the data directly with this notation. Making the data completely inaccessible takes data hiding a bit too far. Of course, member functions and friend functions can access the data, so to access an individual element, you must define a member function that does that job. The most direct way is to have the member function take an array element index as an argument and to return the corresponding element value:

```
// let user access elements by index-- the direct way, but
//                                     not good enough for us
double darray::value(int i)
{
    return arr[i];
}
```

Then, say, `funds.value(3)` would represent the value of the fourth element of the array represented by the *funds* object. (Element numbering starts at 0, remember.) But this approach lacks style; for one thing, `funds.value(3)` doesn't look like an array element. Wouldn't you rather write `funds[3]` than `funds.value(3)`?

There's an elegant way to satisfy this wish. First, note that the `[]` in `funds[3]` really constitute an operator. This operator takes two operands, `funds` and `3`, and returns the value stored at the address `funds+3`. Since `[]` is an operator, you can overload it with an operator function that returns the element value. That's what the `operator[]()` definition does. Suppose, then, you use the *funds* object in the following statement:

```
double abscond = funds[3];
```

The compiler will replace this with the corresponding operator function:

```
double abscond = funds.operator[](3);
```

Because the *funds* object invokes the method with an argument of `3`, the method's return statement

```
return arr[i];
```

is interpreted as follows:

```
return funds.arr[3];
```



In short, operator overloading makes the expression `funds[3]` represent an element in the *funds* object.

Note that the operator function is of type `double &`, that is, it's a reference. This is vital because you can assign a value to a function that returns a reference. The effect, as you may recall from Chapter 8, is assigning to the referred-to variable. This feature lets you use statements like the following:

```
funds[2] = 3432.23;
```

This looks reasonable in form, but remember that this statement really means the following:

```
funds.operator[](2) = 3432.23;
```

And that syntax is possible only when a function returns a reference.

One more point: the `operator[]()` function only defines element access through array notation. Redefining the `[]` operator does not redefine pointer addition and the dereferencing operator (`*`):

```
double whammy[5];
//  whammy[2] is the same as *(whammy + 2)
darray grammy[5];
// *(grammy + 2) is not even defined
```

So if you are the sort of programmer that enjoys replacing expressions like `whiffle[10]` with `*(whiffle + 10)`, you can't do so with a *darray* object.

The next operator defines a form of array multiplication:

```
// multiply two arrays element by element
darray darray::operator*(const darray & a)
{
    if(size != a.size)
    {
            cout << "Array multiplication error: arrays must "
                            "be of same size. Bye.\n";
            exit(1);
    }
    darray product(size);
    for (int i = 0; i < size; i++)
            product[i] = arr[i] * a.arr[i];
    return product;
}
```

What does this function do, and how does it work? Many situations require you (or your program) to multiply each element in one array by the corresponding elements of another. For instance, you could have one array representing the numbers of various hat styles sold and a second array representing the price of each style. To get the dollar sales

figures for each item, you have to multiply each sales element by the corresponding price element to produce a new array of dollar sales for each style. In other words, a program has to do something like this:

```
double items_sold[10], prices[10], sales[10];
...
for (int i = 0; i < size; i++)
    sales[i] = item_sold[i] * prices[i];
}
```

The `operator*()` function makes writing this operation simpler and clearer, while hiding the details. For instance, suppose you represent two arrays with the *darray* objects *items_sold* and *prices*. Then the `operator*()` function lets you do the following to create a `sales` array:

```
darray sales = items_sold * prices;
```

What could be simpler?

The process makes sense only for arrays of the same size, so the function aborts the program by calling the `exit()` function if you try to multiply two arrays of different sizes. This method is a good example of how defining a class lets you augment the power of an array.

Now we have just two more functions to go. The first overloads the assignment operator so that it copies the pointed-to data rather than the pointers themselves:

```
// define class assignment
darray & darray::operator=(const darray & a)
{
    delete arr;
    size = a.size;
    arr = new double[size];
    for (int i = 0; i < size; i++)
            arr[i] = a.arr[i];
    return *this;
}
```

The definition follows the pattern developed in Chapter 10 for overloading assignment for classes containing pointers:

- The function argument is an object passed by reference (type *darray &*).
- The function returns a reference to a new object, which is a copy of the one passed as an argument.
- The function uses `delete` to free the memory to which *arr* points. This clears the old data, preparing `arr` to point to new data.
- The function allocates space for the data being copied, assigns the address to `arr`, then copies the data from the old array to the new.



In case you've forgotten why the assignment operator returns a value, it's to make it resemble ordinary assignment. Recall that an assignment expression such as

```
x = y
```

has the value of the left-hand side if *x* is a standard type such as `int`. But if `x` and `y` are *darray* objects, then that expression becomes the following function call:

```
x.operator=(y)
```

The definition above causes the operator function to return a reference to `x`, so this expression, too, has the value of the left-hand side of the original expression `x = y`.

The final method displays the array contents:

```
// quick display, 5 values to a line
ostream & operator<<(ostream & os, const darray & a)
{
    for (int i = 0; i < a.size; i++)
    {
            os << a.arr[i] << " ";
            if (i % 5 == 4)
               os << "\n";
    }
    if (i % 5 != 0)
            os << "\n";
    return os;
}
```

The method uses the modulus operator to display five values per line. The `if` statement at the end adds a newline if there are fewer than five items on the final line. If you want to organize the information differently, you can use the index operator function to access the elements individually. But if you just want to see what's in the array, use this method.

USING THE CLASS

Let's be prudent and test this class in a program before trying to derive other classes from it. Listing 11-3 provides a short program for that purpose. The program follows the usual C++ pattern for organization: a header file containing the class definition (Listing 11-1), a methods file containing the member and friend function definitions (Listing 11-2), and a program that uses the class (Listing 11-3). Therefore you should use whatever technique your implementation requires for a multiple-file program. Chapters 1 and 8 provided some general guidelines.

LISTING 11-3 arr_cls.cpp

```
// arr_cls.cpp -- use the darray class
// compile with darray.cpp
#include <iostream.h>
#include "darray.h"
int main(void)
{
    cout << "Enter number of regions: ";
    unsigned int regions;       // read number of regions
    cin >> regions;

    darray tons(regions);       // create an "array" of that size
    cout << "Enter the regional tinsel sales in tons:\n";
    for (int i = 0; i < regions; i++)
    {
            cout << "Region " << i+ 1 << ": ";
            cin >> tons[i];     // use array notation
    }

    darray toncosts(regions);
    cout << "Enter the regional tinsel costs per ton "
              "in dollars:\n";
    for (i = 0; i < regions; i++)
    {
            cout << "Region " << i + 1 << ": ";
            cin >> toncosts[i];
    }

    darray costs = tons * toncosts;  // array *, assignment
    cout << "Total expenditures:\n";
    cout << costs << "(all in dollars)\n";

    return 0;
}
```

```
Enter number of regions: 3
Enter the regional tinsel sales in tons:
Region 1: 213.3
Region 2: 248.1
Region 3: 222.2
Enter the regional tinsel costs per ton in dollars:
Region 1: 149.99
Region 2: 119.99
Region 3: 139.99
Total expenditures:
31992.867 29769.519 31105.778
(all in dollars)
```



Notes on Using the darray Class

Let's look at some of the main points about using the *darray* class. Because it's possible to establish the number of elements in a *darray* object at run time, you can enter the number of elements from the keyboard and then create a *darray*-style array of that size:

```
cin >> regions;
darray tons(regions);
```

Thus you get the flexibility of dynamic memory combined with the ease of use of an ordinary array; the details about using `new` are hidden from the user.

The next nice touch is that the `operator[]()` function lets you use array notation to access individual elements:

```
cin >> tons[i];
cin >> toncosts[i];
```

This makes using the *darray* object as simple as using an ordinary array.

The most complex operation that the program undertakes is multiplying each element of the `tons` array by the corresponding element of the `toncosts` array, but the code makes the process look simple:

```
darray costs = tons * toncosts;
```

Behind this simple facade, however, there is much activity! The right side uses the overloaded multiplication operator to multiply each element in the `tons` object by the corresponding element in the `toncosts` array, assigning the results to the local *product* object in the operator function. The `operator*()` function then returns this object to `main()`, which initializes the `costs` array to the returned object. That means the program calls the special constructor that initializes one object to another.

Finally, the statement

```
cout << costs << "(all in dollars)\n";
```

displays the contents of the new array.

Deriving a Class

Now that *darray* has joined the ranks of the working class, let's derive a new class from it. Suppose you find you're working on a project that does a lot of arithmetic with arrays—adding two arrays (combining inventories), subtracting an array from another (subtracting commissions), dividing the elements of one array by the elements of another (getting advertising costs per capita), multiplying every value in an array by the

same number (across the board price increase), and so on. You'd like to use the *darray* class, but it doesn't support all these operations. What you can do is define a new class that inherits all the properties of the *darray* class and that additionally has all the new arithmetic methods you need. By basing the new class on *darray* instead of starting from scratch, you take advantage of all the work that already has gone into developing *darray*, and you take advantage of the fact that *darray* has already been tested. In other words, you reuse proven code. The net result is less work and, perhaps, a better product.

DEFINING A DERIVED CLASS

A derived class has to identify the class from which it is derived. The C++ way is to include the base class name in the derived class definition. Suppose, for example, you call the derived class *arithdar* (*arith*metic *d*ouble *ar*ray). Then you can begin the class definition like this:

```
class arithdar : public darray
{
```

The colon indicates the *arithdar* class is based upon the *darray* class. This particular heading indicates that *darray* is a `public` base class; this is termed *public derivation.* That, in turn, means that the derived class can access the `public` and `protected`, but not the `private`, members of the base class. The `public` members of the base class become `public` members of the derived class, and the `protected` members of the base class become `protected` members of the derived class. For instance, the *darray* `operator[]()` function becomes a public function of the *arithdar* class, too, while the `arr` member of the *darray* class becomes a protected member of the *arithdar* class. In short, the *arithdar* class inherits `public` and `protected` members from the base class. You don't have to redefine them for the new class. A derived class does not inherit the *private* members of a base class. Table 11-1 summarizes how member visibility carries over to a derived class with `public` derivation.

Table 11-1 Visibility of inherited class members

Base Class Member Visibility	Derived Class Member Visibility, *public* Inheritance	Derived Class Member Visibility, *Private* Inheritance
`public`	`public`	`private`
`protected`	`protected`	`private`
`private`	not inherited	not inherited



You can also use `private` instead of `public` when deriving a class:

```
class gradeschool : private school
{
```

This is termed *private derivation*. In that case, `public` and `protected` members of the base class become `private` members of the derived class, meaning they can be accessed only through member functions of the derived class. Also, being `private`, these members cannot be inherited by classes based upon the derived class. Table 11-1 summarizes how member visibility carries over to a derived class with `private` derivation. We won't be using this form of derivation.

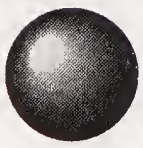
Once you derive a class, you can add new members to it. In fact, you *must* provide new constructors. That's because the name of a constructor matches the name of the class:

```
darray wewere;          // needs darray() constructor
arithdar redford;       // needs arithdar() constructor
```

When you create an object of a derived class, the program first calls the constructor for the base class, then the constructor for the derived class. This makes sense because the constructor for the derived class may build upon data members from the base class; hence the base class object has to be constructed first. It's a bit like building the first floor of a building before adding the second story. Thus when you define the new constructors, they shouldn't duplicate the work of the base constructors. Instead, they should handle just the additional details that the derived class requires. For example, the constructor could initialize new data members.

A base class constructor is paired with the derived class constructor having the same signature, that is, the same argument list. For instance, evoking the default constructor for an *arithdar* object also evokes the default constructor for a *darray* object. And evoking the *arithdar* constructor with a *darray* argument also evokes the *darray* constructor with a *darray* argument.

You don't have to add a new destructor unless the new class requires clean-up work beyond that performed by the base destructor. When an object expires, the program first calls the derived destructor, if any, then the base destructor. The new class doesn't need a destructor, but we'll provide one anyway so that we can examine how derived classes handle destructors.



When creating an object of a derived class, a program first calls the base class constructor, then the derived class constructor. When an object of a derived class expires, the program first calls the derived class destructor, if any, then the base class destructor.

Okay, now you know how to start the definition of a derived class:

```
class arithdar : public darray
{
```

Now let's add some new class methods. Since our main goal is demonstrating how class derivation operates and not on developing the most complete possible class definition, we'll implement just a couple new methods to show how the process works. Listing 11-4 shows the class definition. It only adds `public` member functions and no new data members. However, in general, you can add both methods and data members, and each can be `public`, `protected`, or `private`. Note that you don't have to do a thing to the base class to derive a class. All the derivation takes place in defining the new class and its methods. Thus you can derive a class even if you don't have access to the source code for the base class.

LISTING 11-4 arithdar.h

```
// arithdar.h -- derived array class with more arithmetic
#include "darray.h"

class arithdar : public darray  // derived from darray class
{
public:
// must define constructors having the name of the new class
    arithdar();
    arithdar(unsigned int n, double val = 0.0);
    arithdar(const double *pn, unsigned int n);
    arithdar(const arithdar & a);
    arithdar(const darray & a);

// destructor is inherited, but you can define a new one
    ~arithdar();

// new methods
    double sum();
    double average();

// we leave the definition of overloaded operators, such
// as addition, subtraction, sign reversal, multiplication
// by a single value, etc., as an exercise for the reader
// in the review questions at the end of the chapter
};
```



IMPLEMENTING THE DERIVED CLASS

Next, let's examine how to implement the derived class and look at the rationale for some of the methods, beginning with the constructors. Their design has an additional goal beyond implementing the class correctly: they demonstrate how a program handles constructors for derived classes. To help you see when each constructor is in use, each constructor displays a short message announcing when it has been called.

The default constructor for *arithdar* needn't do a thing. The *arithdar* class has the same data members as the *darray* class (the `arr` pointer and the `size` member), and that means you need a constructor to allocate the memory to which `arr` points and to set the value of `size`. But, because a program always calls the base constructor before calling the derived class constructor, the *darray()* default constructor takes care of initializing those members. All the *arithdar* default constructor has to do is exist:

```
arithdar::arithdar()
{
}
```

Since it does nothing, does it have to exist? Yes, because if you omit it, the *darray* default constructor doesn't get called either.

Although the default *arithdar* constructor need do nothing, we'll define the constructor this way:

```
arithdar::arithdar()
{
    cout << "default arithdar constructor\n";
}
```

That way, when you run a program, you can see when this constructor goes into action.

The next constructor has this prototype:

```
arithdar(unsigned int n, double val = 0.0);
```

This constructor creates a dynamic array with `n` elements and initializes each element to `val`. That is, the statement

```
arithdar lear(3, 2.0);
```

creates an *arithdar* object with three array elements, each set to the value `2.0`, just like the equivalent *darray* constructor did for *darray* objects. But there is a problem here, and it's not an obvious one. Recall that invoking the `arithdar()` constructor first invokes the corresponding *darray* constructor, that is, the *darray* constructor with the same signature. The `darray()` constructor is the one that initializes `arr` and `size`, so it is the constructor that should receive the argument list (`3` and `2.0`). But they are passed to the `arithdar()` constructor instead! The problem is getting the arguments to the

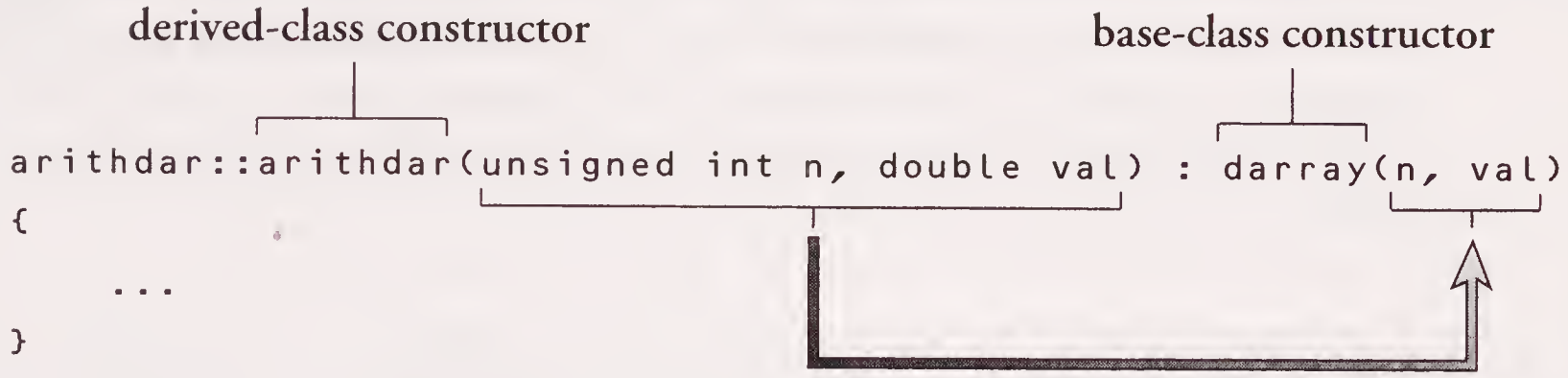
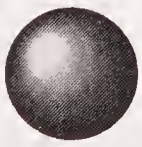


Figure 11-2 *Passing arguments through to a base class constructor*

`darray()` constructor so that it can do its job. The C++ solution is a special argument-passing mechanism you can use with class methods. Here's how it looks:

```
arithdar::arithdar(unsigned int n, double val) : darray(n, val)
{
    cout << "unsigned, double arithdar constructor\n";
}
```

The colon in the function heading means pass the `arithdar()` arguments `n` and `val` to the `darray()` function. The program will pass the arguments and execute the `darray()` function *before* executing any statements in the body of the `arithdar()` function. Thus the derived constructor can assume that the base constructor has completed its work before the derived constructor begins. This is a special case of a more general initialization process for class methods. We'll look at the general case later in this chapter. Meanwhile, we'll use this procedure to pass arguments to base constructors. See Figure 11-2. Again, the constructor reports when it was called.



A constructor for a derived class must use a special argument-passing mechanism to pass those arguments required by the corresponding base class constructor.

```
derived::derived(type1 x, type2 y) : base(x,y)
{
    ...
}
```



Here *derived* is the derived class, *base* is the base class, and `x` and `y` are variables used by the base class constructor. If, say, the *derived* constructor is passed the arguments 10 and 12, this mechanism then passes 10 and 12 on to the *base* constructor. This form appears in the function definition, but not in the prototype.

You can use the same technique for the remaining constructors. Just pass the constructor arguments to the corresponding base class constructor. Also, we'll have each type report when it's been used. Listing 11-5 shows the rest of the definitions.

Initializing Objects to Objects

Two of the constructors let you initialize one object to another, and they merit further attention. First, consider the following prototype:

```
arithdar(const darray & a);
```

It lets you initialize an *arithdar* object to a *darray* object. Because both classes have the same data representation, this makes sense. It is also useful. First, by initializing an *arithdar* object to a *darray* object, you make it possible to apply the *arithdar* methods to data previously placed in a *darray* object. Second, recall that any constructor with exactly one argument defines a type conversion from the argument type to the class type. Thus the `arithdar(const darray & a)` function defines *darray* to *arithdar* conversion. This means that you can do the following:

```
darray motels(10);
arithdar inns;
inns = motels;              // converting darray to arithdar
```

The last line uses the constructor to create a temporary and anonymous *arithdar* object initialized to the `motels` value. Then the contents of that temporary object are copied into `inns`.

The last example may make you wonder if you have to redefine assignment for the *arithdar* class. No, you don't have to. The *arithdar* class uses the same data representation as the *darray* class, so it can use the *darray* assignment operator. Recall that the *darray* class specification redefined assignment for the *darray* class so that it made a copy of the original array. The expression `inns = hotels` will use that assignment function even if *inns* and *hotels* are type *arithdar*—another bonus from inheritance. However, if the *arithdar* had *additional* data members, then you *would* have to redefine assignment. In that case, the new definition would override the base class definition.

A derived class uses the base class methods unless the derived class provides a replacement method of the same name.

What about the constructor that initializes one *arithdar* object to another?

```
arithdar(const arithdar & a);
```

You probably recall that invoking a derived constructor first invokes the corresponding base class constructor. What's the corresponding base class constructor here? After all, no *darray* constructor had an *arithdar* `&` argument. The answer lies in the fact that a derived class object automatically is converted to a base class object when used as an argument to a base member function. So the type *arithdar* `&` argument becomes a *darray* `&` argument for a *darray* constructor, meaning the `darray(const darray & a)` constructor is called.

A derived class object is converted to a base class object when used as an argument to a base class member function.

Next, you don't have to define an *arithdar* destructor because the base destructor already frees the memory to which `arr` points. But let's define it so that you can monitor when it's used:

```
arithdar::~arithdar()
{
    cout << "arithdar destructor\n";
}
```

Defining the `sum()` and `average()` methods for the derived class raises no new issues. Listing 11-5 shows the definitions for all the new methods. Note that aside from the default constructor, all the constructors use the argument-passing mechanism to pass arguments to the corresponding *darray* constructors.

LISTING 11-5 arithdar.cpp

```
// arithdar.cpp -- arithdar class methods
#include <iostream.h>
#include <stdlib.h>                    // exit() prototype
#include "arithdar.h"

arithdar::arithdar()
{
```



```
    cout << "default arithdar constructor\n";
}

arithdar::arithdar(unsigned int n, double val) : darray(n, val)
{
    cout << "unsigned, double arithdar constructor\n";
}

arithdar::arithdar(const double *pn, unsigned int n)
    : darray(pn, n)
{
    cout << "double *, unsigned arithdar constructor\n";
}

arithdar::arithdar(const arithdar & a) : darray(a)
{
    cout << "arithdar & arithdar constructor\n";
}

arithdar::arithdar(const darray & a) : darray(a)
{
    cout << "darray & arithdar constructor\n";
}

arithdar::~arithdar()
{
    cout << "arithdar destructor\n";
}

// new methods
double arithdar::sum()
{
    double total = 0.0;
    for (int i = 0; i < size; i++)
            total += arr[i];
    return total;
}

double arithdar::average()
{
    if (size < 1)
    {
            cout << "Computing an average requires at least "
                            "one array element -- bye.\n";
            exit(1);
    }
    return sum() / size;
}
```

USING THE DERIVED CLASS

Listing 11-6 modifies Listing 11-3 to use and test the derived class. When compiling the program, you need to compile three files: `derived.cpp`, `arithdar.cpp`, and `darray.cpp`. The `arithdar.cpp` file provides the new methods, and the `darray.cpp` file provides the inherited methods. Actually, to allow tracing the constructor and destructor calls, we replaced `darray.cpp` with `darray_v.cpp`, a verbose version that uses the same sort of announcement messages that are incorporated into `arithdar.cpp`. That is, the new function definitions replace

```
darray()
{
    arr = NULL;
    size = 0;
}
```

with

```
darray()
{
    arr = NULL;
    size = 0;
    cout << "default darray constructor\n";
}
```

and so on.

LISTING 11-6 derived.cpp

```
// derived.cpp -- use a derived class
// compile with arithdar.cpp and darray_v.cpp
#include <iostream.h>
#include <stdlib.h>                    // exit() prototype
#include "arithdar.h"
int main(void)
{
    cout << "Enter number of regions: ";
    unsigned int regions;
    cin >> regions;
    if (regions < 1)
    {
            cout << "That's no fun -- bye\n";
            exit(0);
    }

    cout << " \nExecuting arithdar tons(regions); now:\n";
    arithdar tons(regions);
```



```
    cout << "Enter the regional tinsel sales in tons:\n";
    for (int i = 0; i < regions; i++)
    {
            cout << "Region " << i+ 1 << ": ";
            cin >> tons[i];
    }
    cout << "Total tons: " << tons.sum() << "\n";

    cout << " \nExecuting arithdar toncosts(regions); now:\n";
    arithdar toncosts(regions);
    cout << "Enter the regional tinsel costs per ton "
                "in dollars:\n";
    for (i = 0; i < regions; i++)
    {
            cout << "Region " << i + 1 << ": ";
            cin >> toncosts[i];
    }
    cout << "Average cost per ton = "
        << toncosts.average() << "\n";

    cout << " \nExecuting "
                    "arithdar costs = tons * toncosts; now:\n";
    arithdar costs = tons * toncosts;
    cout << "Regional expenditures:\n";
        cout << costs << "(all in dollars)\n";
        cout << "Total expenditure = $" << costs.sum() << "\n";

        return 0;
    }
```

With both derived class and base class constructors being called in several different contexts, there's a lot going on. When you examine the output, you'll see that more than half the output is devoted to tracking what's happening behind the scenes. Pay particular attention to which constructors are called and when they are called:

```
Enter number of regions: 3

Executing arithdar tons(regions); now:
unsigned, double darray constructor      ←first call base constructor for tons
unsigned, double arithdar constructor    ←then call derived constructor
Enter the regional tinsel sales in tons:
Region 1: 220
Region 2: 320
Region 3: 260
Total tons: 800                          ←calculated by arithdar method

Executing arithdar toncosts(regions); now:
```

```
unsigned, double darray constructor     ← first call base constructor for toncosts
unsigned, double arithdar constructor   ← then call derived constructor
Enter the regional tinsel costs per ton in dollars:
Region 1: 24
Region 2: 28
Region 3: 26
Average cost per ton = 26               ← calculated by arithdar method

Executing arithdar costs = tons * toncosts; now:
unsigned, double darray constructor     ← create product in darray::operator*()
darray & darray constructor             ← return object to main()
darray destructor                       ← free temporary product object
darray & darray constructor             ← first call base constructor for costs
darray & arithdar constructor           ← then call derived constructor
Regional expenditures:
5280 8960 6760                          ← calculated by darray method
(all in dollars)
Total expenditure = $21000              ← calculated by arithdar method
arithdar destructor                     ← first call derived destructor for costs
darray destructor                       ← then call base destructor
darray destructor                       ← call base destructor for return object
arithdar destructor                     ← first call derived destructor for toncosts
darray destructor                       ← then call base destructor
arithdar destructor                     ← first call derived destructor for tons
darray destructor                       ← then call base destructor
```

Program Notes

Let's examine parts of the program more closely. First, the statement

```
arithdar tons(regions);
```

creates a type *arithdar* object with `regions` elements. The sample run sets `regions` to `3`. As discussed, and as the output shows, the program first calls the *darray* base class constructor, then the *arithdar* derived class constructor. The signature of the declaration matches `arithdar(unsigned int n, double val)`, so that is the constructor used. (Recall that the class prototype provides a default value of `0` for the second argument.) The same sequence of events marks the creation of the `toncosts` object.

The program uses the *arithdar* class methods `sum()` and `average()` to compute the sum of the `tons` elements and the average of the `toncosts` elements, thereby much simplifying the user's labor.

The most elaborate sequence of function calls accompanies the following innocuous-appearing statement:

```
arithdar costs = tons * toncosts;
```



The compiler looks at the right-hand side and sees that it multiplies two *arithdar* objects. Therefore, it looks for an operator function for multiplication of two *arithdar* quantities. There isn't any, so next the compiler looks for the equivalent definition in the *darray* class. It finds one, and that's what's used to perform the multiplication:

```
// multiply two arrays element by element
darray darray::operator*(const darray & a)
{
    if(size != a.size)
    {
            cout << "Array multiplication error: arrays must "
                    "be of same size. Bye.\n";
            exit(1);
    }
    darray product(size);
    for (int i = 0; i < size; i++)
            product[i] = arr[i] * a.arr[i];
    return product;
}
```

Note that this function creates a *darray* object called *product.* Therefore the function calls a *darray* constructor when creating *product.* When the function terminates, so does the *product* object, so the program calls the *darray* destructor. But before that happens, the operator function returns the object to `main()`. The return mechanism creates a new, temporary object which is initialized to *product.* Both objects are type *darray,* so the *darray()* constructor that initializes one *darray* object to another is called. This sequence of events accounts for the following program output:

```
unsigned, double darray constructor     ← create product in darray::operator*()
darray & darray constructor             ← return object to main()
darray destructor                       ← free product object
```

Next, the program creates the `costs` array and initializes it to the temporary array. Because `costs` is type *arithdar,* the program first calls the base class constructor, then the derived class constructor. That explains the following output lines:

```
darray & darray constructor             ← first call base constructor for costs
darray & arithdar constructor           ← then call derived constructor
```

After displaying the results of multiplying the two objects, the program sums the expenditures, then deletes the various objects when the `main()` function terminates. The program deletes the objects in the reverse order of creation, as is typical of automatic variables. Thus the deletion order is `costs`, then the temporary object used to hold the return value, then `toncosts`, then `tons`. Note that the program calls the derived class destructor first, then the base class destructor for each object. The temporary object was type *darray,* so only the `~darray()` constructor is called for it.

As noted earlier, a lot goes on behind the scenes. Perhaps the main point to this exercise is to foster the realizations that C++ programs with classes use constructors frequently and that class design often requires a variety of constructors to handle different situations, including initialization, argument passing, function returns, and type conversions.

MANAGING YOUR INCLUDE FILES

Let's take a break from OOP concepts and look at program management. The examples have been using the usual C++ approach to organizing the source code:

- Place the class definition in a header file having an `h` extension, as in `arithdar.h`.
- Place the class member functions definition in a methods file program having the same base name as the corresponding class definition file, but having the usual source code extension, as in `arithdar.cpp`.
- Place the program using these resources in a third file, as in `derived.cpp`.
- Use the `#include` facility to include the header file in the other two files.

Adding a derived class adds to these requirements, for a derived class also needs access to the base class definition file and the base class method files. In the last example, that would be `darray.h` and `darray.cpp` (or, if you want the construct-and-tell version, `darray_v.cpp`). To provide access to the methods files, just compile them along with the main program file:

```
CC derived.cpp arithdar.cpp darray.cpp
```

(In Turbo C++, simply place these files in the project list.)

Note, however, that the main program (`derived.cpp`) includes just one of the class header files:

```
#include "arithdar.h"
```

That's because the `arithdar.h` file itself included the `darray.h` file, so including `arithdar.h` includes both files. This approach simplifies remembering what files to include, but it has a potential problem: you may wind up including `darray.h` twice. For instance, the program might be using *darray* objects already and thus already include the `darray.h` file. Then, when you decide to add *arithdar* objects, you include the `arithdar.h` file, which in turn includes `darray.h`. Including a header file like `darray.h` twice is an error because that results in duplicate class definitions. That is, each inclusion inserts a copy of the class definition into the source code, so the source code winds up repeating itself about what the *darray* class is.



The C++ preprocessor `#ifndef` directive (it stands for *if n*ot *def*ined) provides a way to avoid that problem. If the symbolic name following this directive has not been defined previously with a `#define` statement, the preprocessor processes the lines between the `#ifndef` directive and a subsequent `#endif` directive. Otherwise, the preprocessor ignores those lines, and they never reach the compiler. For instance, consider the following code:

```
#define FOO 3
#ifndef FOO
#define FOO 20
#endif
```

Here *FOO* has been defined, so the preprocessor will skip the `#define FOO 20` line, and `FOO` retains the value `3`. But if `FOO` had not been previously defined, the preprocessor would set `FOO` to `20`. In general, you don't have to set a particular value to define a symbolic name. For instance, the following defines `IBM` even though it doesn't assign it a value:

```
#define IBM
```

To solve the multiple include problem, modify the `darray.h` file so that it looks like this:

```
// darray.h
#ifndef _ _DARRAY_ _
#define _ _DARRAY_ _
class darray
{
   ...
}
#endif
```

Let's see how this works. First, suppose the preprocessor encounters an `#include "darray.h"` directive and the file has not been included previously. In this case, the symbol `_ _DARRAY_ _` should be undefined at that point. (We preceded and followed the class name with two underscore characters to make accidental duplication of the name unlikely.) Then the lines between `#ifndef` and `#endif` are included in the program. These lines not only provide the class definition, they also define the `_ _DARRAY_ _` symbol. (Remember, it's not necessary to assign a value to a symbolic name to define it.) Next, suppose the program encounters a second `#include "darray.h"` directive. This time, `_ _DARRAY_ _` has already been defined when the first `#include` directive was processed, so the preprocessor skips to the `#endif` and doesn't pass along the class definition. In brief, the first occurrence of `#include "darray.h"` causes the preprocessor to define `_ _DARRAY_ _` and include the definition. Subsequent occurrences of `#include "darray.h"` cause the preprocessor to skip the `#ifndef` section, thus including

an empty file. This is a standard technique for setting up header files, and we'll use it from here on out. You should retrofit `darray.h` and `arithdar.h` in this manner.

A DERIVED CLASS WITH BOUNDS CHECKING

Class derivation lets you modify a base class to meet specific needs. The *arithdar* class, for instance, met a need for more arithmetic support. But, if you develop different programming needs, you can go back and derive another new class from the same base. Let's look at another example now and enhance the *darray* class by adding bounds checking for arrays.

First, let's consider what bounds checking is and why you might want it. C++ arrays, like C arrays, have a little weakness: nothing but your vigilance and the programmer's code of honor prevents you from doing the following:

```
double peanuts[10];
peanuts[80] = 258.44;    // bad mistake, but permitted
```

The `peanuts` array has only ten elements, but this code assigns a value to the eighty-first element. The computer will locate the beginning of the `peanuts` array, then go to a location 80 units `(80 * sizeof (double) bytes)` later in memory, then place the value `258.44` there. In other words, it may very likely overwrite other data, producing an unpleasant surprise later. The compiler won't catch this mistake because it does not check to see whether the subscript falls in the legitimate range of 0–9. The act of checking if an array subscript is a valid one is called *bounds checking*. Some languages do it, but C++ doesn't. This lack of bounds checking lets the compiler work faster and produce more compact code, but it also increases the chances of making hard-to-catch errors.

To make the world safe from unsuitable array indices, let's create a new array class (the *bound_dar* class) to make that sort of blunder impossible. Not only will this give you more practice deriving classes, it eventually will lead to a new topic—deriving a class from a derived class.

Installing bounds checking requires only minor modifications to the *darray* class. Let's start by reviewing how a *darray* object handles subscripts. If *jinx* is a *darray* object, you access individual elements by using subscript notation, as in `jinx[i]`. This notation, in turn, invokes the `darray::operator[]()` function. So all you have to do is create a `bound_dar::operator[]()` function that rejects bad subscripts. Then, if you create a *bound_dar* object, it will use the *bound_dar* subscripting function instead of the *darray* subscripting function. Here's one approach:

```
// redefined operator
double & bound_dar::operator[](int i)
{
```

```
    ok(i);          // check index before continuing
    return arr[i];
}
```

This modifies the original `operator[]()` function by adding the `ok()` function call. The role of this function will be to determine if the subscript `i` is out of bounds; if it is, the function will terminate the program. You could encode the checking directly into the operator definition, but bounds checking conceptually is a separate process from accessing an element. Also, if some future member function also needs to do bounds checking, it can use the `ok()` function.

Here's one possibility for defining `ok()`:

```
void bound_dar::ok(int i) // quit if index is out of bounds
{
    if (i < 0)
    {
      cout<< "Error in array limits:\n"
          << "index " << i << " less than 0\n";
      exit(2);
    }
    else if (i >= size)
    {
      cout << "Error in array limits:\n"
           << "index " << i << " greater than "
           << size - 1 << "\n";
        exit(3);
        }
}
```

Here `size`, recall, is the class member that's set to the number of elements in the array. If the subscript is too large (equal to or larger than `size`) or too small (less than `0`), `ok()` calls the standard C library function `exit()` to terminate the program. The `exit()` return value goes to the operating system, and the convention is that a nonzero exit value indicates a problem. If, for example, a UNIX shell script runs a program using this class and detects an exit value of *3*, it can conclude that the program quit because of a too-large subscript. Note that the new class will catch subscript errors during run-time, not compile time.

The next step is to add these definitions to a derived class, which we'll call *bound_dar*, for *bound*ed *d*ouble *ar*ray. Listing 11-7 shows the class definition. It adds the new `ok()` function. There's no reason to make `ok()` `public`, so we've made it `protected`. Then, as all good derived classes require, the definition declares constructors for the class. Finally, it provides a prototype for redefining the `[]` operator.

LISTING 11-7 bnddar.h

```
// bnddar.h -- define array class that checks index values
#ifndef __BNDDAR__
#define __BNDDAR__

#include "darray.h"

class bound_dar : public darray
{
protected:
    void ok(int i);                          // index checker
public:
// constructors
    bound_dar();
    bound_dar(unsigned int n, double val = 0.0);
    bound_dar(const double *pn, unsigned int n);
    bound_dar(const bound_dar & a);
    bound_dar(const darray & a);
// operators
    double & operator[](int i);          // redefined

};

#endif
```

Listing 11-8 presents the class method definitions. Because the *bound_dar* class doesn't add any data members to the *darray* class, all that the *bound_dar* constructors have to do is pass their arguments, if any, to the corresponding *darray* constructors.

LISTING 11-8 bnddar.cpp

```
// bnddar.cpp -- methods for bound_dar class
#include <iostream.h>
#include <stdlib.h>
#include "bnddar.h"

// protected method
void bound_dar::ok(int i) // quit if index is out of bounds
{
    if (i < 0)
    {
        cout << "Error in array limits:\n"
             << "index " << i << " less than 0\n";
        exit(2);
```



```
    }
    else if (i >= size)
    {
       cout << "Error in array limits:\n"
            << "index " << i << " greater than "
            << size - 1 << "\n";
       exit(3);
    }
}

// constructors
bound_dar::bound_dar()
{
}

bound_dar::bound_dar(unsigned int n, double val) : darray(n, val)
{
}

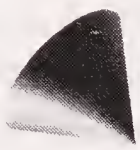
bound_dar::bound_dar(const double *pn, unsigned int n)
    : darray(pn, n)
{
}

bound_dar::bound_dar(const bound_dar & a) : darray(a)
{
}

bound_dar::bound_dar(const darray & a) : darray(a)
{
}

// redefined operator
double & bound_dar::operator[](int i)
{
    ok(i);          // check index before continuing
    return arr[i];
}
```

The final step is exercising the new class. Listing 11-9 does that with a `for` loop that uses the wrong test for terminating the loop, causing the program to try to access an element beyond the final one. Let's see how the *bound_dar* class handles that affront. Remember to compile this program with the base class methods provided by `darray.cpp` and `bnddar.cpp`.



LISTING 11-9 use_bdar.cpp

```
// use_bdar.cpp -- use the bounded array
// compile with darray.cpp and bnddar.cpp
#include <iostream.h>
#include "bnddar.h"

int main(void)
{
    bound_dar prices(5, 19.99);   // create, initialize object

    for (int i = 0; i <= 5; i++)    // one step too far
            cout << "$" << prices[i] << "\n";
    cout << "Bye!\n";
    return 0;
}
```

Here is the output:

```
$19.99
$19.99
$19.99
$19.99
Error in array limits:
index 5 greater than 4
```

The program correctly processes the first five elements, then terminates the program when the index exceeds the maximum value of `4`. Note that the program never reaches the final output statement, showing that `ok()` did, indeed, terminate the program when the subscript went out of bounds. Thus this modest extension of the class definition provides a major increase in program security.

EXTENDING INHERITANCE

In C++ you can derive a class from a derived class. In that case, the final class inherits the properties of all its ancestors. This is useful. For instance, you may have started out with a base class designed for a particular programming project. Then you may have moved to a second project and found that you could adapt this class, through class derivation, to meet somewhat different needs. Later, you might find the derived class to be a suitable base class for yet some other project. It's all part of the C++ plan to let you reuse code, adapting it to your current needs. We'll illustrate this process by extending the

bound_dar class, which was derived from the *darray* class, to allow array indexing to begin at values other than zero. For example, if an array represents your tax returns for the years 1985 through 1990, you'll be able to index the array from `1985` to `1990` instead of from `0` to `5`. Little touches like that can make programming clearer and more enjoyable.

The new class—let's call it *lim_dar*—derives from the *bound_dar* class and thus inherits the `protected` and `public` parts of that class. In particular, it inherits the

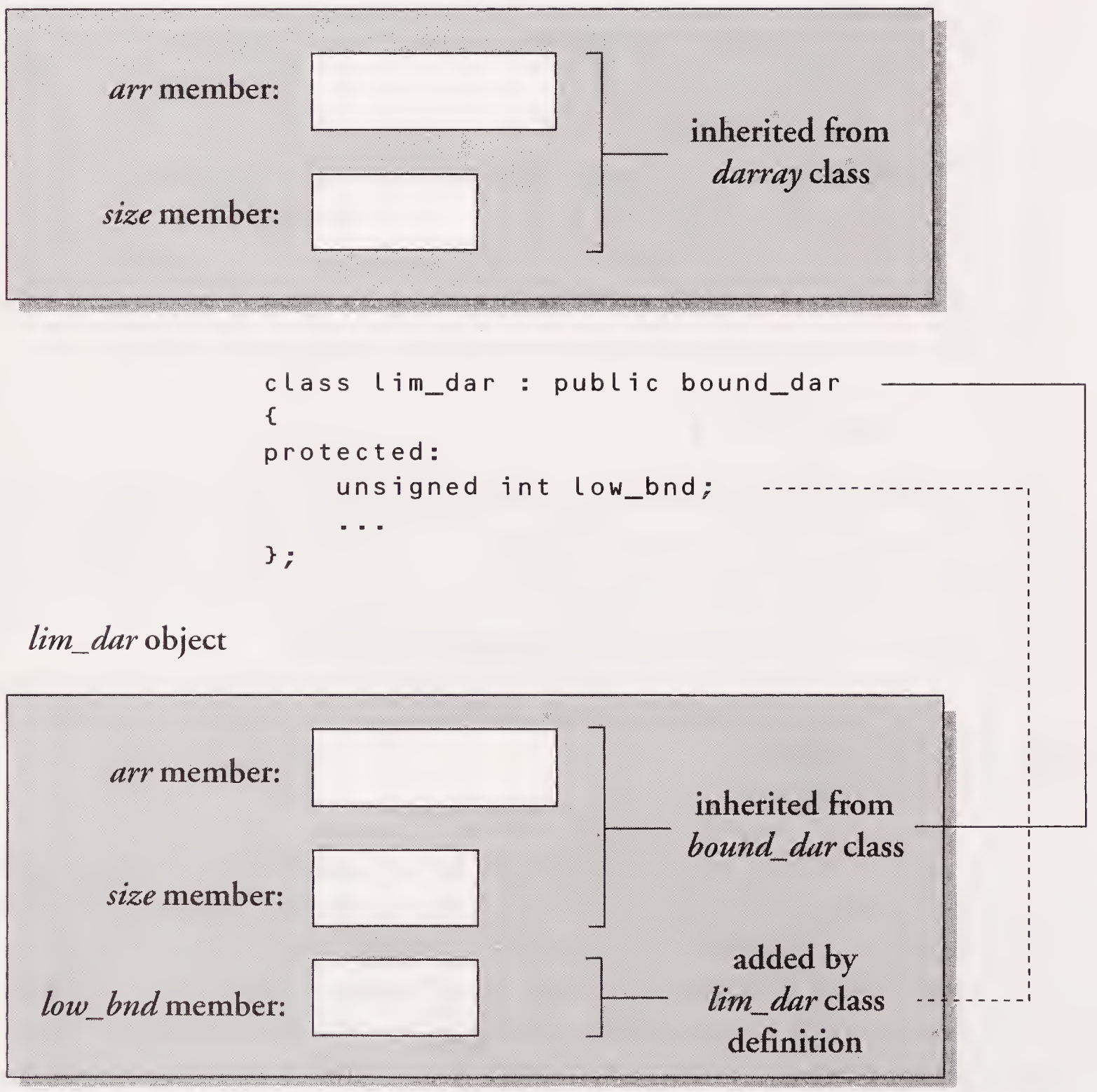


Figure 11-3 *Adding a new member*

bounds-checking feature. Because *bound_dar* derives from the *darray* class, *lim_dar* inherits, via *bound_dar*, the `protected` and `public` parts of *darray*, too. Thus the *lim_dar* class contains an *arr* member to point to the beginning of the array and a `size` member indicating the number of elements. The class automatically inherits the *darray* and *bound_dar* class member functions as well.

What new class members will *lim_dar* need to provide its new indexing service? The `arr` member already states where the array is stored, and the `size` member already states the number of elements. What's missing is an element stating the value of the first index. That can be remedied by adding a new data element, `low_bnd`, to indicate the subscript value corresponding to the first element. So if subscripting is to start with 1985, then `low_bnd` will be set to that value. The new subscript limit, in turn, requires new versions of the `operator[]()` function and the `ok()` function, for the prior versions assumed subscripting began with zero. Also, let's add three methods to support the new subscript feature. One method reports the lower bound, one reports the upper bound, and one lets you reset the lower bound. Listing 11-10 shows the class definition. Note that all it takes to add a new data member to a derived class is to declare it in the definition for the derived class. See Figure 11-3.

LISTING 11-10 lim_dar.h

```
// lim_dar.h -- lim_dar class -- definable lower bound
#ifndef __LIM_DAR__
#define __LIM_DAR__

#include "bnddar.h"

class lim_dar : public bound_dar
{
protected:
    unsigned int low_bnd;       // new data member
    void ok(int i);             // redefine ok()
public:
// constructors
    lim_dar();
    lim_dar(unsigned int n, double val = 0.0);
    lim_dar(unsigned int n, int lb, double val = 0.0);
    lim_dar(const double *pn, unsigned int n);
    lim_dar(const lim_dar & a);
    lim_dar(const bound_dar & a);
    lim_dar(const darray & a);
// new methods
    void new_lb(int lb);        // reset lower bound
    int lbound();               // return lower bound
```

```
    int ubound();                    // return upper bound
// redefined operators
    double & operator[](int i);
    lim_dar & operator=(const lim_dar & a);
};

#endif
```

Listing 11-11 presents the class method definitions. Once again, class constructors aren't inherited, so the class definition requires new constructors. This time they have something to do—initialize `low_bnd`, the new class data member, to the chosen initial subscript value. As mentioned, the new class redefines `ok()` and `operator[]()` to handle the new subscript bounds. Also, the class redefines the assignment operator. That's because the class now has three data members (`arr`, `size`, and `low_bnd`) instead of two, but the assignment operator inherited from the *darray* class handles only the first two members. The new version handles all three. By the way, the *bound_dar* class, because it had the same two data members as the *darray* class, could use the *darray* definition of assignment, which is why the *bound_dar* class did not have to redefine the assignment operator.

Note that providing a new assignment operator is not analogous to providing a new constructor. A new constructor adds to what the ancestral constructors do, but a new assignment operator entirely supersedes the old one. This is not inconsistent, for the assignment operators for the base and derived class share the same name, and only one version can be invoked at a time. That is, the various assignment operators are overloaded functions, and the program uses the signature, or argument list, to determine which version to use. But constructors for the base and derived class have different names, hence invoking one doesn't rule out calling the other.



LISTING 11-11 lim_dar.cpp

```
// lim_dar.cpp -- methods for lim_dar class
#include <iostream.h>
#include <stdlib.h>
#include "lim_dar.h"

// protected method
    // lower bound for array index is now low_bnd, and
    // upper bound is now low_bnd + size - 1
void lim_dar::ok(int i) // variable lower bound
{
    if (i < low_bnd)
    {
```

```
            cout << "Error in array limits:\n"
                 << "index " << i << " less than "
                 << low_bnd <<  "\n";
            exit(2);
        }
        else if (i >= size + low_bnd)
        {
            cout << "Error in array limits:\n"
                 << "index " << i << " greater than "
                 << size + low_bnd - 1 << "\n";
            exit(3);
        }
}

// constructors -- initialize the new data member
lim_dar::lim_dar() : bound_dar()
{
    low_bnd = 0;  // default sets starting subscript to 0
}

lim_dar::lim_dar(unsigned int n, double val) : bound_dar(n, val)
{
    low_bnd = 0;  // default value
}

lim_dar::lim_dar(unsigned int n, int lb, double val)
    : bound_dar(n, val)
{
    low_bnd = lb; // set starting subscript explicitly
}

lim_dar::lim_dar(const double *pn, unsigned int n)
    : bound_dar(pn, n)
{
    low_bnd = 0;
}

lim_dar::lim_dar(const lim_dar & a) : bound_dar(a)
{
   low_bnd = a.low_bnd;
}

lim_dar::lim_dar(const bound_dar & a) : bound_dar(a)
{
    low_bnd = 0;
}

lim_dar::lim_dar(const darray & a) : bound_dar(a)
{
```



```
    low_bnd = 0;
}

// more methods
void lim_dar::new_lb(int lb)   // reset starting subscript to lb
{
    low_bnd = lb;
}

int lim_dar::lbound()          // return starting subscript
{
    return low_bnd;
}

int lim_dar::ubound()          // return subscript of last element
{
    return size + low_bnd - 1;
}

// redefined operators
double & lim_dar::operator[](int i)
{
    ok(i);
    return arr[i - low_bnd];
}

lim_dar & lim_dar::operator=(const lim_dar & a)
{
   if (this == &a)                 // if object assigned to self
      return * this;               // don't change anything
   delete arr;
   size = a.size;
   low_bnd = a.low_bnd;            // copy new member
   arr = new double[size];
   for (int i = 0; i < size; i++)
      arr[i] = a.arr[i];
   return *this;
}
```

Let's look at the code for a few selected class methods. First, consider these two prototypes:

```
lim_dar(unsigned int n, double val = 0.0);
lim_dar(unsigned int n, int lb, double val = 0.0);
```

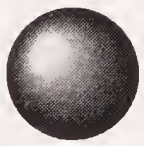
The first takes two arguments, with `n` being the number of elements and `val` being the value to which each element is set. The second takes one more argument, with `lb` rep-

resenting the value of the first subscript. As the function definitions show, the first constructor sets the first subscript to zero, while the second sets it explicitly:

```
lim_dar::lim_dar(unsigned int n, double val) : bound_dar(n, val)
{
    low_bnd = 0;  // default value
}

lim_dar::lim_dar(unsigned int n, int lb, double val)
    : bound_dar(n, val)
{
    low_bnd = lb; // set starting subscript explicitly
}
```

The definitions show that each function passes the `n` and `val` values to the `bound_dar()` constructor, which, in turn, passes them on to the *darray* constructor to initialize the *arr* and `size` members. But, because it's the *lim_dar* class that provides the *`low_bnd`* member, the *lim_dar* constructors are responsible for initializing that member.



The derived class constructors are responsible for initializing any data members added to those inherited from the base class. The base class constructors are responsibile for initializing the inherited data members.

As the prototypes show, each function supplies a default value of `0` for the array values. Thus the declaration

```
lim_dar bolts(10);
```

creates an array of 10 elements each set to `0` and with the array subscripting starting at `0`. And the declaration

```
lim_dar months(12,1);
```

creates an array of 12 elements each set to `0` and with the array subscripting starting at `1`.

The class provides two constructors for initializing a *lim_dar* object to a ancestral objects:

```
lim_dar::lim_dar(const bound_dar & a) : bound_dar(a)
{
    low_bnd = 0;
}

lim_dar::lim_dar(const darray & a) : bound_dar(a)
```



```
{
    low_bnd = 0;
}
```

Recall that a constructor with a single argument serves to define conversion from the argument type to the class type. So these two constructors, in addition to letting you initialize a *lim_dar* object to a *bound_dar* or *darray* object, also let you assign objects of either ancestrial type to a *lim_dar* object and to pass objects of either type to a function having an argument that's a reference to a *lim_dar* object

You can easily add to this list of constructors. For instance, consider this:

```
lim_dar(const bound_dar & a, int lb); // possible prototype
```

It would let you initialize a *lim_dar* object to a *bound_dar* object, resetting the lower bound. However, you get the same result by using the current constructor, then using the `new_lb()` method to reset the lower bound.

Finally, look at the redefined subscript operator:

```
// redefined operators
double & lim_dar::operator[](int i)
{
    ok(i);
    return arr[i - low_bnd];
}
```

In its heart of hearts, C++ does insist on using *0* as the first subscript of an array, and this function translates the exterior view of array subscripts to the interior view. For example, consider the following declaration:

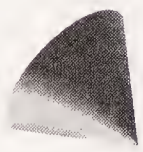
```
lim_dar vintages(5, 1977);
```

It creates an array with five elements, with the subscripts beginning with `1977`. Suppose you make the following statement:

```
vintages[1979] = 12.20;
```

The `operator[]()` function then translates 1979 to 1979 minus 1977, or 2, so the function returns a reference to the `vintages.arr[2]` element.

Listing 11-12 presents a typical use for the *lim_dar* class—using years as subscripts. Remember to compile it with *bnddar.cpp* and *darray.cpp* in order to bring in the inherited methods, as well as with *lim_dar.cpp*.



LISTING 11-12 use_lim.cpp

```
// use_lim.cpp -- use the lim_dar class
// Compile with lim_dar.cpp, bnddar.cpp and darray.cpp
#include <iostream.h>
```

```
#include "lim_dar.h"

const int Year = 1977;
const int Years = 5;
int main(void)
{
    lim_dar vintages(Years, Year);
    cout << "Enter bids for the following vintages of "
            "Chateau Spiff:\n";
    for (int year = Year; year < Year + Years; year++)
    {
            cout << "Year " << year << ": $";
            cin >> vintages[year];
    }
    cout << "Recapitulating, here are the bids in dollars:\n";
    cout << vintages;

    return 0;
}
```

The declaration

```
lim_dar vintages(Years, Year);
```

creates a *lim_dar* object with 5 elements (because `Years` is *5*), and it causes the first subscript to be `1977` (the value of `Year`) instead of the usual `0`. Here's a sample run (1979 was the year Uncle Jacques got his big chance to run the operation):

```
Enter bids for the following vintages of Chateau Spiff:
Year 1977: $38.58
Year 1978: $34.85
Year 1979: $0.62
Year 1980: $22.24
Year 1981: $18.65
Recapitulating, here are the bids in dollars:
38.58 34.85 0.62 22.24 18.65
```

As the final line of output demonstrates, the *lim_dar* class has inherited, via the *bound_dar* class, the overloaded `<<` operator as defined in the *darray* class.

WHICH METHODS?

The last few examples covered several new concepts, so it's a good time to review one of the most important matters in inheritance—which methods get called and when. Suppose, for example, you have the following three declarations:



```
darray desiree(6);
bound_dar oliver(10);          // based on darray
lim_dar babar(12, 1958);       // based on bound_dar
```

As the examples have shown, creating an object invokes the constructor for a class plus all the constructors for ancestor classes, with the most ancestral class constructor called first. So creating *desiree* invokes the *darray* constructor, creating *oliver* invokes the *darray* constructor, then the *bound_dar* constructor, and creating *babar* invokes the *darray* constructor, then the *bound_dar* constructor, then the *lim_dar* constructor. This order makes sense, for you have to start with the base before adding to it.

Similarly, when an object expires, the program calls the class destructors in reverse order, with the most ancestral destructor called last. This, too, makes sense, for it processes the added-on features and members before getting to the base object.

Only the constructors and destructors automatically produce these cascading sequences of calls. If you have multiply-defined versions of a regular class method, the program invokes just the most recently defined (in terms of ancestry) method:

```
desiree[2] = 2.56;        // uses darray::operator[]()
oliver[4] = 65.5;         // uses bound_dar::operator[]()
babar[1960] = 12.12;      // uses lim_dar::operator[]()
bound_dar jordan;
jordan = oliver;          // used darray::operator=()
lim_dar toby;
toby = babar;             // uses lim_dar::operator=()
```

The `jordan` assignment uses the *darray* operator for assigning one *bound_dar* object to another because the *bound_dar* class doesn't override the *darray* definition. But assigning one *lim_dar* object to another uses the *lim_dar* version of assignment, because that's the most recent version of assignment. Multiplying two *lim_dar* objects together uses the *darray* definition of multiplication because neither the *bound_dar* nor the *lim_dar* redefines that operator.

This is not the complete story on which method is invoked when you use an object. When you use an object reference or a pointer to an object, the situation becomes more complex. We'll return to that story after discussing a related topic, class conversions.

CLASS CONVERSIONS

C++ provides automatic class conversion when you assign an object to an ancestral class. For instance, you can do the following:

```
bound_dar mickey(20, 88.8);    // object of derived class
```

```
darray donald;                  // object of base class
donald = mickey;                // convert to an ancestor type

lim_dar goofy;
mickey = goofy;                 // convert to parent type
donald = goofy;                 // convert to grandparent type
```

Converting *goofy* to, say, a *darray* type by assigning it to *donald* doesn't mean the `goofy` object itself is altered. Instead, it means `donald` is created in the *darray* format using values taken from the *goofy* object. There are some points to note about this process:

1. C++ uses memberwise assignment (each member of one object is an exact copy of each member of the other object) unless you've redefined the assignment operator.

2. If you have redefined the assignment operator and if the object being assigned to is of a different class than the object being assigned, C++ uses the assignment definition associated with the object being assigned to rather than the function defined for the assigned object. For instance, assigning the *lim_dar* object *goofy* to the *darray* object *donald* uses the *darray* definition of assignment, not the *lim_dar* definition.

3. Any data members defined for the assigned object but not for the assignee are dropped in the conversion. For instance, assigning the *lim_dar* object *goofy* to the *darray* object *donald* copies *goofy*'s array contents and array size to `donald`, but the `low_bnd` information is lost.

4. An object can be converted to any of its ancestral types.

Let's take a closer look at the second point. Consider the example above that assigns the *lim_dar* object *goofy* to the *bound_dar* object *mickey*. The *lim_dar* class defines assignment, but the statement assigning *goofy* to *mickey* doesn't use `lim_dar::operator=()` because that function requires the left-hand side to be a *lim_dar* object, and *mickey* isn't one. Instead, the program looks for a *bound_dar* definition for assignment. Since there isn't one, the program moves up the ancestral tree and uses the *darray* definition of assignment.

So you can convert a derived type to an ancestral type automatically by assigning a derived type to an ancestral type. Similarly, you can initialize a base type to a derived type. What about the reverse direction? Can you convert a base type to a derived type? Yes, you can, but only if you provide instructions. Remember that any class constructor with a single argument acts as a type conversion specification for converting the argument type to the class being constructed. Thus to convert type *darray* to type *bound_dar*, you need a constructor of the following prototype:

```
bound_dar(const darray &);
```

That's one reason we included just such a constructor in the *bound_dar* class definition. The other, related, reason was to make it possible to initialize a *bound_dar* object to a *darray* object.

When a class derives from a chain of classes, you should provide constructors to convert to each ancestral type. For instance, the *lim_dar* class defines conversion for both its ancestor:

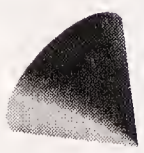
```
lim_dar(const bound_dar &);
lim_dar(const darray &);
```

Suppose, then, you try the following:

```
darray desiree(6, 30);
lim_dar eclair;
eclair = desiree;
```

C++ will use the *lim_dar(const darray &)* constructor to manage conversion when assigning *desiree* to a *lim_dar* object.

The program in Listing 11-13 illustrates conversions from derived types to base types and from base types to derived types. Once again, we've substituted the verbose versions of the various constructors so that you can see which function was called when.



LISTING 11-13 test_lim.cpp

```
// test_lim.cpp -- examine type conversions
// uses verbose versions of darray.cpp, bnddar.cpp
// and lim_dar.cpp
#include <iostream.h>
#include "lim_dar.h"

int main(void)
{
    lim_dar desiree(5, 36.0);
    cout << "lim_dar object: " << desiree;
    darray didi;
    cout << "Assigning a descendant to an ancestor:\n";
    didi = desiree;
    cout << "darray object: " << didi;

    cout << "Assigning an ancestor to a descendant\n";
    desiree = didi;
    cout << "lim_dar object: " << desiree;
    return 0;
}
```

Here is the annotated output:

```
darray (unsigned, double) constructor          ← create desiree
bound_dar (unsigned, double) constructor       ← create desiree
lim_dar (unsigned, double) constructor         ← create desiree
lim_dar object: 36 36 36 36 36
default darray constructor                     ← create didi
Assigning a descendant to an ancestor:         ← automatic conversion
darray object: 36 36 36 36 36
Assigning an ancestor to a descendant
darray (darray &) constructor                  ← convert darray to lim_dar
bound_dar (darray &) constructor               ← convert darray to lim_dar
lim_dar (darray &) constructor                 ← convert darray to lim_dar
lim_dar object: 36 36 36 36 36
darray destructor                              ← delete didi
darray destructor                              ← delete desiree
```

Creating the *lim_dar* object `desiree` takes three constructors, because *lim_dar* is derived from a chain of two classes. See Figure 11-4. Creating the *darray* object `didi` takes one constructor, because *darray* is a base class. Assigning `desiree` to `didi` invokes no class constructors, because C++ handles conversion to an ancestor type automatically. Next, assigning `didi` to `desiree` requires several constructors. The program uses the *lim_dar(const darray & a)* constructor, but evoking a *lim_dar* constructor automatically evokes the correspondint constructors for its ancestor classes. See Figure 11-5. Finally, the program deletes *didi* and *desiree* objects. (note: the initial Turbo C++ release implemented assignment somewhat differently and reported an initial call to the destructor, presumbably because it used an additional temporary object.) No lines announce *bound_dar* and *lim_dar* destructors because these classes didn't define a destructors. The darray class, recall, needed a destructor because its constructors used *new*. The constructors for the derived classes didn't use *new* and so didn't need specially defined destructors.

POINTERS AND REFERENCES TO OBJECTS

Suppose you write a function that uses a pointer to an object of a base class, such as *darray*, as an argument. For example, suppose you wrote this:

```
void dumdar(darray * pd)
{
    cout << "The object in question represents an array with "
```



```
        << pd->arsize() << " elements\n";
}
```

Can you then use *dumdar()* with an argument that is a pointer to an object of a derived class, such as *arithdar* or *bound_dar* or *lim_dar*? Yes, you can, for C++ allows you to assign the address of a derived class to a pointer to the base class. For example, consider these statements:

```
darray chicago;
bound_dar atlanta;
lim_dar topeka;
darray * p_dar;   // pointer to darray class
p_dar = &chicago; // valid
p_dar = &atlanta; // valid
p_dar = &topeka;  // valid
```

The pointer type `darray *` is compatible with the address of any object of a class derived from *darray* or its descendents.

Similarly, a reference to a base type is compatible with any object of a derived class. For example, if you define a function with a `darray &` type argument, you can pass a *bound_dar* or *lim_dar* object to that function.

the *lim_dar* class derives from the *bound_dar* class, and the *bound_dar* class derives from the *darray* class

```
lim_dar desiree(5,36.0);
```

Declaration invokes three constructors:

1. Signature matches the *lim_dar(int, double)* constructor
2. Program determines that *darray* begat *bound_dar* and that *bound_dar* begat *lim_dar*
3. *darray(int, double)* invoked
4. *bound_dar(int, double)* invoked
5. *lim_dar(int, double)* invoked

constructors invoked in order of inheritance, with the base constructor being called first

Figure 11-4 *Constructor chains*

This accommodating nature of object pointers and references is a convenience. It also can create a problem, but a resolvable one, as you will see in the next section.

VIRTUAL FUNCTIONS

As you've seen, when you create a series of derived classes with each class having its own version of a member function, an object uses its own class version of that member function. For example, a *darray* object uses the `operator[]()` function defined for the base *darray* class, while a *lim_dar* object uses the `operator[]()` function defined for the derived *lim_dar* class. This simple relationship doesn't always guarantee that the correct method is used, however. The problems arise when you invoke a class member function by using a pointer or a reference instead of by using a regular object variable. For example, you can get into trouble by defining a function that takes a reference to a class *darray* object as an argument:

```
void show_it(darray & ah, int lim)
{
    for (i = 0; i < lim; i++)
            cout << ah.arr[i] << "\n";
}
```

lim_dar object *darray* object

```
desiree = didi;
```

Find the *lim_dar(const darray &)* constructor and call it using *didi* as an argument:

Invokes the following sequence of calls:

```
darray(didi);
bound_dar(didi);
```

Figure 11-5 *Assigning a base object to a derived object*



This function prints `lim` array elements, one array element per line. It uses only the public *darray* method and `operator[]()`, so `show_it()` doesn't have to be a member function or a friend function. Now consider the following program fragment:

```
darray james(4, 3.0);          // a darray object
bound_dar jarvis(5, 2.2);      // a bound_dar object
show_it(james,3);              // works
show_it(jarvis, 10);           // valid, but doesn't check bounds
```

The *bound_dar* class checks array subscripts to see if they are in bounds, but the last function call fails to invoke bounds-checking, even though it has a *bound_dar* object as an argument. To see why, consider the process step-by-step:

1. The `show_it()` function's formal argument is a reference to a type *darray* object.
2. Passing the argument `jarvis` to `show_it()` causes the formal argument `ah`, a reference to *darray*, to refer to *jarvis*, a *bound_dar* object.
3. This association of `ah` with `jarvis` means the program has a type *darray* reference acting as an alias for a type *bound_dar* object.
4. By default, C++ uses the reference type, not the type of the referred-to object, to determine which methods are used.
5. Thus the `show_it()` function treats `ah` as a *darray* object and uses the `darray::operator[]()` function instead of the `bound_dar::operator[]()` function for subscripting.

Hence `show_it()` fails to check the array limits, possibly much to the programmer's embarrassment.



When a C++ reference to a base type refers to an object of a derived class, C++, by default, uses the reference type as a guide to which member functions to use.

Pointers behave similarly. Consider the following statements:

```
lim_dar france;                 // lim_dar object
lim_dar italy(12, 2000, 2.0);   // lim_dar object
darray * p_dar;
p_dar = &france;                // darray pointer to
                                // lim_dar object
*p_dar = italy;                 // uses darray::operator=()
```

Because `p_dar` is type `darray*`, the program will use the *darray* definition of assignment, not the *lim_dar* definition, to assign `italy` to `*p_dar`. So the `low_bnd` member isn't copied. Again, the program fails to use the correct operator definition.

Let's look at a simple set of classes to illuminate this problem and its eventual solution. Listing 11-14 defines a base class called *cd* that provides a simple description of a compact disk. The listing also defines a derived class called *classic* that stores additional information for classical compact disks. The base class allocates memory with `new`, and so it requires the usual supporting cast of functions needed to handle that approach:

- A destructor to free allocated memory
- A special constructor for initializing one object to another by copying data instead of addresses
- An assignment operator for assigning one object to another by copying data instead of addresses

The derived class also uses `new` to add a new member, and its definition provides the same set of facilities to the derived class. Note that `cd1.h` uses the `#ifndef` technique discussed earlier for protecting against multiple inclusions of a header file.

LISTING 11-14 cd1.h

```
// cd1.h -- classes for describing CDs

#ifndef __CD__
#define __CD__

// base class
class cd {
protected:
    char * performers;
    char * label;
    int selections;      // number of selections
    double playtime;
public:
    cd(char * s1, char * s2, int n, double x);
    cd(const cd & d);
    cd();
    ~cd();
    void report();
    cd & operator=(const cd & d);
};

// derived class
class classic : public cd {
```

```
protected:
    char * works;         // add names of works performed
public:
    classic();
    classic(char * w, char * s1, char * s2, int n, double x);
    classic(const classic & d);
    ~classic();
    void report();
    classic & operator=(const classic & d);
};

#endif
```

Listing 11-15 provides the method definitions for the two classes. It takes up several lines of code, but the main point to note is that each class has its own `report()` method. This will let you easily see which method is called. The constructors follow the forms of earlier string examples, with `new` allocating space for a string and `strcpy()` copying a passed string into the allocated space.

LISTING 11-15 cd1.cpp

```
// cd1.cpp -- methods for cd classes
#include <iostream.h>
#include <string.h>
#include "cd1.h"

// cd methods

cd::cd (char * s1, char * s2, int n, double x)
{
    performers = new char [strlen(s1) + 1];
    strcpy(performers, s1);
    label = new char [strlen(s2) + 1];
    strcpy(label, s2);
    selections = n;
    playtime = x;
}

cd::cd ()
{
    performers = new char [8];
    strcpy(performers, "Unknown");
    label = performers;
    selections = 0;
    playtime = 0.0;
```

```
}

cd::cd (const cd & d)
{
    performers = new char [strlen(d.performers) + 1];
    strcpy(performers, d.performers);
    label = new char [strlen(d.label) + 1];
    strcpy(label, d.label);
    selections = d.selections;
    playtime = d.playtime;
}

cd::~cd()
{
    delete performers;
    delete label;
}

void cd::report()
{
    cout << "Performance by " << performers;
    cout << ": Label = " << label << "\n";
    cout << selections << " selections, playing time = ";
    cout << playtime << "\n\n";
}

cd & cd::operator=(const cd & d)
{
    if(this == &d)        // if object is assigned to itself,
       return *this;      // delete nothing, and return it
    delete performers;
    performers = new char [strlen(d.performers) + 1];
    strcpy(performers, d.performers);
    delete label;
    label = new char [strlen(d.label) + 1];
    strcpy(label, d.label);
    selections = d.selections;
    playtime = d.playtime;
    return *this;
}

// classic methods
classic::classic(char * w, char * s1, char * s2, int n, double x)
            : cd(s1, s2, n, x)
{
    works = new char [strlen(w) + 1];
    strcpy(works, w);
}
```



```
classic::classic()
{
    works = new char [5];
    strcpy(works, "None");
}

classic::classic (const classic & d)
{
    works = new char [strlen(d.works) + 1];
   strcpy(works, d.works);
}

classic::~classic()
{
    delete works;
}

void classic::report()
{
    cout << "Works: " << works << "\n";
    cd::report();
}

classic & classic::operator=(const classic & d)
{
    if(this == &d)        // if object is assigned to itself,
       return *this:      // delete nothing, and return it
    delete performers;
    performers = new char [strlen(d.performers) + 1];
    strcpy(performers, d.performers);
    delete label;
    label = new char [strlen(d.label) + 1];
    strcpy(label, d.label);
    selections = d.selections;
    playtime = d.playtime;
    delete works;
    works = new char [strlen(d.works) + 1];
    strcpy(works, d.works);
    return *this;
}
```

Now let's see what happens when you use a pointer or a reference to identify an object. Listing 11-16 declares two objects, one of type *cd* and one of type *classic*. First, it uses the objects directly to invoke the `report()` method. Then it uses a pointer of type `cd *` to invoke *report()* for each object. This experiment illustrates how C++ handles class member functions invoked via pointers. Finally, it uses two functions (`hubba()`

and `bravo()`) to invoke `report()` for each object. The `hubba()` function passes an object by value, and `bravo()` passes it by reference. This experiment illustrates how C++ handles objects represented by references. Remember to compile Listing 11-16 together with Listing 11-15.

LISTING 11-16 usecd1.cpp

```
// usecd1.cpp -- calling cd methods in VARIOUS WAYS
// compile with cd1.cpp
#include <iostream.h>
#include "cd1.h"
void hubba(cd disk);
void bravo(cd & disk);
int main(void)
{
    cd c1("Beatles", "Capitol", 14, 35.5);
    classic c2 = classic("Piano Sonata in B flat, Fantasia in C",
                         "Alfred Brendel", "Philips", 2, 57.17);
    cd *pcd = &c1;

    cout << "Using object directly:\n";
    c1.report();    // use cd method
    c2.report();    // use classic method

    cout << "Using type cd * pointer to objects:\n";
    pcd->report();  // use cd method for cd object
    pcd = &c2;
    pcd->report();  // use cd method for classic object

    hubba(c1);
    hubba(c2);
    bravo(c1);
    bravo(c2);

    return 0;
}

void hubba(cd disk)
{
    cout << "Hubba hubba!  (object passed by value)\n";
    disk.report();
}

void bravo(cd & disk)
{
    cout << "Bravo!  (object passed by reference)\n";
    disk.report();
}
```

Here is the program output—note which `report()` version is called each time data for the *classic* object are displayed:

```
Using object directly:
Performance by Beatles: Label = Capitol          ← cd object, method
14 selections, playing time = 35.5

Works: Piano Sonata in B flat, Fantasia in C     ← classic object, method
Performance by Alfred Brendel: Label = Philips
2 selections, playing time = 57.17

Using type cd * pointer to objects:
Performance by Beatles: Label = Capitol          ← cd object, method
14 selections, playing time = 35.5

Performance by Alfred Brendel: Label = Philips   ← classic object, cd method
2 selections, playing time = 57.17

Hubba hubba!  (object passed by value)
Performance by Beatles: Label = Capitol          ← cd object, method
14 selections, playing time = 35.5

Hubba hubba!  (object passed by value)
Performance by Alfred Brendel: Label = Philips   ← classic object, cd method
2 selections, playing time = 57.17

Bravo!  (object passed by reference)
Performance by Beatles: Label = Capitol          ← cd object, method
14 selections, playing time = 35.5

Bravo!  (object passed by reference)
Performance by Alfred Brendel: Label = Philips   ← classic object, cd method
2 selections, playing time = 57.17
```

When `pcd` points to the *classic* object `c2`, the expression `pcd->report()` invokes the *cd* version of `report()` because `pcd` is declared as a pointer to *cd*. Similarly, in the `bravo()` function, although *disk* refers to the *classic* object `c2`, the expression `disk.report()` invokes the *cd* version of `report` because `disk` is declared as a reference to *cd*.

The situation for the `hubba()` function is slightly different. Because this function passes by value, its `disk` parameter is a new variable of type *cd*. Passing a *classic* object to the function then initializes the new *cd* object using the values of `c2`. Because *disk* is a type *cd* object, it naturally uses `cd::report`.

THE VIRTUAL SOLUTION

The problem, then, is that using a reference or a pointer to identify an object can cause the program to use the wrong member function. This happens when the pointer or reference associated with a base type identifies an object of a derived type. In that situation, the program uses the class member function defined for the base type instead of the equivalent function defined for the derived type. Ideally, the program would use the method associated with the object type instead of the one associated with the pointer or reference type. So why doesn't it? The reason is that this goal does make life more difficult for the compiler. To see why, suppose a program relies upon user input via a menu to control which type of object is associated with a reference or a pointer. This makes the type of object associated with the pointer a run-time decision. That means the compiler cannot tell in advance what type of object will be assigned to a base-class pointer. Normally, the version of an overloaded function used in a C++ program is determined at *compile* time. That's called *static linking*. But now we're asking a program to decide which function to use during *run* time. A function like `bravo()` might use `cd::report()` one time and `classic::report()` another time. This process of choosing which version of a function to use while the program is running is termed *dynamic linking*.

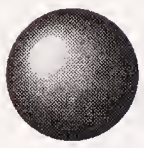
So far, this discussion hasn't brought us closer to our goal, but it has given a name to it—dynamic linking. (Some may consider giving a name to a goal work enough for one day, but we students of C++ are made of sterner stuff, so let's continue.) We want a program to be able to dynamically link a function name to a particular function definition as the program runs. The program statement

```
pcd->report();
```

thus would invoke the `cd::report()` method if `pcd` points to a *cd* object and invoke the `classic::report()` method if `pcd` points to a *classic* object. As the program ran, it would check to see to which type of object `pcd` pointed, then use the corresponding function definition.

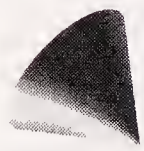
Although dynamic linking complicates life for the compiler, C++ makes implementing it simple for the user. When declaring member functions in a base class definition, use the C++ keyword `virtual` for functions that are to be linked dynamically. This makes the method a *virtual function*, meaning that the program will use dynamic linking instead of static linking for the function. You have to declare a function virtual only in the base class. If you subsequently redefine a virtual function in a derived class, it's automatically virtual, too. Only class member functions can be virtual functions; regular functions and friend functions don't qualify.





Destructors for a base class should be virtual functions. That way, if you delete an object identified by a pointer or reference, the program will first call the object destructor instead of the destructor associated with the pointer or reference type.

The term "virtual function" is meant to suggest that a virtual function acts like a place holder in a program, one that's replaced by a real function when the program runs. Normally, when the compiler (actually, the compiler-linker combination) sees a function call, it replaces it with code telling the program which code address to jump to in order to find that function (static linking). But with a virtual function, the compiler tells the program yes, there will be a function call here, but just relax. When the time comes, you'll be told which function to use (dynamic linking). Listing 11-17 modifies the *cd* class definition to use virtual functions. All it does is preface the *cd* prototypes for `report()` and `~cd()` with the keyword *virtual*.



LISTING 11-17 cd2.h

```
// cd2.h -- use a virtual function
#ifndef __CD__
#define __CD__

// base class
class cd {
protected:
    char * performers;
    char * label;
    int selections;
    double playtime;
public:
    cd(char * s1, char * s2, int n, double x);
    cd(const cd & d);
    cd();
    virtual ~cd();                          // virtual destructor
    virtual void report();                  // virtual method
    cd & operator=(const cd & d);
};


// derived class
class classic : public cd {
protected:
    char * works;
public:
```

```
    classic();
    classic(char * w, char * s1, char * s2, int n, double x);
    classic(const classic & d);
    ~classic();
    void report();                              // now virtual also
    classic & operator=(const classic & d);
};

#endif
```

The only other change you have to make is to include `cd2.h` instead of `cd1.h` in the methods file (Listing 11-15) and in the program file (Listing 11-16). After making those changes and running the program in the modified Listing 11-16, you should get this output:

```
Using object directly:
Performance by Beatles: Label = Capitol                    ← cd::report()
14 selections, playing time = 35.5

Works: Piano Sonata in B flat, Fantasia in C
Performance by Alfred Brendel: Label = Philips             ← classic::report()
2 selections, playing time = 57.17

Using type cd * pointer to objects:
Performance by Beatles: Label = Capitol                    ← cd::report()
14 selections, playing time = 35.5

Works: Piano Sonata in B flat, Fantasia in C               ← classic::report()
Performance by Alfred Brendel: Label = Philips
2 selections, playing time = 57.17

Hubba hubba!  (object passed by value)
Performance by Beatles: Label = Capitol                    ← cd::report()
14 selections, playing time = 35.5

Hubba hubba!  (object passed by value)
Performance by Alfred Brendel: Label = Philips             ← cd::report()
2 selections, playing time = 57.17

Bravo!  (object passed by reference)
Performance by Beatles: Label = Capitol                    ← cd::report()
14 selections, playing time = 35.5

Bravo!  (object passed by reference)
Works: Piano Sonata in B flat, Fantasia in C               ← classic::report()
Performance by Alfred Brendel: Label = Philips
2 selections, playing time = 57.17
```

Now the pointer invokes the method associated with the pointed-to object rather than the method associated with the pointer type. See Figure 11-6. Similarly, the reference in `bravo()` invokes the method associated with the referred-to object instead of the method associated with the reference variable type. This is dynamic linking at work.

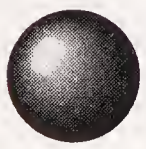
Note, however, that the `hubba()` function still uses the base class method for a derived class argument. That's because passing an argument by value generates a new object. So *disk* in `hubba()` is really a *cd* object. The `hubba()` function has no way of accessing the original *classic* object, so it can't use the *classic* method. Because passing an object by value can lose information about the type of argument passed, it's usually a poor idea.



When passing an object as an argument to a function, you usually should use a reference argument or a pointer argument rather than passing the object by value. That enables method calls within the function to use virtual member functions.

Dynamic linking extends C++'s function polymorphism by letting a program choose the most appropriate function definition during run time. Function overloading takes you part of the way. Indeed, it's all you need if your programs always have an object invoke its own methods directly. But if you have a base class and one or more derived classes, and if you wish or need to use pointers or references to handle both base and derived classes, then you need dynamic linking and virtual functions.

Now that you've seen how inheritance and virtual functions work, you can appreciate the following simple rules:



- **If you want class members to be available to derived classes, make them *protected* members instead of private members.**
- **If you expect to redefine a function in derived classes, declare it as a virtual function in the base class; this allows objects accessed via a base class pointer or reference to use the class methods appropriate to the object type instead of those of the pointer or reference type.**
- **Declare the destructor of a base class as a virtual function.**

This concludes the section of the chapter on simple inheritance. Next, you'll learn about ways to combine two or more classes into a new class. As you might suspect, that's

a major new topic, so if you've been waiting for a chance to grab a sandwich, to take a nap, or read a few articles in *Golf and Dog*, do it now before going on.

Classes with Member Classes

You may find you need a data type having the properties of two other classes. C++ offers two ways to combine class properties. One is to incorporate one or more classes as members of another class. We'll take that up in this section. The other, more powerful, method is using multiple inheritance, in which a derived class has more than one base class. We'll take that up upon finishing with member classes.

Illustrating either approach of combining two classes requires having two classes to combine. The *darray* class can serve as one, and we can develop a new class for the second. The *person* class holds and displays the first and last name of a person. Listing 11-18 shows the class definition.

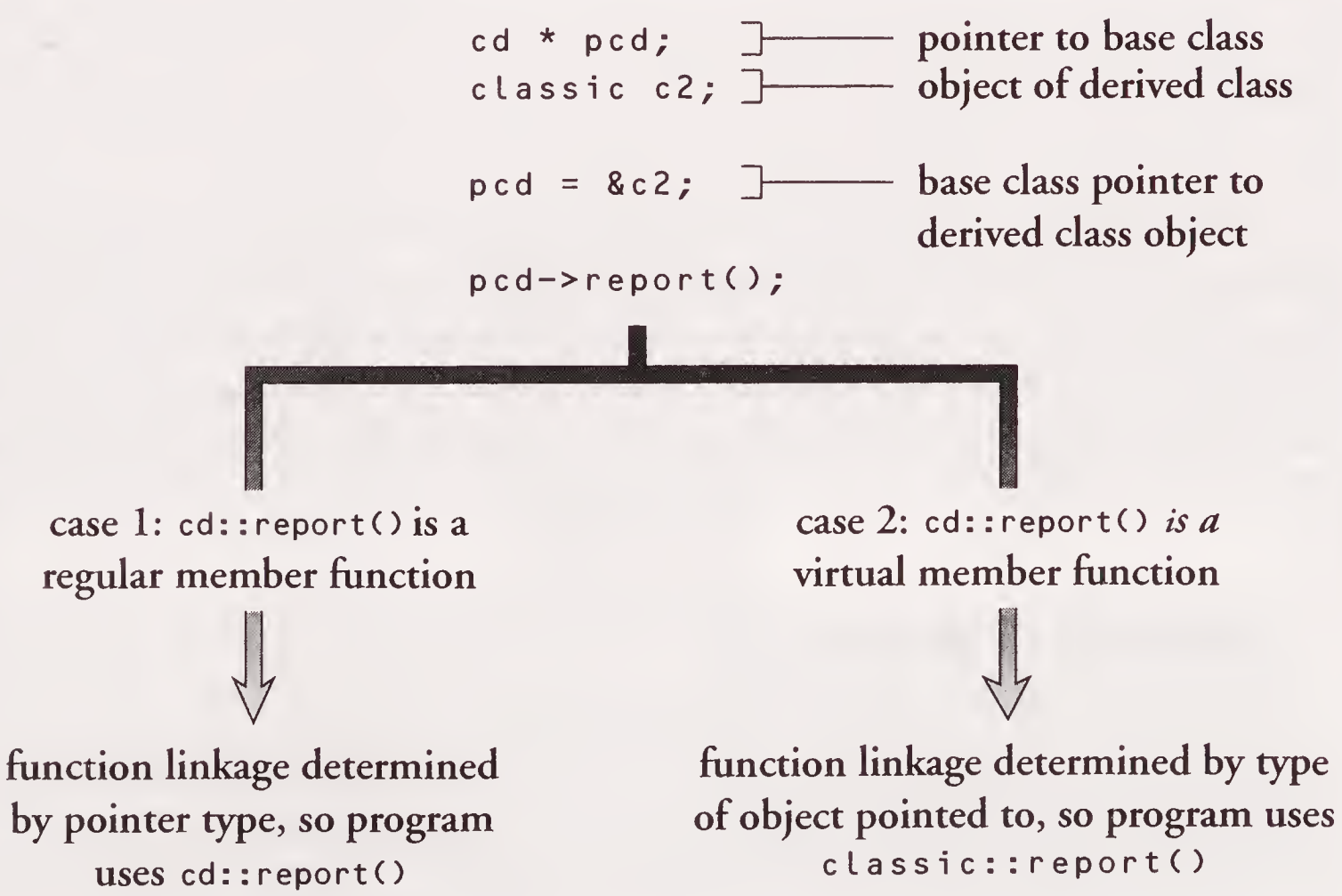


Figure 11-6 *Regular member function versus virtual member function*



LISTING 11-18 person.h

```
// person.h -- a class for a person's name
#ifndef __PERSON__
#define __PERSON__

#include <iostream.h>

class person
{
protected:
    char * fname;
    char * lname;
public:
    person();
    person(const char * sf, const char * sl);
    person(const person & p);
    virtual ~person();
    char * first();              // return first name
    char * last();               // return last name
    void lastfirst();            // display last, first
// overloaded operators
    virtual person & operator=(const person & p);
    friend ostream & operator<<(ostream & os,
                                const person & p);
};

#endif
```

The constructors use `new`, so once again the class definition must supply the usual supporting cast of destructor, initializing-to-an-object constructor, and assignment operator methods. Also, it supplies a few methods for accessing the names in different ways. Because a later example will use *person* as a base class, the class destructor is declared `virtual`. Similarly, the new `operator=()` function is a virtual function because it will later be redefined in a derived class. Listing 11-19 shows the *person* class methods.



LISTING 11-19 person.cpp

```
// person.cpp -- person class methods

#include <iostream.h>
#include <string.h>
#include "person.h"

person::person()
```

```
{
    fname = NULL;
    lname = NULL;
}

person::person(const char * sf, const char * sl)
{
    fname = new char [strlen(sf) + 1];
    strcpy(fname, sf);
    lname = new char [strlen(sl) + 1];
    strcpy(lname, sl);
}

person::person(const person & p)
{
    fname = new char [strlen(p.fname) + 1];
    strcpy(fname, p.fname);
    lname = new char [strlen(p.lname) + 1];
    strcpy(lname, p.lname);
}

person::~person()
{
    delete fname;
    delete lname;
}

char * person::first()
{
    return fname;
}

char * person::last()
{
    return lname;
}

void person::lastfirst()
{
    cout << lname << ", " << fname;
}

person & person::operator=(const person & p)
{
    if (this == &p)       // if object is assigned to itself,
        return *this;     // delete nothing, and return it
    delete fname;
    delete lname;
```



```
    fname = new char [strlen(p.fname) + 1];
    strcpy(fname, p.fname);
    lname = new char [strlen(p.lname) + 1];
    strcpy(lname, p.lname);
    return *this;
}

ostream & operator<<(ostream & os, const person & p)
{
    os << p.fname << " " << p.lname;
    return os;
}
```

MEMBER CLASSES

Now we're ready to create a class for which some members are themselves objects. The *salesrep* class, defined in Listing 11-20, has three members. The first is an `int` identifying a sales district. The second is a *person* object identifying a sales representative. The third is a *darray* object representing sales figures. Note that you declare member classes just as you declare any other type. After all, the *person* and *darray* classes are user-defined types. The definition supplies three constructors. One is the default constructor, and the second initializes a *salesrep* object to a region, individual names, and an array size and an initialization value for the array elements. The third constructor allows you to use a *person* object and a *darray* object in the initialization. You probably can dream up other constructors that would make the class more flexible in use. However, they wouldn't make the example any clearer, so we'll stick to the three.



LISTING 11-20 salesrep.h

```
// salesrep.h -- class with two member classes

#ifndef __SALESREP__
#define __SALESREP__
#include <iostream.h>
#include "person.h"
#include "darray.h"

class salesrep
{
protected:
    int region;          // regional ID
    person repname;      // a person class member
    darray sales;        // a darray class member
```

```
public:
    salesrep();
    salesrep(int reg, const char * str1, const char *str2,
            unsigned size, double val = 0.0);
    salesrep(int reg, const person & rep, const darray & sls);
    ~salesrep();
// operator
    friend ostream & operator<<(ostream & os,
                                const salesrep & sr);
};

#endif
```

MEMBER CLASS INITIALIZATION FOR CLASSES WITH OBJECT MEMBERS

So far, creating member classes has been no different than creating members of built-in types. Where member classes get a little tricky is handling the initialization process in the constructor definitions. Creating a *salesrep* object requires a *salesrep* constructor. The `repname` and the `sales` members also are objects and hence require their own constructors. C++ provides that the constructors for the member classes are called first. This makes sense, for the program needs to create these objects before incorporating them into a *salesrep* object. Here's the tricky part: although the member constructors have to be called first, it's the *salesrep* constructor that has the arguments. So, as was the case with inheritance, you need a mechanism to pass arguments from one constructor to another. Again, you can do that in the member function definitions. For example, consider the following constructor prototype:

```
salesrep(int reg, const char * str1, const char *str2,
            unsigned size, double val = 0.0);
```

The constructor needs to pass the two string arguments to the *person* constructor and the final two arguments to the *darray* constructor. It does so as follows:

```
salesrep::salesrep(int reg, const char * str1, const char *str2,
            unsigned size, double val) : repname(str1, str2),
            sales(size, val)
{
    region = reg;
}
```

The part following the single colon is an *initialization list.* As you can see, you can initialize more than one member by using a comma-separated list of items. The expression `repname(str1, str2)` means initialize the `repname` member with the values `str1` and `str2`. Because the `repname` member is a *person* object, the compiler will use the



corresponding *person* constructor for the initialization. Similarly, it uses a *darray* constructor to initialize the `sales` object.

You can use the initialization list for *any* member, not just user-defined members. Just use the member name followed by the initialization value enclosed in parentheses. For instance, here's one way to implement the third constructor:

```
salesrep::salesrep(int reg, const person & rep,
                                  const darray & sls)
            : region(reg), repname(rep), sales(sls)
{
}
```

This initializes each of the three members (`region`, `repname`, and `sales`) to an appropriate value, leaving the function body with nothing to do.

INITIALIZATION AND ASSIGNMENT IN CONSTRUCTORS

Let's take a closer look at the initialization process for constructors. In general, a constructor has two parts: an initialization section and an assignment section. The initialization section is located between a colon after the constructor name and before the opening brace of the function definition. It consists of a comma-separated list. The assignment section consists of the statements between the opening and closing braces of the function definition. See Figure 11-7. You can omit either section if you don't need it.

The chief difference between the two parts is that actions in the initialization section take place *before* the body, or assignment section, of the constructor is executed. For some cases, this distinction is not important. For instance, one of our examples initialized the `region` member in the assignment section, and one initialized it in the initialization section. But some actions can be performed only in the initialization list:

- Any class member that is itself a class object whose constructor requires arguments must be initialized in the initialization section.
- If a derived class passes arguments to a base class constructor, it should do so in the initialization section. That's because the base class constructor is called before the derived class constructor.
- Any class member that is either `const` or a reference can be initialized only in the initialization section.

If a member class has a default constructor, a C++ program will call that constructor automatically if you don't provide a constructor in the initialization section. That's okay if you don't need to provide initialization values. But don't do the following:

```
salesrep::salesrep(int reg, const person & rep,
                                  const darray & sls)
```

```
{
    region = reg;                   // fine
    person repname(rep);            // poor idea
    darray sales(sls);              // ditto

}
```

This code would cause the program to call the default constructors for the *person* and *darray* objects during the initialization phase, then call the object-initialization constructors during the assignment phase. The original version calls the object-initialization constructors directly during the initialization phase, so it's more efficient than this version, which initializes `repname` and `sales` twice.

COMPLETING THE CLASS DEFINITION

Now we can complete the work of defining the *salesrep* class by defining the class methods. Listing 11-21 shows the result. Notice that you don't have to redefine assignment. Recall how assignment works. If you don't define an assignment operator, a C++ program uses the default version of assignment, in which each member of the target object is an exact copy of the source object. However, and this is new information, if a particular member is itself an object, C++ checks to see if you've defined assignment for that object's class. If so, the program uses the class assignment method for that particular

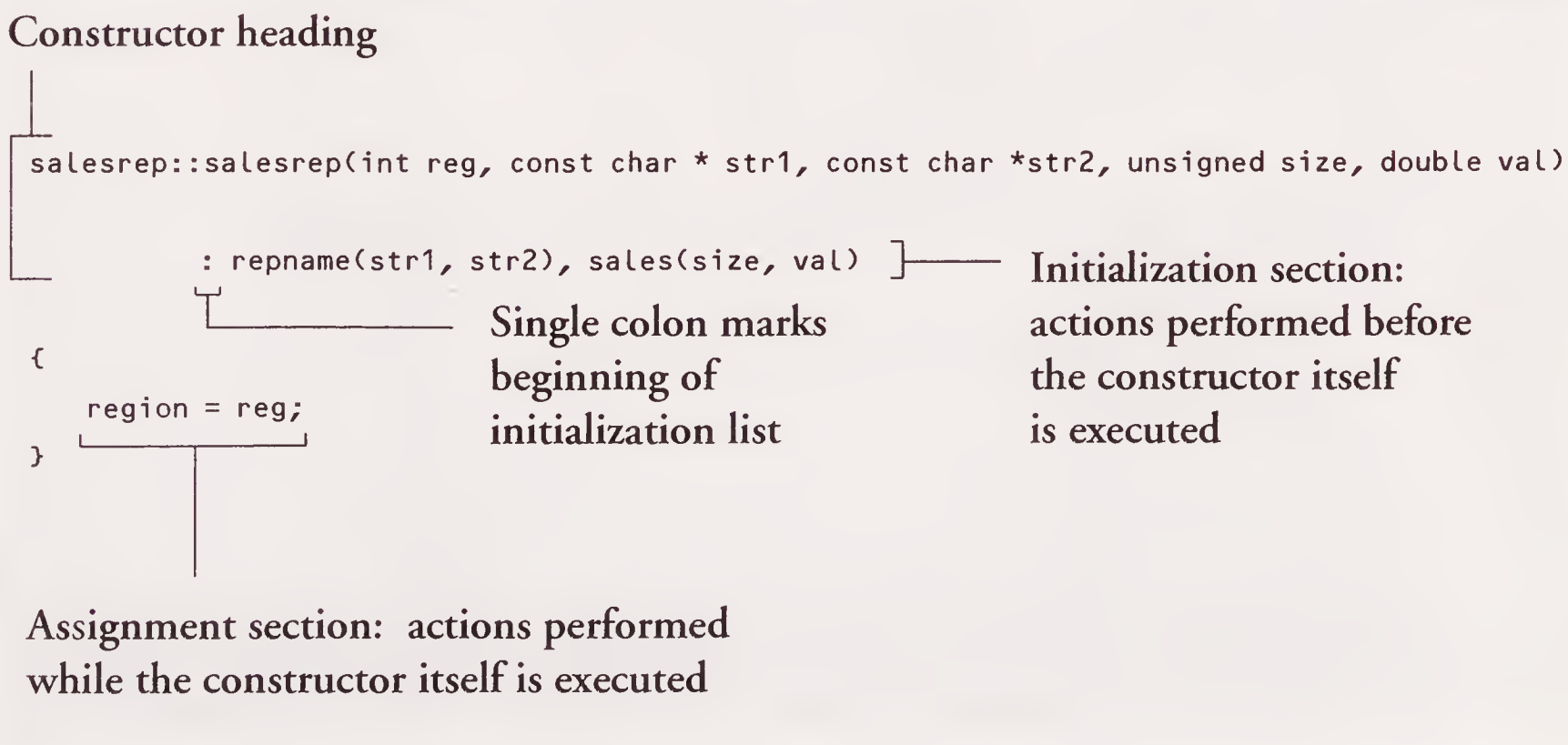


Figure 11-7 *Initialization and assignment sections of a constructor*



member. If not, the program uses the default assignment for that member. The *salesrep* class didn't define assignment, so the program simply copies one `region` member to the other. But the *darray* and *person* classes do define assignment operators, so the program uses those methods to handle the `repname` and `sales` members. See Figure 11-8.



LISTING 11-21 salesrep.cpp

```
// salesrep.cpp -- class methods
#include "salesrep.h"

salesrep::salesrep()
{
    region = 0;
}

salesrep::salesrep(int reg, const char * str1, const char *str2,
          unsigned size, double val) : repname(str1, str2),
          sales(size, val)
{
    region = reg;
}

salesrep::salesrep(int reg, const person & rep,
                              const darray & sls)
          : region(reg), repname(rep), sales(sls)
{
}

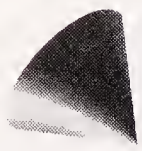
salesrep::~salesrep()
{
}

ostream & operator<<(ostream & os, const salesrep & sr)
{
    os << "Region " << sr.region << ": "
       << sr.repname << "\n" << sr.sales;
    return os;
}
```

Listing 11-22 provides a short test for the new class. It uses the two nondefault initializers and the overloaded << operator. Because it uses the *darray*, the *person,* and the *salesrep* classes, you should compile the three files `darray.cpp`, `person.cpp`, `salesrep.cpp` and `two_in_1.cpp`.

Figure 11-8 *Assignment for the* salesrep *class*





LISTING 11-22 two_in_1.cpp

```
// two_in_1.cpp -- using a class with class members
// uses darray.cpp, person.cpp, and salesrep.cpp
#include <iostream.h>
#include "salesrep.h"
int main(void)
{
    person perkie("Perkie", "Plonk");
    darray coffee(5, 120.0);
    salesrep plonk(4, perkie, coffee);
    salesrep brock(2, "Boomer", "Brock", 4, 30.0);

    cout << plonk;
    cout << brock;
    cout << "Assigning plonk to brock and displaying brock:\n";
    brock = plonk;
    cout << brock;

    return 0;
}
```

Here is the program output:

```
Region 4: Perkie Plonk
120 120 120 120 120
Region 2: Boomer Brock
30 30 30 30
Assigning plonk to brock and displaying brock:
Region 4: Perkie Plonk
120 120 120 120 120
```

DISADVANTAGES OF MEMBER CLASSES

As you've seen, you can use a member class much as you can use a member of a basic type. However, sad to say, you can run into problems if you want to use the methods of the member class. Suppose you want to access the first name member of the *repname* object member of the *plonk* object. Can you use the `first()` function from the *person* class? No, you can't. For instance, consider the following statement:

```
cout << plonk.repname.first();
```

Certainly, *plonk.repname* is a *person* object, hence entitled to use the `person::first()` method. But *plonk.repname* is a `protected` member of the *salesrep* class, hence you can't use it publicly. The only identifier you can use is `plonk`. You can use *salesrep*

methods to access `plonk` members, and the `plonk` member functions can use *person* methods to access *person* members. (The overloaded `<<` operator does so, for example.) But you can't use the *person* methods directly unless you make the `repname` member `public` rather than `protected`. However, making data members public is contrary to the usual OOP philosophy. Thus if you want to keep the *person* member `protected`, and you still want to use the *person* methods, you have to supply a *salesrep* method for each *person* method you want available. For example, to access the `first()` method, you could do this:

```
char * salesrep::first()
{
    return repname.first();
}
```

Then using `plonk.first()` (a *salesrep* method) invokes `plonk.repname.first()` (a *person* method). However, multiple inheritance provides a better solution to this sort of problem, so let's look at it now.

MULTIPLE INHERITANCE

Multiple inheritance means deriving a class from more than one base class. With multiple inheritance, the new class inherits all the data members and all the class methods of all the base classes. Therefore an object of the new class can use the base class methods directly, unlike the case of a class with object members. Multiple inheritance allows a derived class to combine the strengths of other classes, much as the offspring of Rin Tin Tin and C3PO might inherit the courage and sense of smell of the former and the language skills of the latter. C++ added multiple inheritance to its feature list when version 2.0 was released, so you'll need that or a later version of C++. Let's look at the process. We'll let the sales reps go and invest in education instead, developing a class to track students and their courses. Listing 11-23 presents a *student* class definition that inherits from both the *person* and the *darray* classes and also adds a new member to represent a course name.

LISTING 11-23 student.h

```
// student.h -- multiple inheritance

#ifndef __STUDENT__
#define __STUDENT__
#include <iostream.h>
#include "person.h"
```



```
#include "darray.h"

// inherit from both the person class and the darray class
class student : public person, public darray
{
protected:
    char * course;
public:
    student();
    student(const char * cl, const char * str1, const char *str2,
            unsigned size, double val = 0.0);
    student(const char * cl, const person & s,
                    const darray & qs);
   student(const student & st);
    ~student();
// operators
    student operator=(const student & st);
    friend ostream & operator<<(ostream & os,
                                const student & st);

};

#endif
```

To indicate which classes should be base classes, just list them after a colon following the class name:

```
class student : public person, public darray
{
   ...
}
```

This provides the *student* class with `fname` and `lname` members from the *person* class and `arr` and `size` members from the *darray* class. To them the definition adds a `course` member to represent a course name.

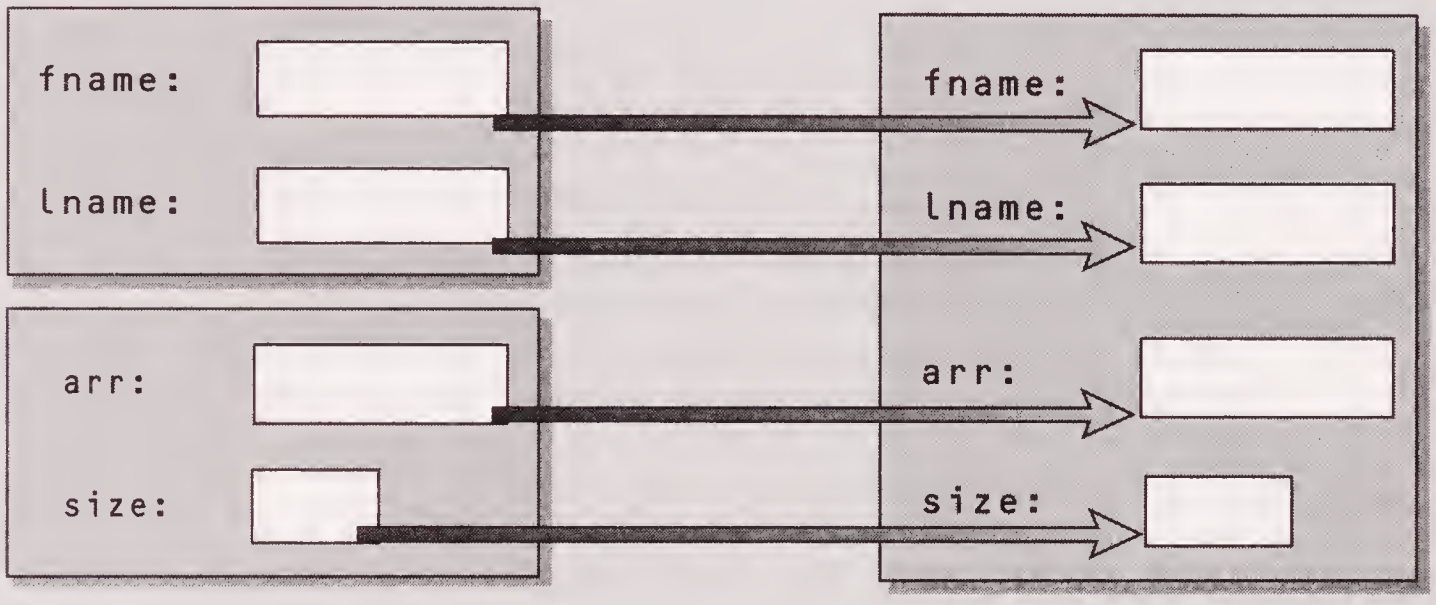
With multiple inheritance, each member of the base classes becomes a member of the derived class. This contrasts with using member classes. That method, as you saw, resulted in each member of the base class becoming a member of a member of the derived class. In short, the base class members were nested in a derived class member. See Figure 11-9.

As usual with inheritance, you have to provide constructors. Also, as usual, you have to provide functions to support the `course` member, which will use memory allocated by `new`. Listing 11-24 shows the class methods.

Multiple Inheritance

```
class inher_class : public person, public darray
{
    ...
};
```

new class inherits members of base classes



Member Classes

```
class mem_class
{
protected:
    person boss;
    darray reps;
    ...
};
```

new class contains entire objects of other classes

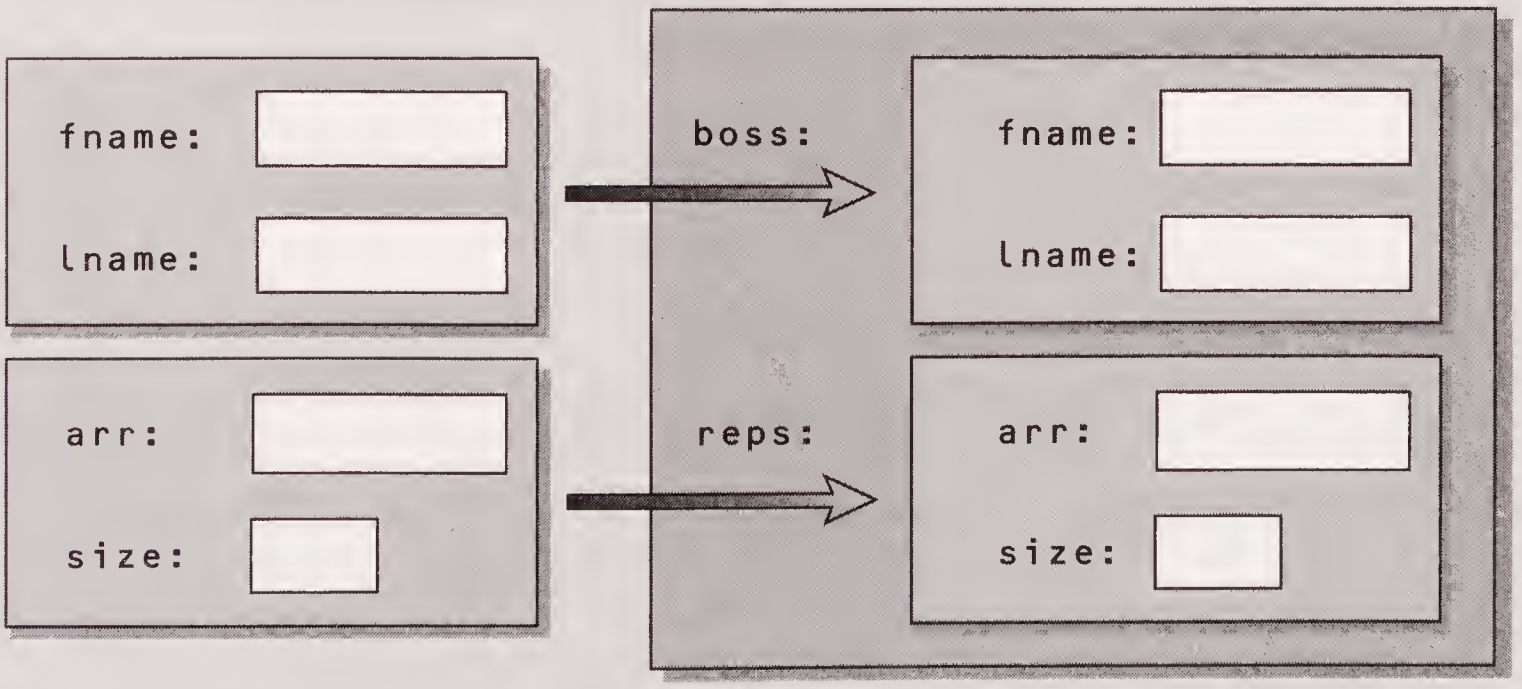


Figure 11-9 *Multiple inheritance versus member classes*



LISTING 11-24 student.cpp

```
// student.cpp -- student class methods

#include <iostream.h>
#include <string.h>
#include "student.h"

student::student()
{
    course = NULL;
}

// pass arguments to person, darray constructors
student::student(const char * cl, const char * str1,
          const char *str2, unsigned size, double val) :
          person(str1, str2), darray(size, val)
{
    course = new char[strlen(cl) + 1];
    strcpy(course, cl);
}

// pass arguments to person, darray constructors
student::student(const char * cl, const person & s,
                 const darray & qs) : person(s), darray(qs)
{
    course = new char[strlen(cl) + 1];
    strcpy(course, cl);
}

student::student(const student & st)
{
    course = new char[strlen(st.course) + 1];
    strcpy(course, st.course);
}

student::~student()
{
    delete course;
}

student student::operator=(const student & st)
{
    if (this == &p)       // if object is assigned to itself,
        return *this;     // delete nothing, and return it
    delete course;
    course = new char[strlen(st.course) + 1];
    strcpy(course, st.course);
```

```
    delete fname;
    fname = new char[strlen(st.fname) + 1];
    strcpy(fname, st.fname);

    delete lname;
    lname = new char[strlen(st.lname) + 1];
    strcpy(lname, st.lname);

   size = st.size;

    delete arr;
    arr = new double[size];
    for (int i = 0; i < size; i++)
          arr[i] = st.arr[i];

    return *this;

}

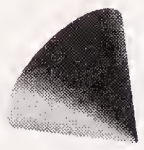
ostream & operator<<(ostream & os, const student & st)
{
    os << "Class: " << st.course << "\n";
    os << "Student: " << st.fname << " " << st.lname << "\n";
    os << "Quizzes:\n";
    os << darray(st);              // use darray::operator<<()
    return os;
}
```

The process is much like single inheritance, except that now you have to pass information to constructors for two base classes:

```
student::student(const char * cl, const person & s,
                 const darray & qs) : person(s), darray(qs)
{
    course = new char[strlen(cl) + 1];
    strcpy(course, cl);
}
```

As with single inheritance, you use the initialization section instead of the assignment section to pass arguments to the base constructors. Note that with multiple inheritance, you use the *constructor* name in the initialization section, while for member objects, as in Listing 11-21, you used the *member object* name in the initialization section.

Listing 11-25 provides a short program using the new class. Each of the three classes (*student*, *person*, *darray*) has overloaded the << operator, and the program shows how to summon all three. This program uses `darray.cpp`, `person.cpp`, `student.cpp`, and `inherit.cpp`, so compile all four files.



LISTING 11-25 inherit.cpp

```
// inherit.cpp -- use multiple inheritance
// uses darray.cpp, person.cpp, and student.cpp
#include <iostream.h>
#include "student.h"

int main(void)
{
    student dorp("AI Programming", "Dweebie", "Dorp", 5);

// use operator[]() from darray class
    for (int i = 0; i < 5; i++)
            dorp[i] = 15 + i % 3;

// use first() from person class
    cout <<"Let's see how " << dorp.first() << " is doing.\n";

//  use three versions of operator<<()
    cout << "student version:\n" << dorp;
    cout << "person version: " << person(dorp) << "\n";
    cout << "darray version: "<< darray(dorp);

    cout << "Have you hugged your AI today? Bye.\n";
    return 0;
}
```

Here is the program output:

```
Let's see how Dweebie is doing.
student version:
Class: AI Programming
Student: Dweebie Dorp
Quizzes:
15 16 17 15 16
person version: Dweebie Dorp
darray version: 15 16 17 15 16

Have you hugged your AI today? Bye.
```

PROGRAM NOTES

Because *student* inherits from two classes, it can use methods from both. For example, the `for` loop uses the subscripting operator from the *darray* class to access elements in the *dorp* object. And one of the output statements uses `first()` from the *person* class.

The situation grows more complicated when you have multiple versions of a method. For instance, all three classes overload the << operator to handle output. As usual, C++ uses the function signature to resolve ambiguities. A type `student` argument invokes the friend function defined with the `student` methods:

```
Class: AI Programming
Student: Dweebie Dorp
Quizzes:
15 16 17 15 16
```

By using type casts, you can convert the object type, thus invoking other versions of the << operator. The following two lines illustrate this point:

```
cout << "person version: " << person(dorp) << "\n";
cout << "darray version: " << darray(dorp) << "\n";
```

The first of these lines converts `dorp` to a *person* object, thus causing the program to use the *person* version of output. Similarly, the second line converts `dorp` to a *darray* object and uses the *darray* display method. (Recall that you can always convert a derived type to a base type, although the process may entail losing access to some data.)

Here's a question: What happens if each base class has a member function by the same name? Suppose, for instance, the *darray* class had a `first()` member function that returns the first element of an array. The *person* class, of course, already has a `first()` function. What would happen then if you tried to invoke `first()`?

```
cout << dorp.first();
```

Would this print the first element of the array or the student's first name? The answer is neither. Instead, the compiler would complain of ambiguity. In that case, you can use the scope operator to indicate which inherited method you wish to use:

```
cout << dorp.person::first();  // print first name
cout << dorp.darray::first();  // print first element
```

Multiple Inheritance with a Common Ancestor

The *student* class was based on two classes—*darray* and *person*—that were unrelated to each other. You can concoct more involved inheritance schemes, such as using base classes that themselves have a common ancestor. As often seems to be the case with classes, introducing a new complexity also introduces new problems. In this case, the default multiple inheritance mechanism doesn't necessarily work well when both base classes themselves share a common ancestor. For instance, suppose you wanted a class that combined the subscripting features of the *lim_dar* class with the arithmetic methods of the *arithdar* class. Then you could use multiple inheritance as follows:

```
class limarith: public  lim_dar, public arithdar
{
    ...
```

The *limarith* class would then inherit the *arithdar* arithmetic methods and the *lim_dar* subscript checking and lower-bound setting. That's good. The problem is that both base classes descend from the *darray* class, inheriting via two separate paths. This means a *limarith* object would inherit two complete sets of *darray* members. It would have two `arr` members and two `size` members. See Figure 11-10. Furthermore, because creating an object invokes constructors for all the ancestor classes, creating a single *limarith* object would invoke the corresponding *darray* constructor twice.

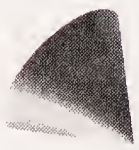
VIRTUAL BASE CLASSES

C++ offers a way to avoid the confusion created by multiple inheritance from the same class, and that is to use a *virtual base class.* Making a class a virtual base class causes a C++ program to incorporate only one copy of that class into a descendent class, even if that class descends by more than one line from the virtual class. See Figure 11-11.



If, through multiple inheritance, a derived class has a base class as a multiple ancestor, an object of the derived class normally incorporates multiple instances of an object of the base class. But if the base class is a *virtual* ancestor, the derived class incorporates just one object of the base class.

To make a base class virtual, just include the keyword `virtual` when listing the base class in a class definition. For instance, to make *darray* a virtual base class for the *arithdar* class, change the class header file to look as shown in Listing 11-26. Nothing else in the class definition nor in the methods definitions need be changed.



LISTING 11-26 v_ar_dar.h

```
// v_ar_dar.h -- derived array class with more arithmetic
// modify arithdar.h to make darray a virtual base class

#ifndef __ARITHDAR__
#define __ARITHDAR__

#include "darray.h"
```

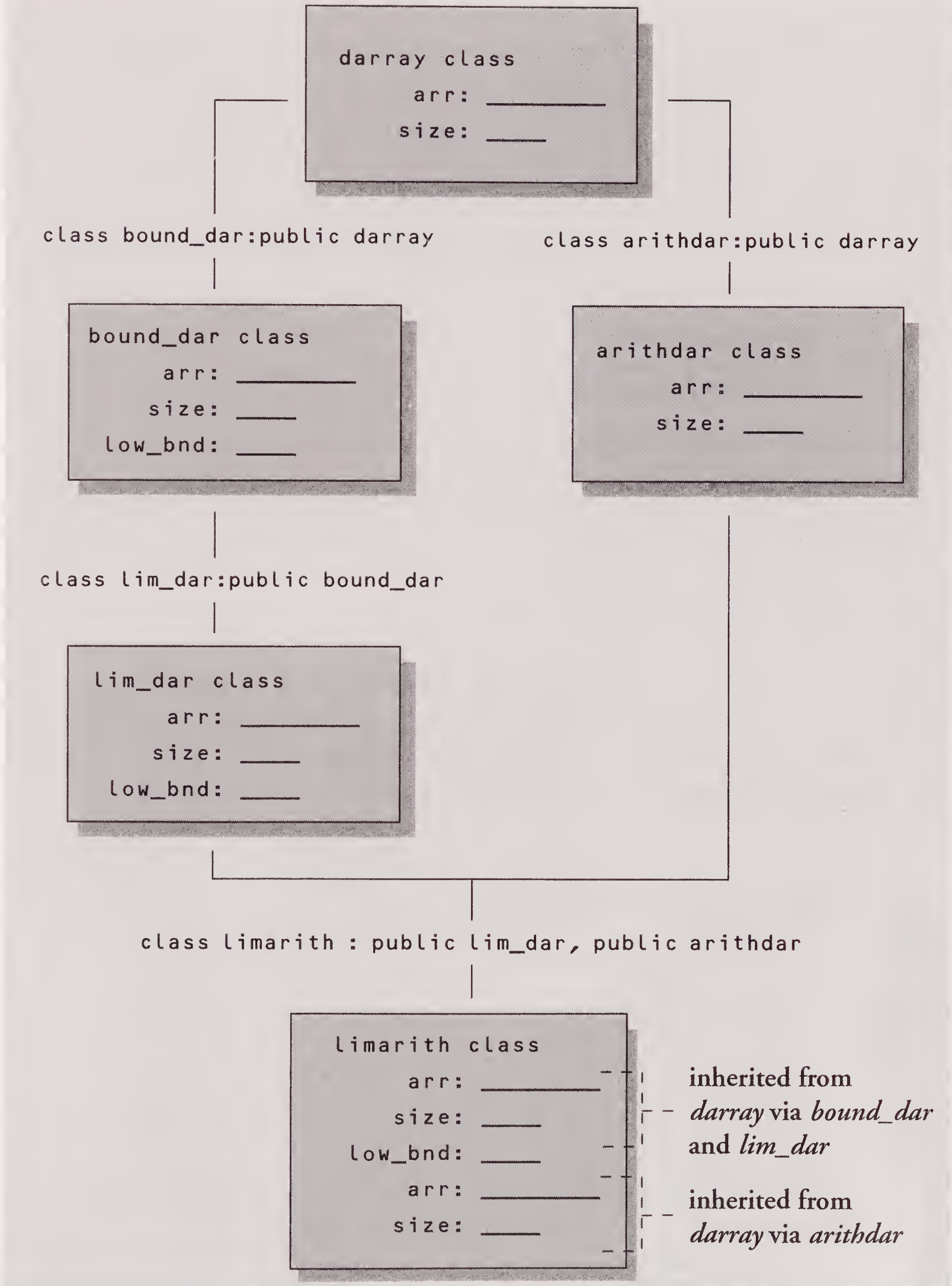


Figure 11-10 *Multiple inheritance from the same ancestor class*

```
class arithdar : virtual public darray // darray a virtual base
{
public:
// constructors not inherited, so must be defined
    arithdar();
    arithdar(unsigned int n, double val = 0.0);
    arithdar(const double *pn, unsigned int n);
    arithdar(const arithdar & a);
    arithdar(const darray & a);

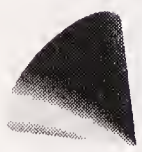
// destructor is inherited, but you can define a new one
    ~arithdar();

// new methods
    double sum();
    double average();

// we leave the definition of overloaded operators, such
// as addition, subtraction, sign reversal, multiplication
// by a single value, etc., as an exercise for the reader
};

#endif
```

Similarly, you can make *darray* a virtual base class for the *bound_dar* class. Because the *lim_dar* class descends from the *bound_dar* class, this makes *darray* a virtual ancestor to *lim_dar*, too. Listing 11-27 shows the modified header file. Note that the keywords *public* and *virtual* can be used in either order.



LISTING 11-27 vbnddar.h

```
// vbnddar.h -- virtual version of bnddar.h
#ifndef __BNDDAR__
#define __BNDDAR__

#include "darray.h"

class bound_dar : public virtual darray // virtual base class
{
protected:
    void ok(int i);                          // index checker
public:
// constructors
    bound_dar();
    bound_dar(unsigned int n, double val = 0.0);
```

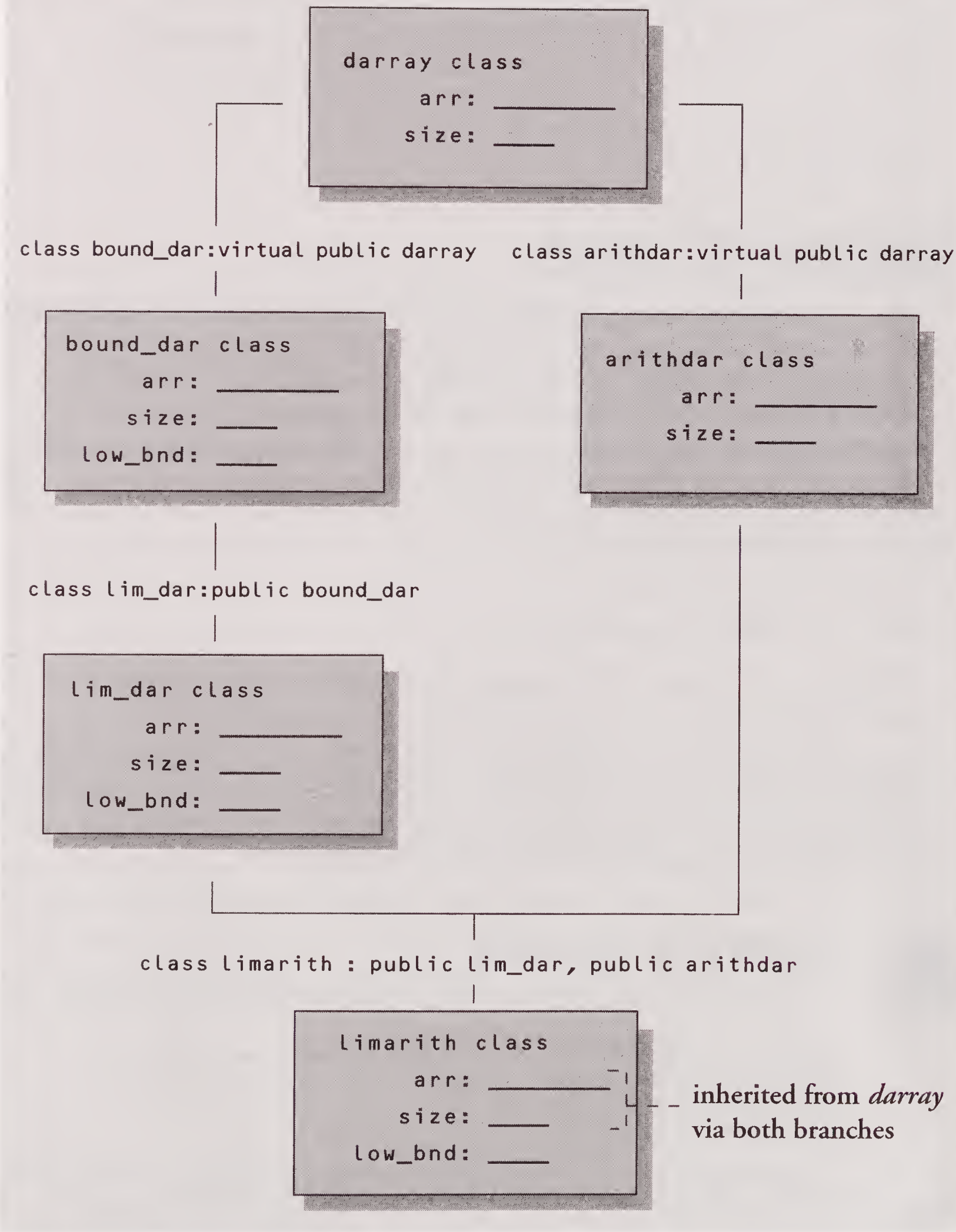


Figure 11-11 *Multiple inheritance from a virtual base class*

```
    bound_dar(const double *pn, unsigned int n);
    bound_dar(const bound_dar & a);
    bound_dar(const darray & a);
// operators
    double & operator[](int i);        // redefined

};

#endif
```

USING VIRTUAL BASE CLASSES

You use virtual base classes much as you use regular base classes. The only difference is in designing constructors. To see how that works, let's first define a new class that inherits from the *darray* class via two different routes. Listing 11-28 defines a *limarith* class derived from the *arithdar* and the *lim_dar* classes. Using the new `v_ar_dar.h` and `vbnddar.h` header files (the ones that use the *virtual* modifier for *darray*) makes *darray* a virtual base class. The `vlim_dar.h` header file is just the `lim_dar.h` file with

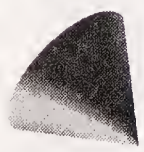
```
#include "bnddar.h"
```

replaced by the following:

```
#include "vbnddar.h"
```

Similarly, you should modify `bnddar.cpp` to include `vbnddar.h` instead of `bnddar.h` and modify `arithdar.cpp` to include `v_ar_dar.h` instead of `arithdar.h`.

The *lim_dar* and *arithdar* classes don't have to be virtual base classes, because *limarith* inherits only one copy of each. Only classes inherited more than once, such as *darray*, need be virtual. The new class doesn't use any data or methods beyond those of its ancestors, so the only methods it needs are constructors.



LISTING 11-28 limarith.h

```
// limarith.h -- inherit from lim_dar and public arithdar

#ifndef __LIMARITH__
#define __LIMARITH__
#include "v_ar_dar.h"
#include "vlim_dar.h"

class limarith: public  lim_dar,  public arithdar
{
public:
```

```
    limarith();
    limarith(unsigned int n, double val = 0.0);
    limarith(unsigned int n, int lb, double val = 0.0);
    limarith(const double *pn, unsigned int n);
    limarith(const limarith & a);
    limarith(const lim_dar & a);
    limarith(const arithdar & a);
};

#endif
```

Now we come to a tricky part, the part that distinguishes using a virtual base class from using a normal base class. Consider the following constructor:

```
limarith(unsigned int n, double val = 0.0);
```

This initializes the `size` and `arr` members of the object. These members are inherited from the *darray* class, so the arguments `n` and `val` should ultimately be passed back to the *darray* constructor. Recall how that worked for single inheritance. A typical *lim_dar* class constructor, for example, used the initialization section to pass arguments back to the *bound_dar* constructor, and that constructor passed arguments back to the *darray* constructor:

```
lim_dar::lim_dar(unsigned int n, double val) : bound_dar(n, val)
{
    low_bnd = 0;  // default value
}
```

Similarly, the *arithdar* constructor passed arguments back to the *darray* constructor. But the present situation provides two distinct paths back to the *darray* constructor. Which inheritance path—via *lim_dar* and *bound_dar* or via *arithdar*—is used to pass arguments back? And won't the *darray* constructor be called twice, once for each path? C++ bypasses these questions by letting a derived class pass arguments directly back to a virtual base class, skipping intermediate classes. Normally, you can pass arguments back only to an immediate ancestor, such as *lim_dar* or *arithdar*, but that's not true for virtual base classes. Thus you can write the constructor this way:

```
limarith::limarith(unsigned int n, double val)
    : darray(n, val)     // pass values directly to darray()
{
}
```

This sends the constructor arguments directly to the *darray* constructor.

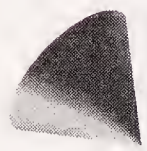
Sending arguments directly to a virtual ancestor class usurps any argument passing an intermediate class may have done otherwise. For example, consider the following definition:



```
limarith::limarith(unsigned int n, int lb, double val)
    : darray(n, val), lim_dar(n, lb, val)
{
}
```

The expression `darray(n, val)` takes care of passing arguments to the *darray* class. The `lim_dar(n, lb, val)` expression passes the indicated arguments to the *lim_dar* constructor. It then can use these arguments, but the program will ignore the part of the *lim_dar* constructor definition that says to pass the `n` and `val` arguments to its *darray* ancestor.

Listing 11-29 lists all the methods for the *limarith* class.



LISTING 11-29 limarith.cpp

```
// limarith.cpp -- class methods
#include <iostream.h>
#include "limarith.h"

limarith::limarith()
{
}

limarith::limarith(unsigned int n, double val)
    : darray(n, val)
{
}

limarith::limarith(unsigned int n, int lb, double val)
    : darray(n, val), lim_dar(n, lb, val)
{
}

limarith::limarith(const double *pn, unsigned int n)
    : darray(pn, n)
{
}

limarith::limarith(const limarith & a) : darray (a), lim_dar(a)
{
}

limarith::limarith(const lim_dar & a) : darray (a), lim_dar(a)
{
}

limarith::limarith(const arithdar & a) : darray(a)
{
```

```
    low_bnd = 0;
}
```

Finally, Listing 11-30 gives us the opportunity to see how this approach works. This program uses several source code files: `bi_inher.cpp`, `limarith.cpp`, `vlim_dar.cpp`, `vbnddar.cpp`, `v_ar_dar.cpp`, and `darray.cpp`. Don't forget to modify `bnddar.cpp` and `lim_dar.h` to include `vbnddar.h` instead of `bnddar.h` and to modify `arithdar.cpp` to include `v_ar_dar.h` instead of `arithdar.h`. The new header files, recall, identify *darray* as a virtual base class.

LISTING 11-30 bi_inher.cpp

```
// bi_inher.cpp -- use limarith.h
// program uses limarith.cpp, vlim_dar.cpp, v_ar_dar.cpp,
// v_bnddar.cpp, and darray.cpp
#include <iostream.h>
#include "limarith.h"
int main()
{
    limarith chow_masters(4, 1987, 1235.50);
    cout << "Sales for Tricks of the Chowdown Masters:\n";
    for (int year = 1987; year < 1991; year++)
        cout << year << ": " << chow_masters[year] << "\n";
    cout << "Total sales: " << chow_masters.sum() << "\n";
    cout << "Those sales again: " << chow_masters;
    cout << "Year 1985: " << chow_masters[1985];
    cout << "Bye!";
    return 0;
}
```

Here is the output:

```
Sales for Tricks of the Chowdown Masters:
1987: 1235.5
1988: 1235.5
1989: 1235.5
1990: 1235.5
Total sales: 4942
Those sales again: 1235.5 1235.5 1235.5 1235.5
Error in array limits:
index 1985 less than 1987
```

As you can see, the program used the array summing method inherited from *arithdar* and the subscripting features inherited from *lim_dar*.



If a class winds up inheriting from the same base class by two different paths of derivation, you usually should make the base class a virtual ancestor. You do that by using the keyword `virtual` in the base class list found in the definitions of the classes immediately derived from the base class.

WHICH METHOD?

You may be wondering how the program knew to use the `operator[]()` function defined in *lim_dar* rather than the version used by *arithdar*. The answer is the same as for single inheritance: C++ uses the most recently defined version. That's recent in terms of ancestry, not in terms of time. Thus if we had defined `operator[]()` for the *limarith* class, the program would have used that version. Failing to find such a version, the program next looks at the immediate ancestors. The *lim_dar* class defines the `operator` function, while the *arithdar* class uses the `operator` definition inherited from *darray*. Thus the *lim_dar* class has the most recent `operator[]()` definition, and that's the one used.

Suppose, however, that the *arithdar* class also defined `operator[]()`. Then both versions would be equally recent, and the compiler would have terminated compilation, reporting an ambiguity. You can resolve that ambiguity by using the scope operator:

```
chow_masters.lim_dar::operator[](year)
```

However, that solution loses the elegance of the original notation, and it drags a base class explicitly into the programming. A better approach is to define the operator for the *limarith* class:

```
double & limarith::operator[](int n)
{
    return lim_dar::operator[](n);
}
```

Then, the expression

```
chow_masters[year]
```

is interpreted as the following:

```
chow_masters.limarith::operator[](year)
```

And that, in turn, invokes the desired `lim_dar::operator[]()` function.

Summary

Inheritance enables you to adapt programming code to your particular needs by deriving new classes from existing classes. A derived class inherits the `public` and the `protected` data members and methods of the base class. You then can add new data members and methods to the class, and you can use the derived class as a base class for further development. Each derived class requires its own constructors. When a program creates a derived class object it first calls the base class constructor, then the derived class constructor. When a program deletes an object, it first calls the derived class destructor, then the base class destructor.

If a class is meant to be a base class, you normally should use `protected` members instead of `private` members so that derived classes can inherit those members. Also, you normally should make the base class methods virtual functions by declaring them with the keyword `virtual`. This enables objects accessed by pointers or references to be handled on the basis of the object type rather than on the basis of the reference type or pointer type.

You can include a class object as a member of another class. Creating an object of the containing class invokes the constructor for the member class first, then the constructor for the containing class. The destructors are called in the opposite order.

As of version 2.0, C++ allows multiple inheritance, wherein a class can inherit the data members and methods of two or more ancestor classes. If two ancestors share a common ancestor, it's usually best to declare the common ancestor as a virtual base class. That causes the class with multiple lines of descent to get one copy of the shared ancestor's members instead of duplicate copies.

Review Questions

1. What does a derived class inherit from a base class?
2. Suppose the return type for the `darray::operator[]()` function were defined as *darray* instead of *darray* `&`. What effect, if any, would that have?
3. In what order are class constructors and class destructors called when a derived class object is created and deleted?
4. If a derived class doesn't add any data members to the base class, does the derived class require constructors?

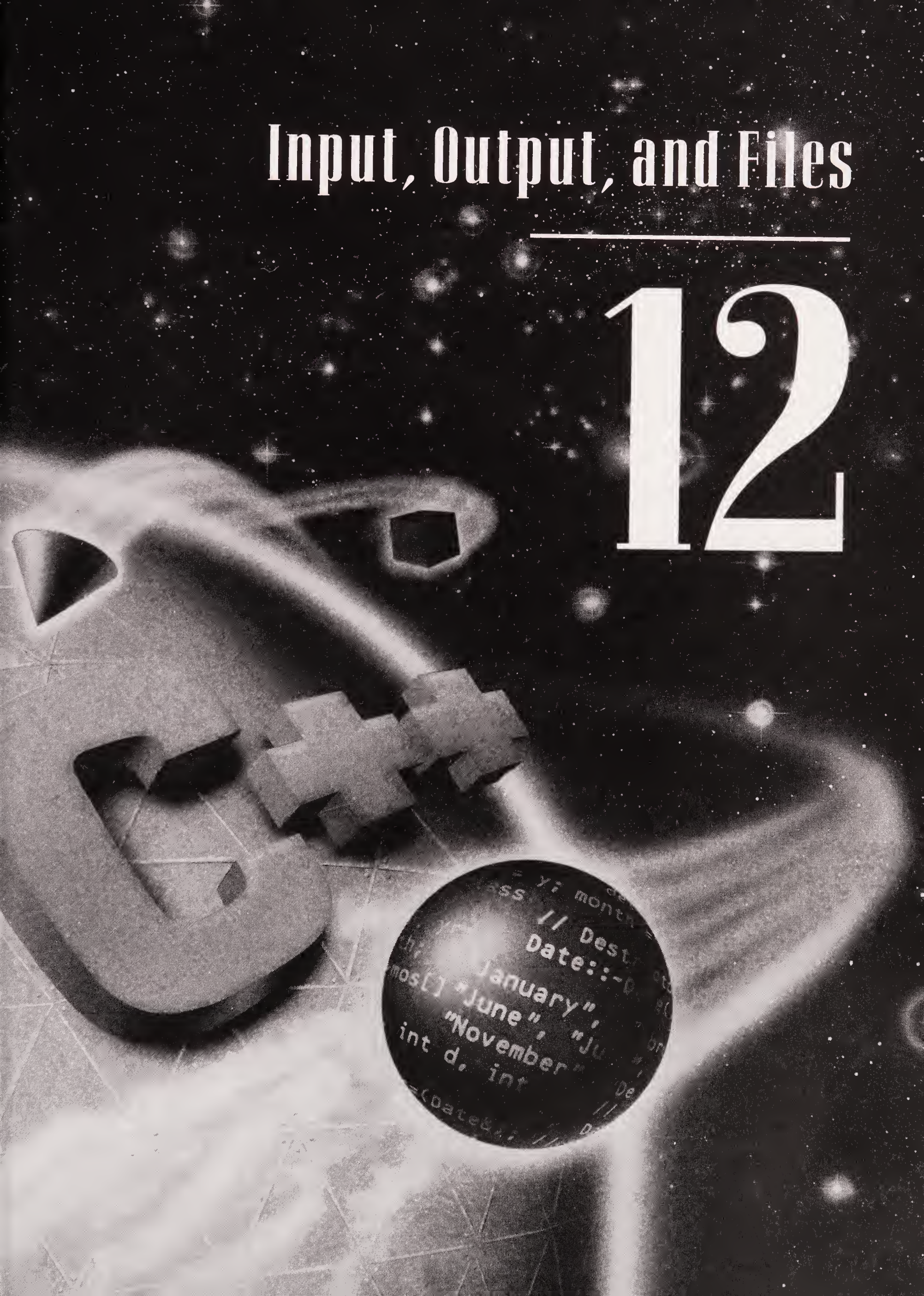
5. Suppose a base class and a derived class both define a method of the same name and a derived class object invokes the method. In what order are the methods called?
6. Why doesn't the *bound_dar* class define an assignment operator?
7. Can you assign an object of a derived class to an object of the base class? Can you assign an object of a base class to an object of the derived class?
8. Suppose we define a function that takes a base class object as an argument. Why can this function also use a derived class object as an argument?
9. Suppose we define a function that takes a reference to a base class object as an argument. Why can this function also use a reference to a derived class object as an argument?
10. Suppose *corporation* is a base class and *department* is a derived class. Also suppose that each class defines a `head()` member function, that `ph` is a pointer to the *corporation* type, and that `ph` is assigned the address of a *department* object. How is `ph->head()` interpreted if the base class defines `head()` as a

    a. regular function.

    b. virtual function.
11. Why is passing an object by reference preferable to passing it by value?
12. If one class contains another class as a member, in what order are the two class constructors called?
13. How does multiple inheritance differ from using member classes as a way of incorporating two classes into a new class?
14. What problem does having a common ancestor in two different lines of inheritance raise? How can you solve that problem?
15. Derive a class from the *person* class that also stores a person's age. Define the << operator so that it prints the age as well as the name.

Input, Output, and Files

12



Discussing C++ input and output (I/O, for short) poses a problem. On the one hand, practically every program uses input and output, and learning how to use them is one of the first tasks facing someone learning a computer language. On the other hand, C++ uses many of its more advanced language features to implement input and output, including classes, derived classes, function overloading, virtual functions, and multiple inheritance. Thus, to really understand C++ I/O, you need to know a lot of C++. To get you started, the early chapters outlined the basic ways for using the *istream* class object *cin* and the *ostream* class object *cout* for input and output. Now we'll take a longer look at C++'s input and output classes, seeing how they are designed and learning how to control the output format. (If you've skipped a few chapters just to learn advanced formatting, you can skim the sections on that topic, noting the techniques and ignoring the explanations.)

The C++ facilities for file input and output are based on the same basic class definitions that `cin` and `cout` are, so this chapter uses the discussion of console I/O (keyboard and screen) as a springboard to investigating file I/O.

An Overview of C++ Input and Output

Most computer languages build input and output into the language itself. For instance, if you look through the lists of keywords for languages like BASIC or Pascal, you'll see that `PRINT` statements, `writeln` statements, and the like are part of the language vocabulary. But neither C nor C++ have built input and output into the language. If you look through the keywords for these languages, you find `for` and `if`, but nothing relating to I/O. C originally left I/O to compiler implementors. One reason for this was to

give implementors the freedom to design I/O functions that best fit the hardware requirements of the target computer. In practice, most implementors based I/O on a set of library functions originally developed for the UNIX environment. ANSI C formalized recognition of this I/O package, called the Standard Input/Output package, by making it a mandatory component of the standard C library. C++ also recognizes this package, so if you're familiar with the family of C functions declared in the `stdio.h` file, you can use them in C++ programs.

C++, however, relies upon a C++ solution rather than a C solution to I/O, and that solution is a set of classes defined in the `iostream.h` and `fstream.h` header files. This class library is not part of the formal language definition (`cin` and *istream* are not keywords); after all, a computer language defines rules for *how* to do things, such as create classes, and doesn't define *what* you should create following those rules. But, just as C implementations come with a standard library of functions, C++ comes with a standard library of classes. At this time, that standard class library consists solely of the classes defined in the `iostream.h` and `fstream.h` header files. (Many implementations provide additional, but nonstandard, class libraries.) This chapter discusses I/O as implemented in C++ 2.0, and you'll get to look at the features of these standard classes soon. But first, let's examine the conceptual framework for C++ I/O.

STREAMS AND BUFFERS

A C++ program views input or output as a *stream* of bytes. On input, a program *extracts* bytes from an input stream, and on output, a program *inserts* bytes into the output stream. For a text-oriented program, each byte can represent a character. More generally, the bytes can form a binary representation of character or numeric data. The bytes in an input stream can come from the keyboard, but they also can come from a storage device, such as a hard disk, or from another program. Similarly, the bytes in an output stream can flow to the screen, to a printer, to a storage device, or to another program. A stream acts as an intermediary between the program and the stream's source or destination. This approach enables a C++ program to treat input from a keyboard in the same manner it treats input from a file; the C++ program merely examines the stream of bytes without needing to know from where the bytes come. Similarly, by using streams, a C++ program can process output in a manner independent of where the bytes are going. Managing input, then, involves two stages:

- Associating a stream with an input to a program
- Connecting the stream to a file

In other words, an input stream needs two connections, one at each end. The file-end connection provides a source for the stream, and the program-end connection

dumps the stream outflow into the the program. (The file-end connection can be to a file, but it also can be to a device, such as a keyboard.) Similarly, managing output involves connecting an output stream to the program and associating some output destination with the stream. It's like plumbing with bytes instead of water. See Figure 12-1.

Usually, input and output can be handled more efficiently by using a *buffer*. A buffer is a block of memory used as an intermediate, temporary storage facility for the transfer information from a device to a program or from a program to a device. Typically, devices like disk drives transfer information in blocks of 512 bytes or more, while programs often process information one byte at a time. The buffer helps match these two disparate rates of information transfer. For instance, suppose a program is supposed to count the number of dollar signs in a hard-disk file. The program could read one character from the file, process it, read the next character from the file, and so on. Reading a file a character at a time from a disk requires a lot of hardware activity and is slow. The buffered approach is to read a large chunk from the disk, store the chunk in the buffer, then read the buffer one character at a time. Because it is much quicker to

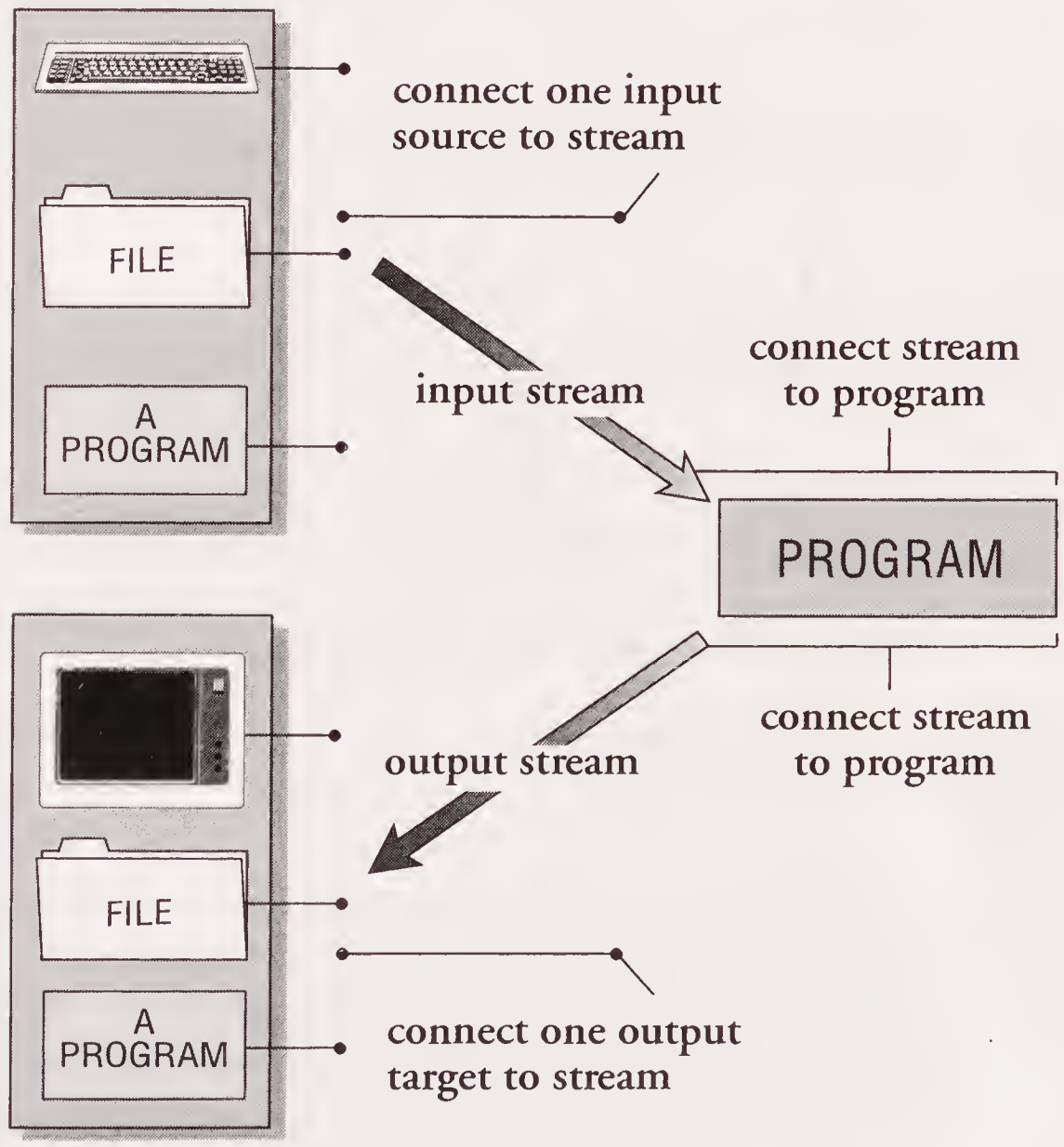
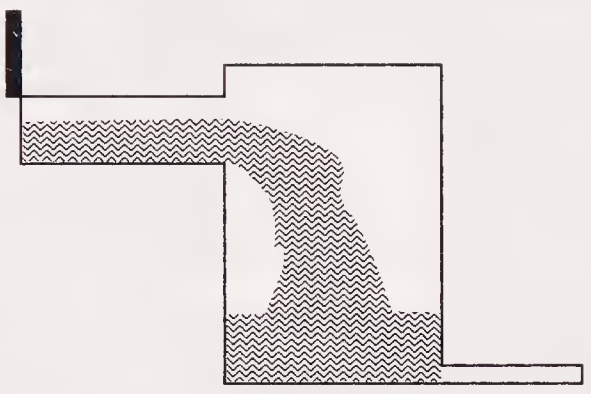
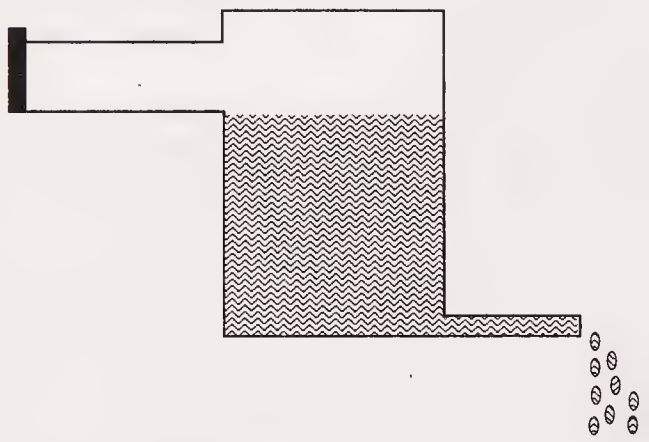


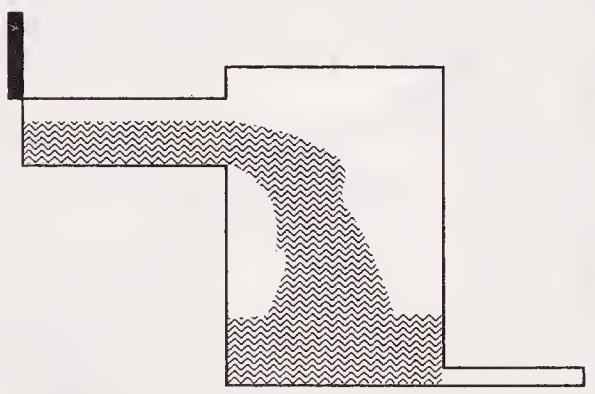
Figure 12-1 *C++ input and output*



Fill stream buffer with block of data



Stream outflow feeds program byte-by-byte



Refill stream buffer with next block of data

Figure 12-2 *A stream with a buffer*

read individual bytes of data from memory than from a hard disk, this approach is much faster as well as easier on the hardware. Of course, once the program reaches the end of the buffer, the program then should read another chunk of data from the disk. The principle is similar to that of a water reservoir that collects megagallons of runoff water during a big storm, then feeds water to your home at a more civilized rate of flow. See Figure 12-2. Similarly, on output, a program can first fill the buffer, then transfer the entire block of data to a hard disk, clearing the buffer for the next batch of output. This

is called *flushing* the buffer. Perhaps you can come up with your own plumbing-based analogy for that process.

Keyboard input provides a character at a time, so in that case a program doesn't need a buffer to help match different data transfer rates. However, buffered keyboard input allows the user to back up and correct input before transmitting it to a program. A C++ program normally flushes the input buffer when you press the Return or Enter key. That's why the examples in this book don't begin processing input until you press one of those keys. For output to the screen, a C++ program normally flushes the output buffer when you transmit a newline character. Depending upon the implementation, a program may flush input on other occasions, too, such as impending input. That is, when a program reaches an input statement, it flushes any output currently in the output buffer. C++ implementations that are consistent with ANSI C should behave in that manner.

STREAMS, BUFFERS, AND THE IOSTREAM.H FILE

The business of managing streams and buffers can get a bit complicated, but the `iostream.h` file defines several classes designed to implement and manage streams and buffers for you. Here are some of those classes (also see Figure 12-3):

- The *streambuf* class provides memory for a buffer along with class methods for filling the buffer, accessing buffer contents, flushing the buffer, and managing the buffer memory.
- The *ios* class represents general properties of a stream, such as whether it's open for reading and whether it is a binary or a text stream, and it includes a pointer member to a *streambuf* object.
- The *ostream* class derives from the *ios* class and provides output methods.
- The *istream* class also derives from the *ios* class and provides input methods.
- The *iostream* class is based on the *istream* and *ostream* class and thus inherits both input and output methods.

To use these facilities, you use objects of the appropriate classes. For example, use an *ostream* object such as `cout` to handle output. Creating such an object opens a stream, automatically creates a buffer, and associates it with the stream. It also makes the class member functions available to you.

The C++ *iostream* class library takes care of many details for you. For instance, including the `iostream.h` file in a program creates four stream objects automatically:

- The *cin* object corresponds to the standard input stream. By default, this stream is associated with the standard input device, typically a keyboard.
- The *cout* object corresponds to the standard output stream. By default, this stream is associated with the standard output device, typically a monitor.

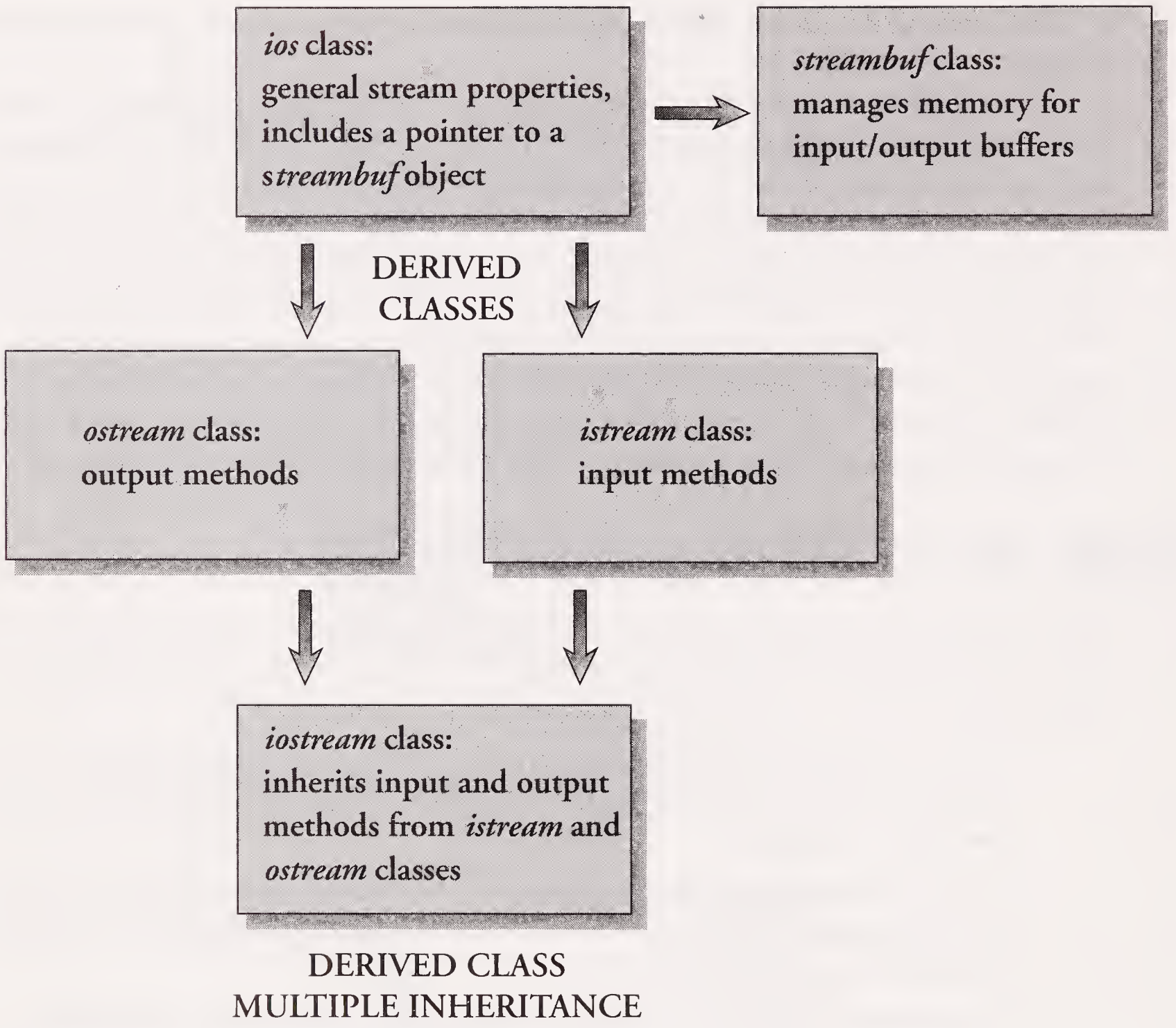


Figure 12-3 *Some classes defined in* `iostream.h`

- The *cerr* object corresponds to the standard error stream, which you can use for displaying error messages. By default, this stream is associated with the standard output device, typically a monitor, and the stream is unbuffered. That means information is sent directly to the screen without waiting for a buffer to fill or for a newline character.
- The *clog* object also corresponds to the standard error stream. By default, this stream is associated with the standard output device, typically a monitor, and the stream is buffered.

What does it mean to say an object represents a stream? Well, for example, when the `iostream.h` file creates a *cout* object for your program, that object will have data members holding information relating to output, such as the field widths to be used in

displaying data, the number of places after the decimal to use, what number base to use for displaying integers, and the address of a *streambuf* object describing the buffer used to handle the output flow. A statement such as

```
cout << "Bjarne free";
```

places the characters from the string `"Bjarne free"` into the buffer managed by `cout` via the pointed-to *streambuf* object. The *ostream* class defines the `operator<<()` function used in this statement, and the *ostream* and *ios* classes also supporting the *cout* data members with a variety of other class methods, such as the ones this chapter discusses later. Furthermore, C++ sees to it that the output from the buffer is directed to the standard output, usually a monitor, provided by the operating system. In short, one end of a stream is connected to your program, the other end is connected to the standard output, and the `cout` object, with the help of a type *streambuf* object, manages the flow of bytes through the stream.

REDIRECTION

The standard input and output streams normally connect to the keyboard and the screen. But many operating systems, including UNIX and MS DOS, support *redirection*, a facility that lets you change the associations for the standard input and the standard output. Suppose, for example, you have an executable DOS C++ program called `counter.exe` that counts the number of characters in its input and reports the result. A sample run might look like this:

```
C>counter
Hello
and goodbye!
CTRL-Z                          ← simulated end of file
Input contained 19 characters.
C>
```

Here input came from the keyboard, and output went to the screen.

With input redirection `(<)` and output redirection `(>)`, you can use the same program to count the number of characters in the `oklahoma` file and to place the results in the `cow_cnt` file:

```
C>counter <oklahoma >cow_cnt
C>
```

The `<oklahoma` part of the command line associates the standard input with the `oklahoma` file, causing *cin* to read input from that file instead of the keyboard. In other words, the operating system changes the connection at the inflow end of the input

stream, while the outflow end remains connected to the program. The `>cow_cnt` part of the command line associates the standard output with the `cow_cnt` file, causing `cout` to send output to that file instead of to the screen. That is, the operating system changes the outflow end connection of the output stream, leaving its inflow end still connected to the program. Both DOS (2.0 and later) and UNIX automatically recognize this redirection syntax. (UNIX and DOS 3.0 and later also permit optional space characters between the redirection operators and the file names.)

The standard output stream, represented by *cout*, is the normal channel for program output. The standard error streams (represented by *cerr* and *clog*) are intended for a program's error messages. By default, all three typically are sent to the monitor. But redirecting the standard output doesn't affect *cerr* or *clog*; thus if you use one of these objects to print an error message, a program will display the error message on the screen even if the regular *cout* output is redirected elsewhere. For instance, consider this code fragment:

```
if (success)
    cout << "Here come the goodies!\n";
else
{
    cerr << "Something horrible has happened.\n";
    exit(1);
}
```

If redirection is not in effect, whichever message is selected is displayed on-screen. If, however, the program output has been redirected to a file, the first message, if selected, would go to the file but the second message, if selected, would go to the screen. By the way, some operating systems permit redirecting the standard error, too. In UNIX, for example, the `2>` operator redirects the standard error.

Actually, the *istream* and *ostream* classes don't necessarily provide for redirection. The Turbo C++ implementation, for example, derives an *istream_withassign* class from the *istream* class for handling redirection, and *cin* is an *istream_withassign* object. Similarly, the *cout*, *cer*, and *clog* objects belong to the *ostream_withassign* class, which is derived from *ostream* in a manner that allows output redirection. Otherwise, these standard objects use the same methods as their respective base classes. For simplicity, we'll refer to *cin* as an *istream* object and *cout* as an *ostream* object.

Output with cout

C++, we've said, considers output to be a stream of bytes. But many kinds of data in a program are organized into larger units than a single byte. An `int` type, for example, may be represented by a two- or four-byte binary value. And a `double` value may be

represented by eight bytes of binary data. But when you send a stream of bytes to a screen, you want each byte to represent a character value. That is, to display the number `-2.34` on the screen, you should send the five characters `-`, `2`, `.`, `3`, and `4` to the screen, and not the internal eight-byte floating-point representation of that value. Therefore, one of the most important tasks facing the *ostream* class is converting numeric types, such as `int` or `float`, into a stream of characters that represents the values in text form. That is, the *ostream* class translates the internal representation of data as binary bit patterns to an output stream of character bytes. (Some day we may have bionic implants to enable us to interpret binary data directly. We leave that development as an exercise for the reader.) To perform these translation tasks, the *ostream* class provides several class methods. We'll look at them now, summarizing methods used throughout the book and describing additional methods that provide a finer control over the appearance of the output.

THE OVERLOADED << OPERATOR

Most often, this book has used `cout` with the <<operator, also called the *insertion* operator:

```
int clients = 22;
cout << clients;
```

In C++, as in C, the default meaning for the `<<` operator is the bitwise left-shift operator (see Appendix E). An expression such as `x<<3` means to take the binary representation of `x` and shift all the bits three units to the left. Obviously, this doesn't have a lot to do with output. But the *ostream* class redefines the `<<` operator through overloading to output for the *ostream* class. In this guise, the `<<` operator is called the insertion operator instead of the left-shift operator. (The left-shift operator earned this role through its visual aspect, which suggests a flow of information to the left.) The insertion operator is overloaded to recognize all the basic C++ types:

- `unsigned char`
- `signed char`
- `short`
- `unsigned short`
- `int`
- `unsigned int`
- `long`
- `unsigned long`

- `float`
- `double`
- `long double`

The list doesn't include `char` because that type is synonymous with either `unsigned char` or `signed char`, depending upon the implementation. The *ostream* class provides a definition for the `operator<<()` function for each of the above types. (Functions incorporating `operator` into the name are used to overload operators, as discussed in Chapter 10.) Thus if you use a statement of the form

```
cout << value;
```

and if `value` is one of the preceding types, a C++ program can match it to an operator function with the corresponding signature. For example, the expression `cout << 88` matches the following method prototype:

```
ostream & operator<<(int);
```

This prototype, recall, indicates that the `operator<<()` function takes one type `int` argument. That's the part that matches the `88` in the previous statement. The prototype also indicates that the function returns a reference to an *ostream* object. That property, as we'll discuss soon, makes it possible to concatenate output, as in the following old rock hit:

```
cout << "I'm feeling sedimental over " << boundary << "\n";
```

If you're a C programmer who has suffered through C's multitudinous `%` type specifiers and the problems that arise when you mismatch a specifier type to a value, using `cout` is almost sinfully easy. (And C++ input, of course, is `cin`fully easy.)

The *ostream* class also defines insertion operator functions for the following pointer types:

- `const signed char *`
- `const unsigned char *`
- `void *`

C++ represents a string, don't forget, by using a pointer to the location of the string. The pointer can take the form of the name of an array of `char` or of an explicit pointer-to-`char` or of a quoted string. Any of these forms is a pointer to `signed char` or to `unsigned char`, depending upon the implementation. Thus all of the following `cout` statements display strings:

```
char name[20] = "Dudly Diddlemore";
char * pn = "Violet D'Amore";
cout << "Hello!";
```

```
cout << name;
cout << pn;
```

The methods use the terminating null character in the string to determine when to stop displaying characters.

C++ 2.0 matches a pointer of any other type with type `void *` and prints a numerical representation of the address. If you want the address of the string, you have to type cast it to another type.

```
int eggs = 12;
char * amount = "dozen";
cout << &eggs;                  // prints address of eggs variable
cout << amount;                 // prints the string "dozen"
cout << (void *) amount;        // prints the address of the amount
                                // variable
```

Note: Not all current C++ implementations have a prototype with the `void *` argument. In that case, you have to type cast a pointer to `unsigned` or, perhaps, `unsigned long` if you wish to print the value.

Output Concatenation

All the incarnations of the insertion operator are defined to return type *ostream* `&`. That is, the prototypes have this form:

```
ostream & operator<<(typename);
```

(Here, *typename* is the type to be displayed.) The *ostream* `&` return type means using this operator returns a reference to an *ostream* object. Which object? The function definitions say that the reference is to the object used to evoke the operator. In other words, an operator function's return value is the same object that evokes the operator. For example, `cout << "potluck"` returns the *cout* object. That's the feature that lets us concatenate output using insertion. For instance, consider the following statement:

```
cout << "We have " << count << " unhatched chickens.\n";
```

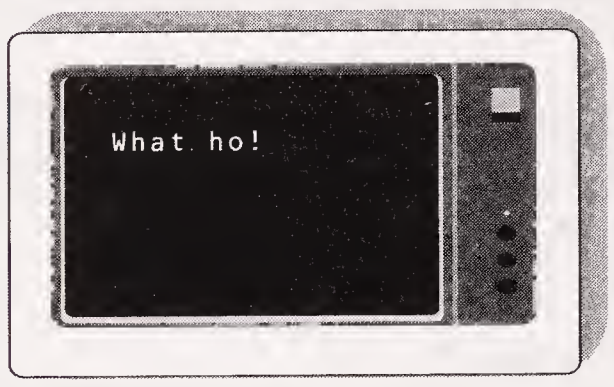
The expression `cout << "We have "` displays the string and returns the *cout* object, reducing the statement to the following:

```
cout << count << " unhatched chickens.\n";
```

Then the expression `cout << count` displays the value of the `count` variable and returns `cout`, which then can handle the final argument in the statement. Also see Figure 12-4. This design technique really is a nice feature, which is why our examples of overloading the << operator in Chapters 10 and 11 shamelessly imitate it.

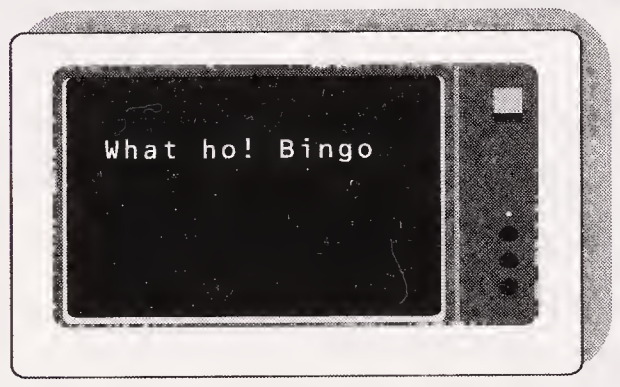
```
char * name = "Bingo"
cout << "What ho! " << name << "!\n";
```

Sends *What ho!* to output buffer and returns cout




```
cout << name << "!\n";
```

Sends *Bingo* to output buffer and returns cout




```
cout << "!\n";
```

Sends *!\n* to output buffer and returns cout (the return value is unused)

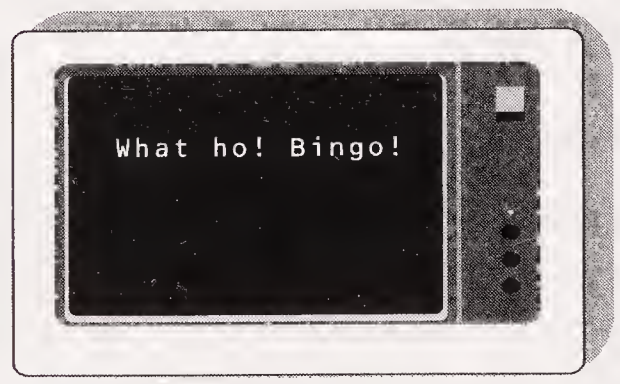


Figure 12-4 *Output concatenation*

THE OTHER OSTREAM METHODS

Besides the various `operator<<()` functions, the *ostream* class provides the `put()` method for displaying characters and the `write()` method for displaying strings.

The `put()` method has the following prototype:

```
ostream & put(char);
```

You invoke it using the usual class method notation:

```
cout.put('W');          // display the W character
```

Here `cout` is the invoking object and `put()` is the class member function. Like the `<<` operator functions, this function returns a reference to the invoking object, so you can concatenate output with it:

```
cout.put('I').put('t'); // displaying It with two put() calls
```

The function call `cout.put('I')` returns `cout`, which then acts as the invoking object for the `put('t')` call.

You can use `put()` with arguments of numeric types other than `char`, such as `int`, and let function prototyping automatically convert the argument to the correct type `char` value. For instance, you can do the following:

```
cout.put(65);          // display the A character
cout.put(66.3);        // display the B character
```

The first statement converts the `int` value *65* to a `char` value, then displays the character having 65 as its ASCII code. Similarly, the second statement converts the type `double` value `66.3` to a type `char66` and displays the corresponding character.

This behavior came in handy prior to Version 2.0 C++, for then the language represented character constants with type `int` values. Thus, a statement such as

```
cout << 'W';
```

would interpret `'W'` as an `int` value, hence display it as the integer 87, the ASCII value for the character. But the statement

```
cout.put('W');
```

worked fine. Because version 2.0 represents `char` constants as type `char`, you now can use either method.

The `write()` method writes an entire string and has the following prototypes:

```
ostream & write(const signed char *, int);
ostream & write(const unsigned char *, int);
```

What about a prototype for ordinary `char`? Well, `char` is implemented either as `signed char` or `unsigned char`, depending on the system, so one or the other of the above prototypes will match it. The first argument to `write()` provides the address of the string to be displayed, and the second argument indicates how many characters to display. Listing 12-1 shows how this works.

LISTING 12-1 write.cpp

```
// write.cpp -- use cout.write()
#include <iostream.h>
#include <string.h>
```

```
int main(void)
{
    char * state1 = "Ohio";
    char * state2 = "Utah";
    char * state3 = "Euphoria";

    int len = strlen(state2);
    for (int i = 1; i <= len; i++)
    {
            cout.write(state2,i);
            cout << "\n";
    }

// concatenate output
    for (i = len; i > 0; i--)
            cout.write(state2,i) << "\n";

// exceed string length
    cout.write(state2, len + 5) << "\n";

    return 0;
}
```

Here is the output:

```
U
Ut
Uta
Utah
Utah
Uta
Ut
U
Utah Euph
```

Note the `cout.write()` call returns the *cout* object. That's because the `write()` method returns a reference to the object that invokes it, and in this case, the *cout* object invokes it. This makes it possible to concatenate output, for `cout.write()` is replaced by its return value, `cout`:

```
cout.write(state2,i) << "\n";
```

Also, note that the `write()` method doesn't stop printing characters automatically when reaching the null character. It simply prints how many characters you tell it to, even if that goes beyond the bounds of a particular string! In this case, we bracketed the string `Utah` with two other strings so that adjacent memory locations would contain

data. We put strings on both sides because compilers differ in the order in which they store data in memory.

The `write()` method can also be used with numeric data. It doesn't translate a number to the correct characters; instead it transmits the bit representation as stored in memory. For instance, a four-byte `long` value such as `1982375632` would be transmitted as four separate bytes. An output device such as a monitor would then try to interpret each byte as if it were ASCII (or whatever) code. So `1982375632` would appear on screen as some four-character combination, most likely gibberish. However, `write 0` does provide a compact, accurate way to store data in a file. We'll return to this possibility later this chapter.

FLUSHING THE OUTPUT BUFFER

Consider what happens as a program uses `cout` to send bytes on to the standard output. Because the *ostream* class buffers output handled by the *cout* object, output isn't sent to its destination immediately. Instead, it accumulates in the buffer until the buffer is full. Then the program flushes the buffer, sending the contents on and clearing the buffer for new data. Typically, a buffer is 512 bytes or an integral multiple thereof. Buffering is a great time saver when the standard output is connected to a file on a hard disk. After all, you don't want a program to access the hard disk 512 times to send 512 bytes. It's much more effective to collect 512 bytes in a buffer, then write them to a hard disk in a single disk operation. For screen output, however, filling the buffer first is less critical. Indeed, it would be inconvenient if you had to reword, say, the message `Press any key to continue` so that it consumed the prerequisite 512 bytes to fill a buffer. Fortunately, the program doesn't necessarily wait until the buffer is full. Sending a newline character to the buffer, for example, normally flushes the buffer. Also, as mentioned before, most implementations flush the buffer when input is pending. That is, suppose you have the following code:

```
cout << "Enter a number: ";
float num;
cin >> num;
```

The fact that the program expects input causes it to display the `cout` message (that is, flush the `Enter a number:` message) immediately, even though the output string lacks a newline. Without this feature, you would have the program waiting for input without having prompted the user with the `cout` message.

If your implementation doesn't flush output when you want it to, you can force flushing by using one of two *manipulators*. The `flush` manipulator flushes the buffer, and the `endl` manipulator flushes the buffer and inserts a newline. You use these manipulators the way you would use a variable name:

```
cout << "Hello, good-looking! " << flush;
cout << "Wait just a moment, please." << endl;
```

Manipulators are, in fact, functions. For instance, you can flush `cout`'s buffer by calling the `flush()` function directly:

```
flush(cout);
```

However, the *ostream* class overloads the `<<` insertion operator in such a way that the statement

```
cout << flush;
```

gets replaced with the `flush(cout);` function call. Thus you can use the more convenient insertion notation to flush with success.

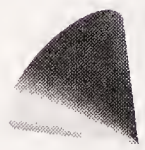
FORMATTING WITH COUT

The *ostream* insertion operators convert values to text form. By default, they format values as follows:

- A type `char` value, if it represents a printable character, is displayed as a character in a field one character wide.
- Numerical integer types are displayed as decimal integers in a field just wide enough to hold the number and, if present, a minus sign.
- Floating-point types are displayed with six places to the right of the decimal, except that trailing zeros aren't displayed. (Note that the number of digits *displayed* has no connection with the precision to which the number is *stored*.) The number is displayed in fixed-point notation or else in E notation (see Chapter 3), depending upon the value of the number. Again, the field is just wide enough to hold the number and, if present, a minus sign.
- Strings are displayed in a field equal in width to the length of the string.

Because each value is displayed in a width equal to its size, you have to provide spaces between values explicitly; otherwise, consecutive values would run together.

Listing 12-2 illustrates the output defaults. It displays a colon (`:`) after each value so you can see the width field used in each case. The program uses the expression `1.0 / 9.0` to generate a nonterminating fraction so you can see how many places get printed.



LISTING 12-2 defaults.cpp

```
// defaults.cpp -- cout default formats
#include <iostream.h>

int main(void)
{
```

```
    cout << "12345678901234567890\n";
    char ch = 'K'; int t = 273;
    cout << ch << ":\n";
    cout << t << ":\n";
    cout << -t <<":\n";

    double f1 = 1.200;
    cout << f1 << ":\n";
    cout << (f1 + 1.0 / 9.0) << ":\n";

    double f2 = 1.67E7;
    cout << f2 << ":\n";
    cout << (f2 + 1.0 / 9.0) << ":\n";
    cout << (f2 + 2.0e5) << ":\n";

    double f3 = 2.3e-4;
    cout << f3 << ":\n";
    cout << f3 / 10 << ":\n";

    return 0;
}
```

Here is the output:

```
12345678901234567890
K:
273:
-273:
1.2:
1.311111:
16700000:
16700000.111111:
1.69e+07:
0.00023:
2.3e-05:
```

Each value fills its field. Note that the trailing zeros of `1.200` are not displayed but that floating-point values without terminating zeros have six places to the right of the decimal displayed. Also, in this particular implementation (Turbo C++), the switch from fixed-point to E notation takes place if the value is bigger than about 1.68E7 or smaller than 1.0E-05.

Changing the Number Base Used for Display

The *ios* class, from which *ostream* derives, stores information describing the format state. For instance, certain bits in one class member determine the number base used, while

another member determines the field width. By using manipulators, you can control the number base used to display integers. By using *ios* member functions, you can control the field width and the number of places displayed to the right of the decimal. Because the *ios* class is a base class for *istream*, you can use its methods with *istream* objects (or descendents), such as *cin*.

Let's see how to set which number base is used in displaying integers. To control whether integers are displayed in base ten, base sixteen, or base eight, you can use the `dec`, `hex`, and `oct` manipulators. For instance, the function call

```
hex(cout);
```

sets the number base format state for the *cout* object to hexadecimal. Once you do this, a program will print values in hexadecimal form until you set the format state to another choice. Note that the manipulators are not member functions, hence they don't have to be invoked by an object.

Although the manipulators really are functions, you normally see them used this way:

```
cout << hex;
```

The *ostream* class overloads the `<<` operator to make this usage equivalent to the function call `hex(cout)`. Listing 12-3 illustrates using these manipulators. It shows the value of an integer and its square in three different number bases. Note that you can use a manipulator separately or as part of a series of insertions.

LISTING 12-3 manip.cpp

```
// manip.cpp -- using format manipulators
#include <iostream.h>

int main(void)
{
    cout << "Enter an integer: ";
    int n;
    cin >> n;

    cout << "n    n*n\n";
    cout << n << "    " << n * n << " (decimal)\n";
// set to hex mode
    cout << hex;
    cout << n << "    ";
    cout << n * n << " (hexadecimal)\n";

// set to octal mode
    cout << oct << n << "    " << n * n << " (octal)\n";

// alternative way to call a manipulator
```

```
    dec(cout);
    cout << n << "    " << n * n << " (decimal)\n";

    return 0;
}
```

Here is some sample output:

```
Enter an integer: 13
n    n*n
13    169 (decimal)
d    a9 (hexadecimal)
15    251 (octal)
13    169 (decimal)
```

Adjusting Field Widths

You probably noticed that the columns in the preceding example don't line up; that's because the numbers have different field widths. You can use the `width` member function to place differently sized numbers in fields having equal widths. The function has these prototypes:

```
int width();
int width(int i);
```

The first form returns the current setting for field width. The second sets the field width to `i` spaces and returns the previous field width value. That allows you to save the previous value in case you wish to restore the width to that value later.

The `width()` method affects only the next item displayed, and the field width reverts to the default value afterwards. For example, consider the following statements:

```
cout << '#';
cout.width(12);
cout << 12 << "#" <<  24 << "#\n";
```

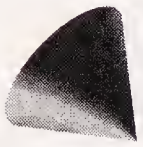
Because `width()` is a member function, you have to use an object (*cout*, in this case) to invoke it. The output statement produces the following display:

```
#          12#24#
```

The `12` is placed in a field 12 characters wide at the right end of the field. This is called *right-justification*. After that, the field width reverts to the default, and the two `#` characters and the `24` are printed in fields equal to their own size.

C++ never truncates data, so if you attempt to print a seven-digit value in a field width of 2, C++ expands the field to fit the data. (Some languages just fill the field with

asterisks if the data doesn't fit. C++ feels that showing all the data is more important than keeping the columns neat; C++ puts substance before form.) Listing 12-4 shows how the `width()` member function works.



LISTING 12-4 width.cpp

```
// width.cpp -- use the width method
#include <iostream.h>

int main(void)
{
    int w = cout.width(30);
    cout << "default field width = " << w << "\n";

    cout.width(5);
    cout << "N";
    cout.width(8);
    cout << "N * N" << "\n";

    for (long i = 1; i <= 100; i *= 10)
    {
        cout.width(5);
        cout << i;
        cout.width(8);
        cout << i * i << "\n";
    }

    return 0;
}
```

Here is the output:

```
        default field width = 0
    N    N * N
    1        1
   10      100
  100    10000
```

First, note that the program applies the field width of 30 to the string displayed by the first `cout` statement but not to the value of `w`. That's because the `width()` method affects only the next single item displays. Also, note that `w` has the value `0`. This is because `cout.width(30)` returns the previous field width, not the one to which it was

just set. The fact that *w* is zero means that zero is the default field width. Because C++ always expands a field to fit the data, this one size fits all. Finally, the program uses `width()` to align column headings and data by using a width of five characters for the first column and a width of eight characters for the second column.

Fill Characters

By default, *cout* fills unused parts of a field with spaces. You can use the `fill()` member function to change that. For instance, the call

```
cout.fill('*');
```

changes the fill character to an asterisk. That can be handy for, say, printing checks so that recipients can't easily add a digit or two. Listing 12-5 illustrates using this member function.

LISTING 12-5 fill.cpp

```
// fill.cpp -- change fill character for fields
#include <iostream.h>

int main(void)
{
    cout.fill('*');
    char * staff[2] = { "Waldo Whipsnade", "Wilmarie Wooper"};
    long bonus[2] = {900, 1350};

    for (int i = 0; i < 2; i++)
    {
            cout << staff[i] << ": $";
            cout.width(7);
            cout << bonus[i] << "\n";
    }

    return 0;
}
```

Here's the output:

```
Waldo Whipsnade: $****900
Wilmarie Wooper: $***1350
```

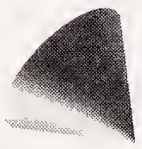
Note that, unlike the field width, the new fill character stays in effect until you change it.

Setting Floating-Point Display Precision

Precision means the maximum number of digits displayed to the right of the decimal point. The default for C++, as you've seen, is six. (Recall, however, that trailing zeros are dropped.) The `precision()` member function lets you select other values. For instance, the statement

```
cout.precision(2);
```

causes *cout* to display no more than two digits to the right of the decimal. Unlike the case with `width()`, but like the case for `fill()`, a new precision setting stays in effect until reset. Listing 12-6 demonstrates precisely this point.



LISTING 12-6 precise.cpp

```
// precise.cpp -- set the precision
#include <iostream.h>

int main(void)
{
    float price1 = 20.40;
    float price2 = 19.0 + 8.0 / 9.0;

    cout << "\"Furry Friends\" is $" << price1 << ".\n";
    cout << "\"Fiery Fiends\" is $" << price2 << ".\n";

    cout.precision(2);
    cout << "\"Furry Friends\" is $" << price1 << ".\n";
    cout << "\"Fiery Fiends\" is $" << price2 << ".\n";

    return 0;
}
```

Here is the output:

```
"Furry Friends" is $20.4.
"Fiery Fiends" is $19.888889.
"Furry Friends" is $20.4.
"Fiery Fiends" is $19.89.
```

Note that setting the precision to two causes the program to round the value displayed for `price2` to `19.89` (i.e., to the nearest cent) instead of merely truncating it to `19.88`. Also note that the output still truncates trailing zeros.

Printing Trailing Zeros

Certain forms of output, such as prices or numbers in columns, look better if trailing zeros are retained. For instance, the output to Listing 12-6 would look better as `$20.40` than as `$20.4`. The *iostream* family of classes doesn't provide a function whose sole purpose is accomplishing that. However, the *ios* class provides a `setf()` (for *set f*lag) function that controls several formatting features, including trailing zeros. The class also defines several constants that can be used as arguments to this function. For instance, the function call

```
cout.setf(ios::showpoint);
```

causes *cout* to display trailing zeros and a trailing decimal point. That is, instead of displaying 2.00 as 2, *cout* will display it as 2.000000, if the default precision of 6 is in effect. Listing 12-7 adds this statement to Listing 12-6.

In case you're wondering about the notation `ios::showpoint`, `showpoint` is an enumerated constant (see Appendix F) defined in the *ios* class definition. Such constants have class scope, which means that you have to use the scope operator (`::`) with the constant name if you use the name outside a member function definition. So, in essence, `ios::showpoint` means a constant defined in the *ios* class.

LISTING 12-7 showzero.cpp

```
// showzero.cpp -- set the precision, show trailing zeros
#include <iostream.h>

int main(void)
{
    float price1 = 20.40;
    float price2 = 19.0 + 8.0 / 9.0;

    cout.setf(ios::showpoint);
    cout << "\"Furry Friends\" is $" << price1 << ".\n";
    cout << "\"Fiery Fiends\" is $" << price2 << ".\n";

    cout.precision(2);
    cout << "\"Furry Friends\" is $" << price1 << ".\n";
    cout << "\"Fiery Fiends\" is $" << price2 << ".\n";

    return 0;
}
```

Now the program produces the following output; note the trailing zeros.

```
"Furry Friends" is $20.400000.
"Fiery Fiends" is $19.888889.
"Furry Friends" is $20.40.
"Fiery Fiends" is $19.89.
```

More About setf()

The `setf()` method controls several other formatting choices, so let's take a closer look at it. The *ios* class has a protected member called `x_flags`. It's a type `long` member in which individual bits (called `flags` in this context) control different formatting aspects such as the number base or whether trailing zeros are displayed. (If you've ever had to set DIP switches to configure computer hardware, bit flags are the programming equivalent.) The `hex`, `dec`, and `oct` manipulators, for example, adjust the three flag bits in `x_flags` that control the number base. The `setf()` function provides another means of adjusting flag bits.

The `setf()` function has two prototypes. The first is this:

```
long setf(long);
```

It's used for setting format information controlled by a single bit. The argument is a `long` value indicating which bit to set. The return value is a type `long` number indicating the former value of that bit. You then can save that value if you later wish to restore the original settings. What value do you pass to `setf()`? If you want to set bit number eleven to 1, then you pass a number having its number eleven bit set to 1. The return value would have its number eleven bit set to the prior setting for that bit. Keeping track of bits sounds (and is) tedious. However, you don't have to do that job, for the *ios* class defines constants representing them. Table 12-1 shows some of these definitions:

TABLE 12-1 formatting constants

Constant Name	Meaning
ios::showbase	use C++ base prefixes (0,0x) on output
ios:: showpoint	show trailing decimal point and zeros
ios::uppercase	use uppercase letters for hex output
ios::showpos	use + before positive numbers

Because these constants are defined within the *ios* class, you must use the scope resolution operator with them. That is, use `ios::uppercase`, not just `uppercase`. Changes remain in effect until overridden. Listing 12-8 illustrates using some of these constants.

LISTING 12-8 setf.cpp

```
// setf.cpp -- use setf() to control formatting
#include <iostream.h>

int main(void)
{
    int temperature = 63;

    cout << "Today's water temperature: ";
    cout.setf(ios::showpos);   // show plus sign
    cout << temperature << "\n";

    cout << "For our programming friends, that's\n";
    cout << hex << temperature << "\n"; // use hex
    cout.setf(ios::uppercase); // use uppercase in hex
    cout.setf(ios::showbase);  // use 0X prefix for hex
    cout << "or\n";
    cout << temperature << "\n";

    return 0;
}
```

Here is the output:

```
Today's water temperature: +63          ← show plus sign
For our programming friends, that's
3f                                      ← use hex format
or
0X3F                    ← use uppercase characters and use 0X prefix
```

Note that the plus sign is used only with the base ten version. C++ treats hexadecimal and octal values as unsigned, hence no sign is needed for them.

The second `setf()` prototype takes two arguments and returns the prior setting.

```
long setf(long, long);
```

This overloaded form of the function is used for format choices controlled by more than one bit. The first argument, as before, is a `long` value containing the desired setting. The second argument is a value that first clears the appropriate bits. For instance, suppose setting bit 3 to 1 means base ten, setting bit 4 to 1 means base 8, and setting bit 5 to 1 means base 16. Suppose output is in base 10 and you want to set it to base 16. Not only do you have to set bit 5 to 1, you also have to set bit 3 to 0—this is called clearing the bit. The clever `hex` manipulator does both tasks automatically. The `setf()` requires a bit more work, for you use the second argument to indicate which bits to clear, then use the first argument to indicate which bit to set. This is not as complicated as it

sounds, for the *ios* class defines constants (shown inTable 12-2) for this purpose. In particular, you should use the constant `ios::basefield` as the second argument and `ios::hex` as the first argument if you're changing bases. That is, the function call

```
cout.setf(ios::hex, ios::basefield);
```

has the same effect as using the `hex` manipulator.

The *ios* class defines three sets of formatting flags that can be handled this way. Each set consists of one constant to be used as the second argument and two to three constants to be used as a first argument. The second argument clears a batch of related bits, then the first argument sets one of those bits to 1. Table 12-2 shows the names of the constants used for the second `setf()` argument, the associated choice of constants for the first argument, and their meanings. For instance, to select left justification, use `ios::adjustfield` for the second argument and `ios::left` as the first argument. Left justification means starting a value at the left end of the field, and right justification means ending a value at the right end of the field. Internal justification means placing any signs or base prefixes at the left of the field and the rest of the number at the right of the field. (Unfortunately, C++ does not provide a self-justification mode. Fixed-point notation means using the 123.4 style for floating-point values regardless of the size of the number, and scientific notation means using the 1.23e04 style regardless of the size of the number.

The `setf()` function is a member function of the *ios* class. Because that's a base class for the *ostream* class, you can invoke the function using the `cout` object. For example, to request left justification, use this call:

TABLE 12-2 Arguments for setf(long, long)

Second Argument	First Argument	Meaning
ios::basefield	*ios::dec*	use base ten
	ios::oct	use base eight
	ios::hex	use base sixteen
ios::floatfield	*ios::fixed*	use fixed-point notation
	ios::scientific	use scientific notation
ios::adjustfield	*ios::left*	use left justification
	ios::right	use right justification
	ios::internal	left-justify sign or base prefix, right-justify value

```
long old = cout.setf(ios::left, ios::adjustfield);
```

To restore the previous setting, do this:

```
cout.setf(old, ios::adjustfield);
```

Listing 12-9 illustrates further examples of using `setf()` with two arguments.

LISTING 12-9 setf2.cpp

```
// setf2.cpp -- use setf() with 2 arguments to control formatting
#include <iostream.h>
#include <math.h>

int main(void)
{
    // use left justification, show the plus sign, show trailing
    // zeros, with three digits to the right of the decimal
    cout.setf(ios::left, ios::adjustfield);
    cout.setf(ios::showpos);
    cout.setf(ios::showpoint);
    cout.precision(3);
    // use e-notation and save old format setting
    long old = cout.setf(ios::scientific, ios::floatfield);

    cout << "Left Justification:\n";
    for (long n = 1; n <= 41; n+= 10)
    {
            cout.width(4);
            cout << n << "|";
            cout.width(12);
            cout << sqrt(n) << "|\n";
    }

    // change to internal justification
    cout.setf(ios::internal, ios::adjustfield);
    // restore default floating-point display style
    cout.setf(old, ios::floatfield);

    cout << "Internal Justification:\n";
    for (n = 1; n <= 41; n+= 10)
    {
            cout.width(4);
            cout << n << "|";
            cout.width(12);
            cout << sqrt(n) << "|\n";
    }

    return 0;
}
```

Here is the output:

```
Left Justification:
+1  |+1.000e+00   |
+11 |+3.317e+00   |
+21 |+4.583e+00   |
+31 |+5.568e+00   |
+41 |+6.403e+00   |
Internal Justification:
+  1|+       1.000|
+ 11|+       3.317|
+ 21|+       4.583|
+ 31|+       5.568|
+ 41|+       6.403|
```

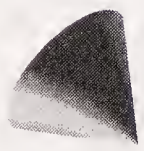
The iomanip.h Header File

Setting some format values, such as the field width, can be awkward using the `iostream.h` tools. To make life easier, C++ supplies additional manipulators in the `iomanip.h` header file. They provide the same services we've discussed, but in a notationally more convenient manner. The two most commonly used are `setprecision()` for setting the precision and `setw()` for setting the field width. Unlike the manipulators discussed previously, these take arguments. The `setprecision()` manipulator takes an integer argument specifying the precision, and the `setw()` manipulator takes an integer argument specifying the field width. Because they are manipulators, they can be concatenated in a `cout` statement. This is particularly convenient when displaying several columns of values. Listing 12-10 illustrates this by changing the field width several times for one output line.

By the way, this program uses a math function, and some C++ systems don't automatically search the math library. For instance, some UNIX systems require that you do the following:

```
$ CC iomanip.C -lm
```

The `-lm` option instructs the linker to search the math library.



LISTING 12-10 iomanip.cpp

```
// iomanip.cpp -- use manipulators from iomanip.h
// some systems require explicitly linking the math library
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
```

```
int main(void)
{
    cout.setf(ios::showpoint);

    cout << setw(6) << "N" << setw(14) << "square root"
            << setw(15) << "fourth root\n";

    double root;
    for (int n = 10; n <=100; n += 10)
    {
       root      = sqrt(n);
            cout << setw(6) << n
                 << setw(12) << setprecision(3) << root
                 << setw(14) << setprecision(4) << sqrt(root)
                 << "\n";
    }

    return 0;
}
```

Here is the output:

```
    N    square root   fourth root
   10          3.162        1.7783
   20          4.472        2.1147
   30          5.477        2.3403
   40          6.325        2.5149
   50          7.071        2.6591
   60          7.746        2.7832
   70          8.367        2.8925
   80          8.944        2.9907
   90          9.487        3.0801
  100         10.000        3.1623
```

Now you can produce neatly aligned columns.

INPUT WITH CIN

Now it's time to turn to input and getting data into a program. The *cin* object represents the standard input as a stream of bytes. Normally, you generate that stream of characters at the keyboard. If you type, say, the character sequence *1992*, the *cin* object extracts those characters from the input stream. You may intend that input to be part of a string, to be an `int` value, to be a `float` value, or to be some other type. Thus ex-

traction also involves type conversion. The *cin* object, guided by the type of variable designated to receive the value, must use its methods to convert that character sequence into the intended type of value.

Typically, you use *cin* as follows:

```
cin >> value_holder;
```

Here `value_holder` identifies the memory location in which to store the input. It can be the name of a variable, a reference, a dereferenced pointer, a member of a structure or of a class, perhaps a pointer. How `cin` interprets the input depends on the data type for `value_holder`. The *istream* class, defined in the `iostream.h` header file, overloads the >> extraction operator to recognize the following basic types:

- `signed char &`
- `unsigned char &`
- `short &`
- `unsigned short &`
- `int &`
- `unsigned int &`
- `long &`
- `unsigned long &`
- `float &`
- `double &`
- `long double &`

A typical operator function has a prototype like the following:

```
istream & operator>>(int &);
```

Both the argument and the return value are references. A reference argument (see Chapter 9) means that a statement such as

```
cin >> staff_size;
```

causes the `operator>>()` function to work with the variable `staff_size` itself rather than with a copy, as would be the case with a regular argument. Because the argument type is a reference, `cin` is able to modify directly the value of a variable used as an argument. The statement above, for example, directly modifies the value of the `staff_size` variable. We'll get to the significance of a reference return value in a moment. But first, let's examine the type conversion aspect of the extraction operator. For arguments of each type in the preceding list of types, the extraction operator converts the character input to the indicated type of value. For instance, suppose `staff_size` is type `int`.

Then the compiler matches the

```
cin >> staff_size;
```

to the following prototype:

```
istream & operator>>(int &);
```

The function corresponding to that prototype then reads the stream of characters being sent to the program, say, the characters `2`, `3`, `1`, `8`, and `4`. For a system using a two-byte `int`, the function then converts these characters to the two-byte binary representation of the integer 23184. If, on the other hand, `staff_size` had been type `double`, *cout* would use the `operator>>(double &)` to convert the same input into the eight-byte floating-point representation of the value 23184.0.

The *istream* class also overloads the >> extraction operator for two pointer types:

- `signed char *`
- `unsigned char *`

For this type of argument, the extraction operator reads the next word from input and places it at the indicated address, adding a null character to make a string. For instance, suppose you have this code:

```
cout << "Enter your first name:\n";
char name[20];
cin >> name;
```

If you respond to the request by typing `Hilary`, then the extraction operator places the characters `Hilary\0` in the `name` array. (As usual, `\0` represents the terminating null character.) The *name* identifier, being the name of a `char` array, acts as the address of the array's first element, making `name` type `char *` (pointer-to-`char`). So which prototype does that match? If your system implements `char` as a signed type, it will use `operator>>(signed char *)`, and if your system implements *char* as an unsigned type, it will use `operator>>(unsigned char *)`.

The fact that each extraction operator returns a reference to the invoking object lets you concatenate input, just as you can concatenate output:

```
char name[20];
float fee;
int group;
cin >> name >> fee >> group;
```

Here, for example, the *cin* object returned by `cin >> name` becomes the object handling `fee`.

HOW CIN >> VIEWS INPUT

The various versions of the extraction operator share a common way of looking at the input stream. They skip over white space (blanks, newlines, and tabs) until encountering a non-white space character. This is true even for the single-character modes (those in which the argument is type `char`, `unsigned char`, or `signed char`), which is not true of C's input functions. See Figure 12-5. In the single character modes, the `>>` operator reads that character and assigns it to the indicated location. In the other modes, the operator reads in one unit of the indicated type. That is, it reads everything from the initial non-white space character up to the first character that doesn't match the destination type. For example, consider the following code:

```
int elevation;
cin >> elevation;
```

Suppose you type the following characters:

```
-123Z
```

The operator will read the `-`, `1`, `2`, and `3` characters, since they are all valid parts of an integer. But the `Z` character isn't valid, so the last character accepted for input is the `3`.

```
char philosophy[20];
int distance;
char initial;

cin >> philosophy >> distance >> initial;
```

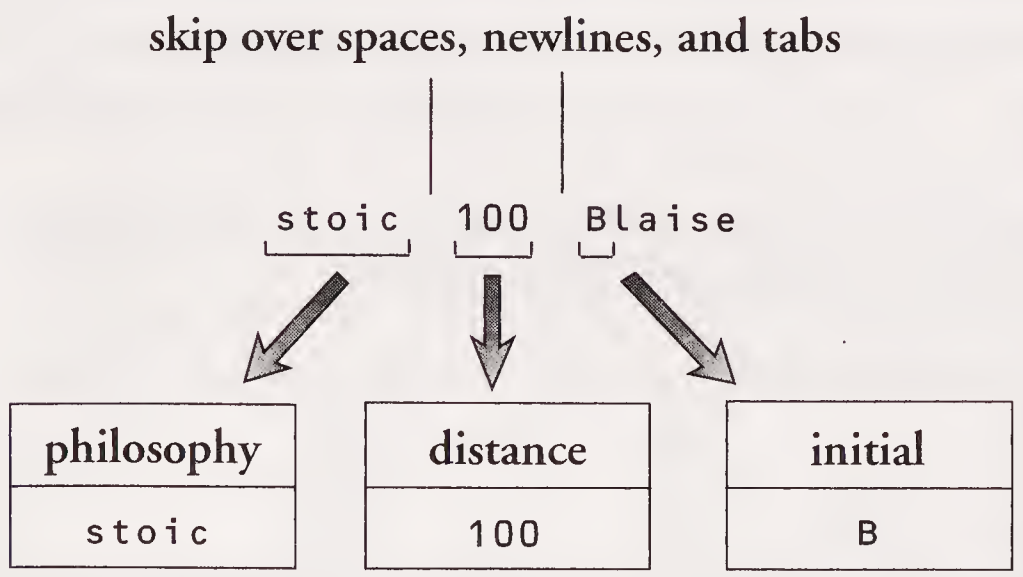


Figure 12-5 cin >> *skips over white space*

The `z` remains in the input stream, and the next `cin` statement will start reading at that point. Meanwhile, the operator converts the character sequence `-123` to an integer value and assigns it to `elevation`.

It can happen that input fails to meet a program's expectation. For example, suppose you entered `Zcar` instead of `-123z`. In that case, the extraction operator leaves the value of `elevation` unchanged and returns the value zero. The zero return value allows a program to check whether input meets the program requirements, as Listing 12-11 shows.

LISTING 12-11 check__it.cpp

```
// check_it.cpp -- validate input
#include <iostream.h>

int main(void)
{
    cout.precision(2);
    cout.setf(ios::showpoint);
    cout << "Enter numbers: ";

    double sum = 0.0;
    double input;
    while (cin >> input)
    {
            sum += input;
    }

    cout << "Last value entered = " << input << "\n";
    cout << "Sum = " << sum << "\n";
    return 0;
}
```

Here's the output when some inappropriate input (*-123Z*) sneaks into the input stream:

```
Enter numbers: 200.0
1.0E1 -50 -123z 60
Last value entered = -123.00
Sum = 37.00
```

Because input is buffered, the second line of keyboard input values didn't get sent to the program until we pressed (RETURN) at the end of the line. But the loop quit process-

ing input at the `Z` character, since it didn't match any of the floating-point formats. The failure of input to match the expected format, in turn, caused the expression `cin>> input` to evaluate to zero, or false, thus terminating the `while` loop.

OTHER ISTREAM CLASS METHODS

Chapters 3, 4, and 5 discuss the `get()` and `getline()` methods. As you may recall, they provide the following additional input capabilities:

- The `get(char &)` and `get(void)` methods provide single-character input that doesn't skip over white space.
- The `get(char *, int, char)` and `getline(char *, int, char)` functions provide string input that reads entire lines by default rather than single words.

Let's look at these two groups of *istream* class member functions.

SINGLE-CHARACTER INPUT

When used with a `char` argument or no argument at all, the `get()` methods fetch the next input character, even if it is a space, tab, or newline character. The `get(char &)` version assigns the input character to its argument, while the `get(void)` version uses the input character as its return value.

Let's try `get(char &)` first. Suppose you have the following loop in a program:

```
int ct = 0;
char ch;
cin.get(ch);
while (ch != '\n')
{
    cout << ch;
    ct++;
    cin.get(ch);
}
cout << ct << '\n';
```

Next, suppose you type the following optimistic input:

```
I C++ clearly.(RETURN)
```

Pressing the (RETURN) key sends this input line to the program. The program fragment will first read the `I` character, display it with *cout*, and increment `ct` to `1`. Next, it will read the space character following the `I`, display it, and increment `ct` to `2`. This contin-

ues until the program processes the (RETURN) key as a newline character and terminates the loop. The main point here is that, by using `get(ch)`, the code reads, displays, and counts the spaces as well as the printing characters. Suppose, instead, that the program had tried to use `>>`:

```
int ct = 0;
char ch;
cin >> ch;
while (ch != '\n')        // FAILS
{
    cout << ch;
    ct++;
    cin >> ch;
}
cout << ct << '\n';
```

First, the code would skip the spaces, thus not counting them and compressing the corresponding output to this:

```
IC++clearly.
```

Worse, the loop would never terminate! Because the extraction operator skips newlines, the code would never assign the newline character to `ch`, so the `while` loop test would never terminate the loop.

The `get(char &)` member function normally returns a reference to the *istream* object used to invoke it. This means you can concatenate other extractions following `get(char)`:

```
char c1, c2, c3;
cin.get(c1).get(c2) >> c3;
```

First, `cin.get(c1)` assigns the first input character to *c1* and returns the invoking object, which is `cin`. This reduces the code to `cin.get(c2) >> c3`, which assigns the second input character to `c2`. The function call returns *cin*, reducing the code to `cin >> c3`. This, in turn, assigns the next non-white space character to `c3`. Note that `c1` and `c2` could wind up being assigned white space, but `c3` can't.

If `cin.get(char &)` encounters the end of a file, either real or simulated from the keyboard ((CTRL)-(Z) for DOS, (CTRL)-(D) at the beginning of a line for UNIX), it does not assign a value to its argument. This is quite right, for if the program has reached the end of the file, there is no value to be assigned. Furthermore, instead of returning the invoking object (`cin`), `cin.get(char &)` returns zero upon detecting the end of a file. Hence you can use the following loop to read characters up to the end of a file:

```
char ch;
while (cin.get(ch))
{
    // process input
}
```

As long as there's valid input, the return value for `cin.get(ch)` is `cin`, which is nonzero, so the loop continues. Upon reaching end-of-file, the return value becomes zero, terminating the loop.

The `get(void)` member function also reads white space, but it uses its return value to communicate input to a program. So you would use it this way:

```
int ct = 0;
char ch;
ch = cin.get();           // use return value
while (ch != '\n')
{
    cout << ch;
    ct++;
    ch = cin.get();
}
cout << ct << '\n';
```

Some C++ implementation functions don't provide this member function.

The `get(void)` member function returns type `int`. This makes the following invalid:

```
char c1, c2, c3;
cin.get().get() >> c3;  // not valid
```

Here `cin.get()` returns a type `int` value. Because that return value is not a class object, you can't apply the membership operator to it. Thus you get a syntax error. However, you can use `get()` at the *end* of an extraction sequence:

```
char c1;
cin.get(c1).get();      // valid
```

The fact that `get(void)` returns type `int` means you can't follow it with an extraction operator. But, because `cin.get(c1)` returns `cin`, it makes it a suitable prefix to `get()`. This particular code would read the first input character, assign it to `c1`, then read the second input character and discard it.

Upon reaching the end-of-file, real or simulated, `cin.get(void)` returns the value `EOF`, which is a symbolic constant defined in the `iostream.h` header file. This design feature allows the following construction for reading input:

```
int ch;
while ((ch = cin.get()) != EOF)
```

```
{
    // process input
}
```

You should use type `int` for `ch` instead of type `char` here because the value `EOF` may not be expressed as a `char` type.

Chapter 5 describes these functions in a bit more detail, and Table 12-3 summarizes the features of the single-character input functions.

Table 12-3 `cin.get(ch)` versus `cin.get()`

PROPERTY	`cin.get(ch)`	`ch = cin.get()`
method for conveying input character	assign to argument `ch`	use function return value to assign to `ch`
function return value for character input	reference to a class *istream* object	code for character as type `int` value
function return value at end-of-file	`0`	`EOF`

Which Form of Single-Character Input?

Given the choice of `>>`, `get(char &)`, and `get(void)`, which should you use? First, decide whether you want input to skip over white space or not. If skipping white space is more convenient, use the extraction operator `>>`. For instance, skipping white space is convenient for offering menu choices:

```
cout << "a. annoy client        b. bill client\n"
     << "c. calm client         d. deceive client\n"
     << "q.\n";
cout << "Enter a, b, c, d, or q: ";
char ch;
cin >> ch;
while (ch != 'q')
{
    switch(ch)
    {
            ...
    }
    cout << "Enter a, b, c, d, or q: ";
    cin >> ch;
}
```

To enter, say, a `b` response, you type (B) (RETURN), generating the two-character response of `b\n`. If you used either form of `get()`, you would have to add code to process that `\n` character each loop cycle, but the extraction operator conveniently skips it. (If you've programmed in C, you've probably encountered the situation in which the newline appears to the program as an invalid response. It's an easy problem to fix, but it is a nuisance.)

If you want a program to examine every character, use one of the `get()` methods. For instance, a word-counting program could use white space to determine when a word came to an end. Of the two `get()` methods, the `get(char &)` method has the classier interface. Also, it's more widely supported. The main advantage of the `get(void)` method is that it closely resembles the standard C `getchar()` function, letting you convert a C to a C++ program by including `iostream.h` instead of `stdio.h`, globally replacing `getchar()` with `cin.get()`, and globally replacing C's `putchar(ch)` with `cout.put(ch)`.

String Input: getline(), get(), and ignore()

Next, let's review the string input member functions, which Chapter 4 discusses more fully. The `getline()` member function and the third version of `get()` both read strings, and both have the same function signature:

```
istream & get(char *, int, char = '\n');
istream & getline(char *, int, char = '\n');
```

The first argument, recall, is the address of the location to place the input string. The second argument is one greater than the maximum number of characters to be read. (The plus one leaves space for the terminating null character used in storing the input as a string.) If you omit the third argument, each function reads up to the maximum characters or until a newline character, which ever comes first. For example, the code

```
char line[50];
cin.get(line, 50);
```

reads character input into the character array `line`. The `cin.get()` function quits reading input into the array after encountering forty-nine characters or, by default, after encountering a newline character, whichever comes first. The difference between `get()` and `getline()` is that `get()` leaves the newline character in the input stream, making it the first character seen by the next input operation, while `getline()` extracts and discards the newline character from the input stream. Not all C++ implementations include `getline()`.

Chapter 4 illustrated using the default form for these two member functions. Now let's look at the final argument, which modifies the function's default behavior. The third argument, which has a default value of `'\n'`, is the termination character. En-

countering the termination character causes input to cease even if the maximum number of characters hasn't been reached. So, by default, both methods quit reading input if they reach the end of a line before reading the allotted number of characters. The difference between `get()` and `getline()` is that `get()` leaves the termination character in the input queue, while `getline()` doesn't. The first Turbo C++ implementaiton of `getline()` included the termination character in the destination string, but most implementations, include Borland C++ and newer releases of Turbo C++, simply read and discard the termination character.

Listing 12-12 demonstrates how `getline()` and `get()` work. It also introduces the `ignore()` member function. It takes two arguments: a number specifying a maximum number of characters to read and a character that acts as a terminating character for input. For example, the function call

```
cin.ignore(80, '\n');
```

reads and discards the next 80 characters or up through the first newline character, whichever comes first. The prototype provides defaults of `1` and `EOF` for the two arguments, and the function return type is `istream &`:

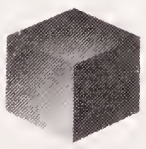
```
istream & ignore(int = 1, int = EOF);
```

The function returns the invoking object. This lets you concatenate function calls as in the following:

```
cin.ignore(80, '\n').ignore(80, '\n');
```

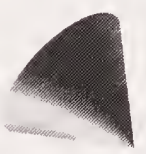
Here the first `ignore()` method reads and discards one line, and the second call reads and discards the second line. Together the two functions read through two lines.

Now check out Listing 12-12.



COMPATIBILTY NOTE

Early releases of Turbo C++ implemented `ignore()` so that it read up to, but not through, the terminating character. Thus, with those versions, you have to use `cin.ignore(80, '\n').get(ch)` to read and discard an entire line.



LISTING 12-12 get__fun.cpp

```
// get_fun.cpp -- using get() and getline()
// note: some C++ inplementations don't support getline ()
#include <iostream.h>

const int Limit = 80;
int main(void)
```

```
{
    char input[Limit];

    cout << "Enter a string for getline() processing:\n";
    cin.getline(input, Limit, '#');
    cout << "Here is your input:\n";
    cout << input << "\nDone with phase 1\n";

    char ch;
    cin.get(ch);
    cout << "The next input character is " << ch << "\n";

    if (ch != '\n')
           cin.ignore(Limit, '\n');    // discard rest of line

    cout << "Enter a string for get() processing:\n";
    cin.get(input, Limit, '#');
    cout << "Here is your input:\n";
    cout << input << "\nDone with phase 2\n";

    cin.get(ch);
    cout << "The next input character is " << ch << "\n";

    return 0;
}
```

Here is a sample program run:

```
Enter a string for getline() processing:
Please pass
me a #3 melon!
Here is your input:
Please pass
me a
Done with phase 1
The next input character is 3
Enter a string for get() processing:
I still
want my #3 melon!
Here is your input:
I still
want my
Done with phase 2
The next input character is #
```

Note that the `getline()` function discards the `#` termination character in the input, while the `get()` function doesn't.

OTHER ISTREAM METHODS

Other *istream* methods include `read()`, `peek()`, `gcount()`, and `putback()`. The `read()` function reads a given number of bytes, storing them in the specified location. For instance, the statements

```
char gross[144];
cin.read(gross, 144);
```

reads 144 characters from the standard input and places them in the `gross` array. Unlike `getline()` and `get()`, `read()` does not append a null character to input, so it doesn't convert input to string form. The `read()` method is not primarily intended for keyboard input. Instead, it most often is used in conjunction with the *ostream* `write()` function for file input and output. The method's return type is `istream &`, so it can be concatenated as follows:

```
char gross[144];
char score[20];
cin.read(gross, 144).read(score, 20);
```

The `peek()` function returns the next character from input without extracting from the input string. That is, it lets you peek at the next character. Suppose you wanted to read input up to the first newline or period, whichever comes first. Then you can use `peek()` to peek at the next character in the input stream in order to judge whether to continue or not:

```
char great_input[80];
char ch;
int i = 0;
while ((ch = cin.peek()) != '.' && ch != '\n')
    cin.get(great_input[i++]);
great_input [i] = '\0';
```

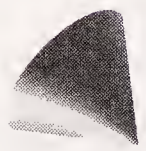
The call to `cin.peek()` peeks at the next input character and assigns its value to `ch`. Then the `while` loop test condition checks that `ch` is neither a period nor a newline. If this is the case, the loop then reads the character into the array, then updates the array index. When the loop terminates, the period or newline character remains in the input stream, positioned to be the first character read by the next input operation.Then the code appends a null character to the array, making it a string.

The `gcount()` method returns the number of characters read by the last unformatted extraction method. That means characters read by a `get()`, `getline()`, or `read()` method but not by the extraction operator `(>>)`, which formats input to fit particular data types. For instance, suppose you've just used the `cin.get(myarray, 80)` to read a line into the `myarray` array and want to know how many character were read. You could use the `strlen()` function to count the characters in the array, but it

would be quicker to use `cin.gcount()` to report how many characters were just read from the input stream.

The `putback()` function inserts a character back in the input string. The inserted character then becomes the first character read by the next input statement. The `putback()` method takes one `char` argument, which is the character to be inserted, and it returns type `istream &`, which allows the call to be concatenated with other `istream` methods. Using `peek()` is like using `get()` to read a character, then using `putback()` to place the character back in the input stream. However, `putback()` gives you the option of putting back a character different from the one just read.

Listing 12-13 uses two approaches to read and echo input up to, but not including, a `#` character. The first approach reads through the `#` character, then uses `putback()` to insert the character back in the input. The second approach uses `peek()` to look ahead before reading input.



LISTING 12-13 istream.cpp

```
// istream.cpp -- some istream methods
#include <iostream.h>
#include <stdlib.h>                    // for exit()
int main(void)
{

// read and echo input up to a # character
   char ch;
   int found_it = 0;

   while(cin.get(ch))                  // terminates on EOF
   {
       if (ch != '#')
          cout << ch;
       else
       {
          cin.putback(ch);             // reinsert character
                found_it = 1;
                break;
       }
   }

   if (found_it)
   {
       cin.get(ch);
       cout << '\n' << ch << " is next input character.\n";
   }
```

```
    else
    {
        cout << "End of file reached.\n";
        exit(0);
    }

    while(cin.peek() != '#')              // look ahead
    {
        cin.get(ch);
        cout << ch;
    }
    cin.get(ch);
    cout << '\n' << ch << " is next input character.\n";

    return 0;
}
```

Here is a sample run:

```
I used a #3 pencil when I should have used a #2.
I used a
# is next input character.
3 pencil when I should have used a
# is next input character.
```

Program Notes

Let's look more closely at some of the code. The first approach uses a `while` loop to read input. The expression `(cin.get(ch))` returns `0` on reaching the end-of-file condition, so simulating end-of-file from the keyboard terminates the loop. If the `#` character shows up first, the program puts the character back in the input stream, sets `found_it` to `1` to record its success, and uses a `break` statement to terminate the loop.

```
while(cin.get(ch))                    // terminates on EOF
{
    if (ch != '#')
            cout << ch;
    else
    {
            cin.putback(ch);          // reinsert character
            found_it = 1;
            break;
    }
}
```

The second approach is simpler in appearance:

```
while(cin.peek() != '#')        // look ahead
{
    cin.get(ch);
    cout << ch;
}
```

The program peeks at the next character. If it is not the # character, the program reads the next character, echoes it, and peeks at the next character. This continues until the terminating character shows up.

File Input and Output

Most computer programs work with files. Word processors create document files. Database programs create and search files of information. Compilers read source code files and generate executable files. A file itself is a bunch of bytes stored on some device, perhaps magnetic tape, perhaps an optical disk, floppy disk, or hard disk. Typically, the operating system manages files, keeping track of their locations, their sizes, when they were created, and so on. Unless you're programming on the operating system level, you normally don't have to worry about those things. What you do need is a way to connect a program to a file, a way to have a program read the contents of a file, a way to have a program create and write to files. Redirection (as discussed beginning on page 583) can provide some file support, but it is more limited than explicit file I/O from within a program. Also, redirection comes from the operating system, not from C++, so it may not be available on all systems. We'll look now at how C++ deals with explicit file I/O from within a program.

The C++ I/O class package handles file input and output much as it handles standard input and output. To write to a file, you create a stream object and use the *ostream* methods, such as the `<<` insertion operator or `write()`. To read a file, you create a stream object and use the *istream* methods, such as the `>>` extraction operator or `get()`. Files require more management than the standard input and output, however. For instance, you have to associate a newly opened file with a stream. You can open a file in read-only mode, write-only mode, or read-and-write mode. If you write to a file, you may wish to create a new file, replace an old file, or add to an old file. Or you may wish to move back and forth through a file. To help handle these tasks, C++ defines several new classes in the `fstream.h` file, including an *ifstream* class for file input, an *ofstream* class for file output, and an *fstream* class for simultaneous file I/O. These classes are derived from the classes in the `iostream.h` file, so objects of these new classes will be able to use the methods you've already learned.

SIMPLE FILE I/O

Suppose you want a program to write to a file. You need to do the following:

- Create an *ofstream* object to manage the output stream.
- Associate that object with a particular file.

To accomplish this, begin by including the `fstream.h` header file. Including this file automatically includes the `iostream.h` file, so you don't have to include that file explicitly. Then declare an *ofstream* object and initialize it using the name of the file to be opened. For example, to open the `cookies` file for output, do the following:

```
ofstream fout("cookies"); // create fout object
```

We named the object *fout* but you can use any valid C++ name, such as `outfile`, `cgate`, or `didi`. We chose the name *fout* for the *ofstream* object as a reminder that you can use it much as you use *cout*. For instance, if you want to put the words `"Dull Data"` into the file, you can do the following:

```
fout << "Dull Data";
```

Indeed, because *ostream* is a base class for the *ofstream* class, you can use all the *ostream* methods, including the various insertion operator definitions and the formatting methods and manipulators. The *ofstream* class uses buffered output, so the program allocates space for an output buffer when it creates an *ofstream* object like *fout*. If you create two *ofstream* objects, the program creates two buffers, one for each object. An *ofstream* object like `fout` collects output byte by byte from the program, then, when the buffer is filled, transfers the buffer contents en masse to the destination file. Because disk drives are designed to transfer data in larger chunks, not byte-by-byte, the buffered approach greatly speeds up the transfer rate of data from a program to a file.

Opening a file for output this way creates a new file if there is no file of that name. If a file by that name exists prior to opening it for output, the act of opening it truncates it so that output starts with a clean file. Later you'll see how to open an existing file and retain its contents.

The requirements for reading a file are much like those for writing to a file:

- Create an *ifstream* object to manage the input stream.
- Associate that object with a particular file.

The steps for doing so are similar to those for writing to a file. First, of course, include the `fstream.h` header file. Then declare an *ifstream* object, initializing it with the file name. For example, to read the `cookies` file, you can do the following:

```
ifstream fin("cookies");        // open cookies for reading
```

We've used the name *fin* for the *ifstream* object to remind you that you can use *fin* much like *cin*. For instance, you can do the following:

```
char ch;
fin >> ch;                  // read a character from the file
char buf[80];
fin >> buf;                 // read a word from the file
fin.getline(buf, 80);       // read a line from the file
```

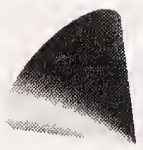
Input, like output, is buffered, so creating an *ifstream* object like `fin` creates an input buffer which the *fin* object manages. As with output, buffering moves data much faster than byte-by-byte transfer.

The connections with a file are closed automatically when the input and output stream objects expire, for example, when the program terminates. Also, you can close a connection with a file explicitly by using the `close()` method:

```
fout.close();     // close output connection to file
fin.close();      // close input connection to file
```

Closing such a connection does not eliminate the stream; it just disconnects it from the file. However, the stream management apparatus remains in place. For instance, the `fin` object still exists along with the input buffer it manages. As you'll see later, you can reconnect the stream to the same file or to another file.

Meanwhile, let's look at a short example. The program in Listing 12-14 asks you for a file name. It creates a file having that name, writes some information to it, and closes the file. Closing the file flushes the buffer, guaranteeing that the file is updated. Then the program opens the same file for reading and displays its contents. Note that the program uses *fin* and *fout* in the same manner as *cin* and *cout*.



LISTING 12-14 file.cpp

```
// file.cpp -- save to a file
#include <fstream.h>

int main(void)
{
    char filename[20];

    cout << "Enter name for new file: ";
    cin >> filename;

// create output stream object for new file and call it fout
    ofstream fout(filename);
```

```
    fout << "For your eyes only!\n";          // write to file
    cout << "Enter your secret number: ";     // write to screen
    float secret;
    cin >> secret;
    fout << "Your secret number is " << secret << "\n";
    fout.close();                             // close file

// create input stream object for new file and call it fin
    ifstream fin(filename);
    cout << "Here are the contents of " << filename << ":\n";
    char ch;
    while (fin.get(ch))                       // read character from file and
           cout << ch;                        // write it to screen
    cout << "Done\n";

    return 0;
}
```

Here is a sample run:

```
Enter name for new file: pythag
Enter your secret number: 3.14159
Here are the contents of pythag:
For your eyes only!
Your secret number is 3.14159
Done
```

OPENING MULTIPLE FILES

You may require that a program open more than one file. The strategy for opening multiple files depends upon how they will be used. If you need two files open simultaneously, then you need to create a separate stream for each file. For instance, a program that collates two sorted files into a third file would create two *ifstream* objects for the two input files and an *ofstream* object for the output file. The number of files you can open simultaneously depends on the operating system, but it typically is of the order of twenty.

However, you may plan to process a group of files sequentially. For instance, you may wish to count how many times a name appears in a set of ten files. Then you can open a single stream and associate it with each file in turn. This conserves computer resources more effectively than opening a separate stream for each file. To use this approach, declare a stream object without initializing it, then use a second statement to associate the stream with a file. For instance, this is how you could handle reading two files in succession:

```
ifstream fin;              // create stream
fin.open("fat.dat");       // associate stream with fat.dat file
...                        // do stuff
fin.close();               // terminate association with fat.dat
fin.open("rat.dat");       // associate stream with rat.dat file
...
fin.close();
```

We'll look at an example shortly, but first, let's examine a technique for feeding a list of files to a program in a manner that allows the program to use a loop to process them.

Command-Line Processing

File-processing programs often use command-line arguments to identify files. Command-line arguments are arguments that appear on the command line when you type a command. For example, to count the number of words in some files on a UNIX system, you would type this command at the UNIX prompt:

```
wc report1 report2 report3
```

Here `wc` is the program name, and `report1`, `report2`, and `report3` are file names passed to the program as command-line arguments.

C++ has a mechanism for letting a program access command-line arguments. Use the following alternative function heading for `main()`:

```
int main(int argc, char *argv[])
```

The `argc` argument represents the number of arguments on the command line. The count includes the command name itself. The *argv* variable is a pointer to a pointer to a `char`. This sounds a bit abstract, but you can treat *argv* as if it were an array of pointers to the command-line arguments, with `argv[0]` being a pointer to the first character of a string holding the command name, `argv[1]` being a pointer to the first character of a string holding the first command-line argument, and so on. That is, `argv[0]` is the first string from the command line, and so on. For instance, suppose you have the following command line:

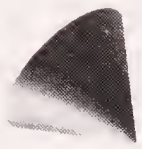
```
wc report1 report2 report3
```

Then `argc` would be `4`, `argv[0]` would be `"wc"`, `argv[1]` would be "`report1`", and so on. The following loop would print each command-line argument on a separate line:

```
for (int i = 1; i < argc; i++)
    cout << argv[i] << "\n";
```

Starting with `i = 1` just prints the command-line arguments; starting with `i = 0` would also print the command name.

Listing 12-15 combines the command-line technique with file stream techniques to count characters in those files listed on the command line.



LISTING 12-15 count.cpp

```
// count.cpp -- count characters in a list of files
#include <fstream.h>
#include <stdlib.h>                    // for exit()

int main(int argc, char *argv[])
{
    if (argc == 1)                     // quit if no arguments
    {
        cerr << "Usage: " << argv[0] << " filename[s]\n";
        exit(1);
    }

    ifstream fin;                      // open stream
    long count;
    long total = 0;
    char ch;

    for (int file = 1; file < argc; file++)
    {
        fin.open(argv[file]);  // connect stream to argv[file]
        count = 0;
        while (fin.get(ch))
            count++;
        cout << count << " characters in " << argv[file] << "\n";
        total += count;
        fin.close();                   // disconnect file
    }
    cout << total << " characters in all files\n";

    return 0;
}
```

On a DOS system, for example, you could compile Listing 12-15 to an executable file called `count.exe`. Then a sample run could look like this:

```
C>count
Usage: c:\count.exe filename[s]
C>count paris rome
3580 characters in paris
```

```
4886 characters in rome
8466 characters in all files
C>
```

If you are using Turbo C++'s integrated environment, you can supply command-line arguments via the Arguments selection in the Run menu.

Note that the program uses `cerr` for the error message. A minor point is that the message uses `argv[0]` instead of `count.exe`:

```
cerr << "Usage: " << argv[0] << " filename[s]\n";
```

This way, if you change the name of the executable file, the program will automatically use the new name.

Suppose you pass a bogus file name to the `count` program. Then the input statement `fin.get(ch)` will fail, terminating the `while` loop immediately, and the program will report *0* characters. But you can modify the program to test whether it succeeded in linking the stream to a file. That's one of the matters we'll take up next.

STREAM CHECKING

The C++ file stream classes inherit a stream-state member from the *ios* class. This member stores information reflecting the stream status: all is well, end-of-file has been reached, I/O operation failed, and so on. If all is well, the stream state is zero (no news is good news). The various other states are recorded by setting particular bits to 1. The file stream classes also inherit methods that report about the stream state. Table 12-4 summarizes them.

Table 12-4 Stream-state Methods

METHOD	RETURNS
`eof()`	nonzero on end-of-file
`fail()`	nonzero if last I/O operation failed or if invalid operation attempted or if there's been an unrecoverable error
`bad()`	nonzero if invalid operation attempted or if there's been an unrecoverable error
`good()`	nonzero if all stream state bits are zero
`rdstate()`	the stream state
`clear(int n = 0)`	returns nothing, but sets stream state to `n`; the default value of `n` is `0`

When an input function detects the end-of-file, for example, it sets the end-of-file bit to 1. If you attempt an invalid operation, such as reading past the end of a file or connecting a stream to a nonexistent file, the fail bit is set to 1. You can monitor these conditions with the stream-state methods. For instance, you can modify the program in Listing 12-15 so that it reports bogus file names, then skips to the next file. Just add a call to `fin.fail()` to the `for` loop as follows:

```
for (int file = 1; file < argc; file++)
{
    fin.open(argv[file]);

// Add this
    if (fin.fail())
    {
            cerr << "Couldn't open file " << argv[file] << "\n";
            continue;
    }
// End of addition

    count = 0;
    while (fin.get(ch))
            count++;
    cout << count << " characters in " << argv[file] << "\n";
    total += count;
    fin.close();                    // disconnect file
}
```

The `fin.fail()` call returns nonzero, or true, if the `fin.open()` call fails. In that case, the program warns you of its problem, and the `continue` statement causes the program to skip the rest of the `for` loop cycle and to start with the next cycle.

Constructions such as

```
while (fin.get(ch))
```

work because the `get()` family of member functions, as well as `getline()` and the extraction operators, return a value of 0 if `fail()` is true. Thus a loop of this sort may terminate for a variety of reasons, including reaching the end of a file, a hardware failure, or attempting to use a stream that hasn't been associated with a file. You can use the `eof()` method after such a loop to verify that it terminated because of reaching end-of-file rather than because of some abnormal condition.

FILE MODES

The *file mode* describes how a file is to be used: read it, write to it, append it, and so on. When you associate a stream with a file, either by initializing a file stream object with a

file name or by using the `open()` method, you can provide a second argument specifying the file mode:

```
ifstream fin("banjo", mode1);
ofstream fout();
fout.open("harp", mode2);
```

The file mode is type `int`, and you can choose from several constants defined in the *ios* class. Table 12-5 lists the constants and their meanings.

If the *ifstream* and *ofstream* constructors and the `open()` methods each take two arguments, how have we gotten by using just one in the previous examples? As you probably have guessed, the prototypes for these class member functions provide default values for the second argument (the *file mode argument*). For instance, the *ifstream* `open()` method and constructor use `ios::in` (open for reading) as the default value for the mode argument, while the *ofstream* `open()` method and constructor use `ios::out` (open for writing) as the default. The *fstream* class doesn't provide a mode default, so you have to provide a mode explicitly when creating an object of that class.

Opening a file in the `ios::out` mode also opens it in the `ios::trunc` mode by default. That means an existing file is truncated when opened; that is, its previous contents are discarded. While this behavior commendably minimizes the danger of running out of disk space, you probably can imagine situations in which you don't want to wipe out every file when you open it. C++, of course, provides other choices. If, for instance, you wish to preserve the file contents and add (append) new material to the end of the file, you can use the `ios::app` mode:

```
ofstream fout("bagels", ios::app);
```

Table 12-5 File Mode Constants

Constant	Meaning
`ios::in`	open file for reading
`ios::out`	open file for writing
`ios::ate`	seek to eof upon opening file
`ios::app`	append to end of file
`ios::trunc`	truncate file if it exists
`ios::nocreate`	open fails if file does not exist
`ios::replace`	open fails if file does exist
`ios::binary`	binary file

See Figure 12-6.

Note that both `ios::ate` and `ios::app` place you (or, more precisely, a file pointer) at the end of the file just opened. The difference between the two is that the `ios::app` mode allows you to add data to the end of the file only, while the `ios::ate` mode lets you write data anywhere in the file, even over old data. One of the later examples demonstrates this technique.

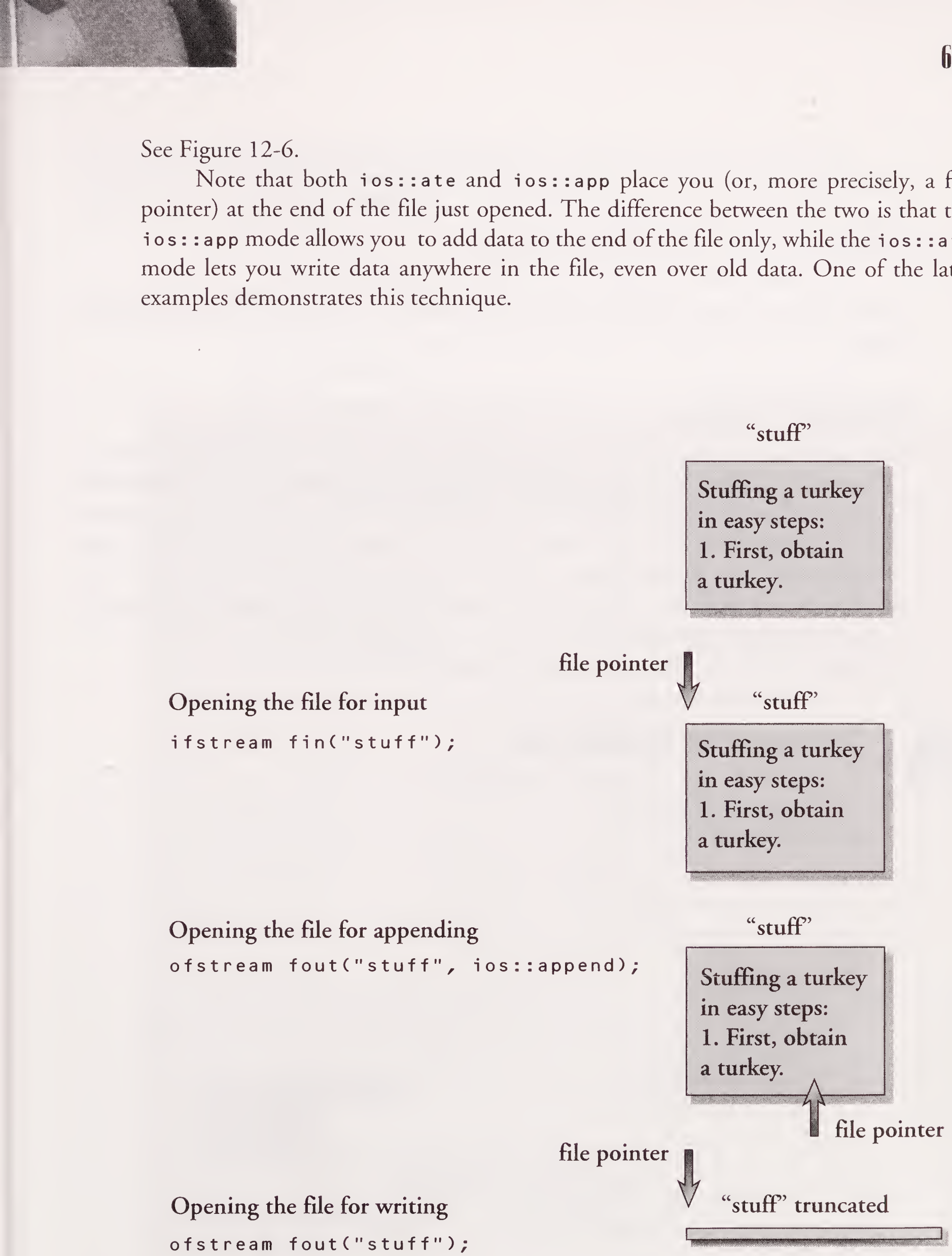


Figure 12-6 *Some file-opening modes*

You can use the C++ bitwise OR operator, represented by the | symbol and discussed in Appendix E, to combine modes. For instance, if you want a program to open a file in the append mode and also want the file opening operation to fail if the file does not exist, use the following statement:

```
ofstream fout("trout", ios::app | ios::nocreate);
```

Clearly, there are many possible combinations of modes. We'll look at a few representative ones.

Appending to a File

Let's begin with a program that appends data to the end of a file. We'll have the program maintain a file containing a guest list. When the program begins, it will display the current contents of the file, if it exists. It can use the `good()` method after attempting to open the file to check if the file exists. Next, the program will open the file for output using the `ios::app` mode. Then it will solicit input from the keyboard to add to the file. Finally, the program will display the revised file contents. Listing 12-16 illustrates how to accomplish these goals. Note how the program uses the `good()` and the `fail()` methods to test if the file has been opened successfully.

LISTING 12-16 appendd.cpp

```
// appendd.cpp -- append information to a file
#include <fstream.h>
#include <stdlib.h>              // for exit()

const char * file = "guests.dat";
const int Len = 40;
int main(void)
{
    char ch;

// show initial contents
    ifstream fin;
    fin.open(file);
    if (fin.good())
    {
            cout << "Here are the current contents of the "
                    << file << " file:\n";
            while (fin.get(ch))
                    cout << ch;
    }
    fin.close();
```

```
// add new names
    ofstream fout(file, ios::app);
    if (fout.fail())
    {
            cerr << "Can't open " << file << " file for output:\n";
            exit(1);
    }

    cout << "Enter guest names (enter a blank line to quit):\n";
    char name[Len];
    cin.get(name, Len).get(ch);
    while (name[0] != '\0')
    {
            fout << name << "\n";
            cin.get(name, Len).get(ch);
    }
    fout.close();

// show revised file
    fin.open(file);
    if (fin.good())
    {
            cout << "Here are the new contents of the "
                 << file << " file:\n";
            while (fin.get(ch))
                    cout << ch;
    }
    fin.close();

    return 0;
}
```

Here's a sample first run. At this point the `guests.dat` file hasn't been created yet, so the program doesn't preview the file.

```
Enter guest names (enter a blank line to quit):
Sylvester Ballone
Phil Kates
Bill Ghan

Here are the new contents of the guests.dat file:
Sylvester Ballone
Phil Kates
Bill Ghan
```

Next time, however, the `guests.dat` file does exist, so the program does preview the file. Also, note that the new data is appended to the old file contents rather than replacing them.

```
Here are the current contents of the guests.dat file:
Sylvester Ballone
Phil Kates
Bill Ghan
Enter guest names (enter a blank line to quit):
Greta Greppo
LaDonna Mobile
Betty Cracker

Here are the new contents of the guests.dat file:
Sylvester Ballone
Phil Kates
Bill Ghan
Greta Greppo
LaDonna Mobile
Betty Cracker
```

You've already read about the major features of the program, but there's one more point meriting review. The program uses this line to read in the guest's name:

```
cin.get(name, Len).get(ch);
```

The `get(name, Len)` portion reads up to, but not including the newline character. The `get(ch)` part reads the newline so that the next input begins at the beginning of the next line. (The code assumes the name doesn't exceed the `Len` limit.) Otherwise, the next input statement would begin at the terminating newline character, read it as a blank line, and thus terminate the loop. (Typically, one first runs into that sort of annoying behavior, then realizes the need to insert the `get(ch)` call.)

Binary Files

When you store data in a file, you can store the data in text form or in binary format. Text form means you store everything as text, even numbers. For instance, storing the value -2.324216e+07 in text form means storing the thirteen characters used to write this number. That requires converting the computer's internal representation of a floating-point number to character form, and that's exactly what the << insertion operator does. Binary format, however, means storing the computer's internal representation of a value. That is, instead of storing characters, store the eight-byte `double` representation of the value. For a character, the binary representation is the same as the text representation—the binary representation of the character's ASCII code. For

numbers, however, the binary representation is much different from the text representation. See Figure 12-7.

Each format has its advantages. The text format is easy to read. You can use an ordinary editor or word processor to read and edit a text file. You easily can transfer a text file from one computer system to another. The binary format is more accurate for numbers, for it stores the exact internal representation of a value. There are no conversion errors or round-off errors. Saving data in binary format can be faster because there is no conversion and because you may be able to save data in larger chunks. And the binary format usually takes less space, depending upon the nature of the data. Transferring to another system can be a problem, however, if the new system uses a different internal representation for values. In that case, you (or someone) may have to write a program to translate one data format to another.

Let's look at a more concrete example. Consider the following structure definition and declaration:

```
struct planet
{
    char name[20];        // name of planet
    double population;    // its population
    double g;             // its acceleration of gravity
};
planet pl;
```

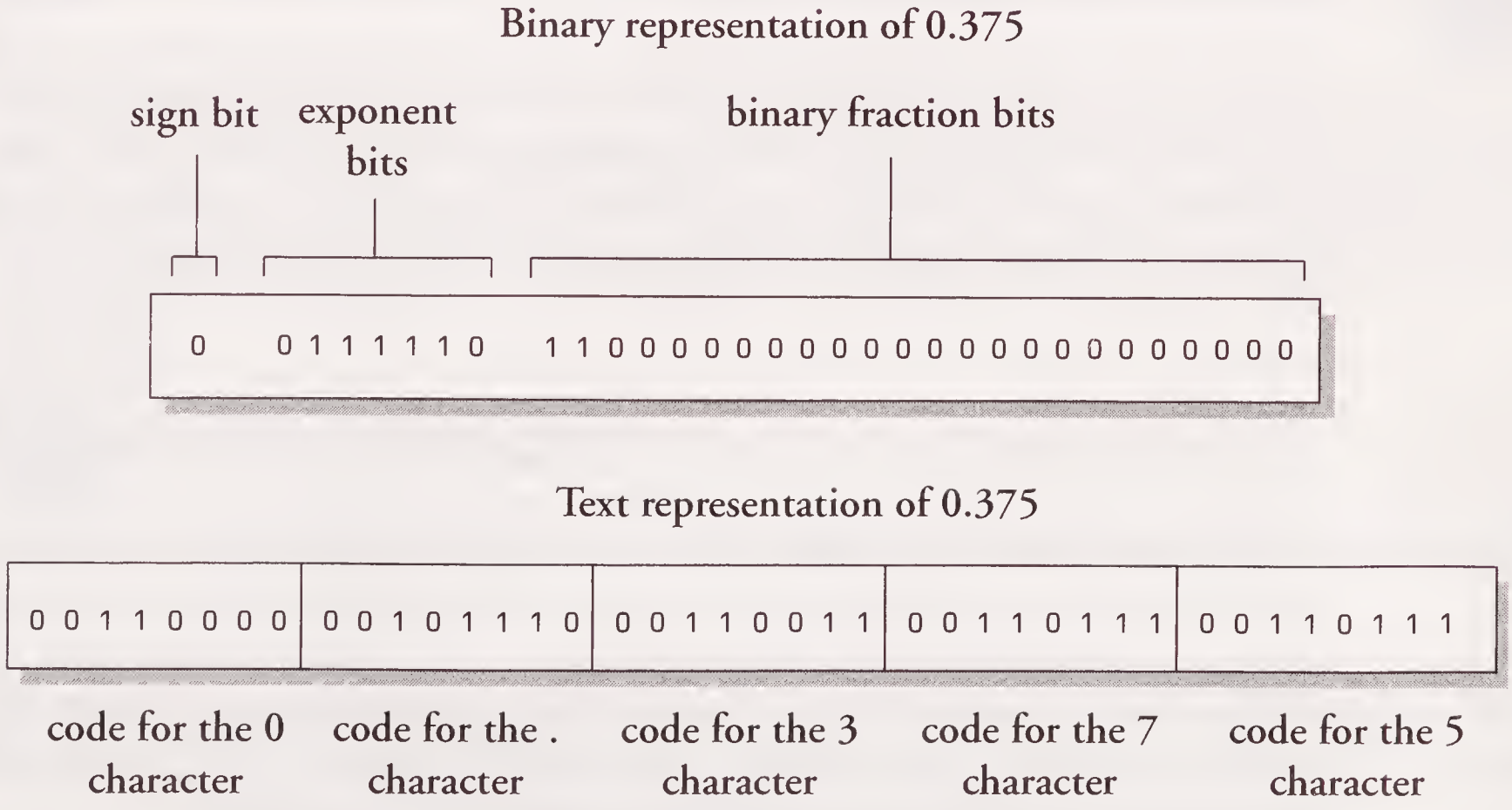


Figure 12-7 *Binary and text representations of a floating-point value*

To save the contents of the structure `pl` in text form, you can do this:

```
ofstream fout("planet.dat", ios::app);
fout << pl.name << " " << pl.population << " " << pl.g << "\n";
```

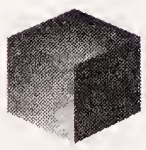
Note that you have to provide each structure member explicitly by using the membership operator, and you have to separate adjacent data for legibility. If the structure contained, say, 30 members, this could get tedious.

To save the same information in binary format, you can do this:

```
ofstream fout("planet.dat", ios::app | ios::binary);
fout.write( (char *) &pl, sizeof pl);
```

This code saves the entire structure as a single unit, using the computer's internal representation of data. You won't be able to read the file as text, but the information will be stored more compactly and precisely than as text. And it certainly is easier to type the code. The two changes are using a binary file mode and using the `write()` member function, so let's examine them.

Some systems, such as DOS, support two file formats: text and binary. If you want to save data in binary form, you'd best use the binary file format. In C++ you do so by using the `ios::binary` constant in the file mode. If you want to know why you should do this on a DOS system, check the discussion on Binary Files and Text Files.



BINARY FILES AND TEXT FILES

Using a binary file mode causes a program to transfer data from memory to a file or vice versa without any hidden translation taking place. Such is not necessarily the case for the default text mode. For instance, consider DOS text files. They represent a newline with a two-character combination—carriage return, linefeed. Macintosh text files represent a newline with a carriage return. UNIX files represent a newline with a linefeed. C++, which grew up on UNIX, also represents a newline with a linefeed. For portability, a DOS C++ program automatically translates the C++ newline to a carriage return, linefeed when writing to a text mode file, and a Macintosh C++ program translates the newline to a carriage return when writing to a file. When reading a text file, these programs convert the local newline back to the C++ form. The text format can cause problems with binary data, for a byte in the middle of a `double` value could have the same bit pattern as the ASCII code for the newline character. Also there are differences in how end-of-file is detected. So you should use the binary file mode when saving data in binary format. (UNIX systems have just one file mode, so on them the binary mode is the same as the text mode.)

To save data in binary form instead of text form, you can use the `write()` member function. This method, recall, copies a specified number of characters from memory to a file. We used it earlier to copy text, but it will copy any type data byte by byte *with no conversion.* For instance, if you pass it the address of a `long` variable and tell it to copy eight characters, it will copy the eight bytes constituting the `long` value verbatim to a file and not convert it to text. The only bit of awkwardness is that you have to type cast the address to type pointer-to-`char`. You can use the same approach to copy an entire `planet` structure. To get the number of bytes, use the `sizeof` operator:

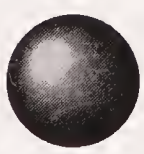
```
fout.write( (char *) &pl, sizeof pl);
```

This statement goes to the address of the `pl` structure and copies the thirty-six bytes (the value of *sizeof pl* expression) beginning at this address to the file being managed by *fout.*

To recover the information from a file, use the corresponding `read()` method with an *ifstream* object:

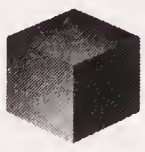
```
ifstream fin("planet..dat", ios::binary);
fin.read((char *) &pl, sizeof pl);
```

This copies *sizeof pl* bytes from the file to the `pl` structure. This same approach can be used with classes. In that case, just the data members are saved, not the methods.



The `read()` and `write()` member functions complement each other. Use `read()` to recover data that has been written to a file with `write()`.

Listing 12-17 uses these methods to create and read a binary file. In form, the program is similar to Listing 12-16, but it uses `write()` and `read()` instead of the insertion operator and the `get()` method. It also uses manipulators to format the screen output.



COMPATIBILITY NOTE

Although the binary file concept is part of ANSI C, many UNIX-based C and C++ implementations do not provide support for the binary file mode. For instance, a UNIX-based C++ compiler may not recognize the `ios::binary` constant. The reason for this oversight is that UNIX systems only have one file type in the first place, so you can use binary operations such as `read()` and `write()` with the standard file format. Therefore, if your UNIX implementation rejects `ios::binary` as a valid constant, just omit it from your program.

LISTING 12-17 binary.cpp

```
// binary.cpp -- append information to a binary file
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>      // for exit()

struct planet
{
    char name[20];       // name of planet
    double population;   // its population
    double g;            // its acceleration of gravity
};

const char * file = "planets.dat";

int main(void)
{
    planet pl;
    cout.setf(ios::fixed, ios::floatfield);

// show initial contents
    ifstream fin;
    fin.open(file, ios::binary);              // binary file
    //NOTE: some UNIX systems require just fin.open(file);
    if (fin.good())
    {
       cout << "Here are the current contents of the "
            << file << " file:\n";
       while (fin.read((char *) &pl, sizeof pl))
       {
            cout << setw(20) << pl.name << ": "
                 << setprecision(0) << setw(12) << pl.population
                 << setprecision(2) << setw(6) << pl.g << "\n";
       }
    }
    fin.close();

// add new data
    ofstream fout(file, ios::app | ios::binary);
    //NOTE: UNIX may require ofstream fout (file ios::app);
    if (fout.fail())
    {
            cerr << 'Can't open ' << file << " file for output:\n";
            exit(1);
    }
```

```
    char ch;
    cout << "Enter planet name (enter a blank line to quit):\n";
    cin.get(pl.name, 20).get(ch);
    while (pl.name[0] != '\0')
    {
            cout << "Enter planetary population: ";
            cin >> pl.population;
            cout << "Enter planet's acceleration of gravity: ";
            cin >> pl.g;
            cin.ignore(80,'\n');
            fout.write((char *) &pl, sizeof pl);
            cout << "Enter planet name (enter a blank line "
                  "to quit):\n";
            cin.get(pl.name, 20).get(ch);
    }
    fout.close();

// show revised file
    fin.open(file, ios::binary);
    if (fin.good())
    {
       cout << "Here are the new contents of the "
            << file << " file:\n";
       while (fin.read((char *) &pl, sizeof pl))
       {
            cout << setw(20) << pl.name << ": "
                 << setprecision(0) << setw(12) << pl.population
                 << setprecision(2) << setw(6) << pl.g << "\n";
       }
    }
    fin.close();

    return 0;
}
```

Here is a sample initial run:

```
Enter planet name (enter a blank line to quit):
Earth
Enter planetary population: 5333000000
Enter planet's acceleration of gravity: 9.81
Enter planet name (enter a blank line to quit):

Here are the new contents of the planets.dat file:
               Earth:   5333000000  9.81
```

And here is a sample follow-up run:

```
Here are the current contents of the planets.dat file:
            Earth:   5333000000  9.81
Enter planet name (enter a blank line to quit):
Bill's Planet
Enter planetary population: 23020020
Enter planet's acceleration of gravity: 8.82
Enter planet name (enter a blank line to quit):

Here are the new contents of the planets.dat file:
                    Earth:     5333000000       9.81
             Bill's Planet:       23020020       8.82
```

We've already explained the major features of the program, but let's reexamine an old point. The program uses this line after reading the planet's g value:

```
cin.ignore(80,'\n');
```

The `ignore(80,'\n')` portion reads and discards input up through the newline character. The `get(ch)` part reads the newline so that the next input begins at the beginning of the next line. The next input statement reads the planet name, which then is checked to see if the first character is the null character. For this test to work correctly, it is essential that input extraction begin at the start of the line and not at the end of the previous line.

If your system supports `getline()`, you can use `cin.getline(pl.name,20)` instead of `cin.get(pl.name, 20).get(ch)`.

Random Access

For our last example, let's look at random access. That means moving directly to any location in the file instead of moving through it sequentially. The random access approach is often used with database files. A program will maintain a separate index file giving the location of data in the main data file. Then it can jump directly to that location, read the data there, and perhaps modify it. This approach is done most simply if the file consists of a collection of equal-sized *records*. Each record would represent a related collection of data. For instance, in the preceding example, each file record would represent all the data about a particular planet. A file record corresponds rather naturally to a program structure or class.

We'll base our example on the binary file program in Listing 12-18, for the `planet` structure provides a template for a file record. To add to the creative tension of programming, the example will open the file in a read-and-write mode so that it can both read and modify a record. You can do thisby creating an *fstream* object. The *fstream* class derives from the *iostream* class, which, in turn, is based on both *istream* and *ostream* classes, so it inherits the methods of both. It also inherits two buffers, one for input and

one for output, and synchronizes the handling of the two buffers. That is, as the program reads the file or writes to it, it moves both an input pointer in the input buffer and an output pointer in the output buffer in tandem.

The example will do the following:

- Display the current contents of the `planets.dat` file
- Ask which record you wish to modify
- Modify that record
- Show the revised file

A more ambitious program would use a menu and a loop to let you select from this list of actions indefinitely, but our version will perform each action just once. This allows you to examine several aspects of read/write files without getting bogged down in matters of program design.

The first question to answer is what file mode to use. In order to read the file, you need the `ios::in` mode. For binary I/O, you need the `ios::binary` mode. (Again, on UNIX systems you can omit, indeed, may have to omit, this mode.) In order to write to the file, you need the `ios::out` or the `ios::app` mode. However, the append mode allows a program to add data to the end of the file only. The rest of the file is read only; that is, you can read the original data, but not modify it. So you have to use `ios::out`. However, by default, using that mode truncates a file, discarding the data. You can negate that aspect by also using the `ios::ate` mode. This moves the file pointer to the end of the file, preserving the file data. As mentioned earlier, you use the | operator to combine modes. Thus you need the following statement to set up business:

```
finout.open(file,ios::in | ios::out | ios::ate | ios::binary);
```

Next, you need a way to move through a file. The *fstream* class inherits two methods for this: the `seekg()` moves the input pointer to a given file location, and the `seekp()` moves the output pointer to a given file location. (Actually, since the *fstream* class uses buffers for intermediate storage of data, the pointers point to locations in the buffers, not in the actual file.) You also can use `seekg()` with an *ifstream* object and `seekp()` with an *ostream* object. Here are the `seekg()` prototypes:

```
istream & seekg(streampos);
istream & seekg(streamoff, seek_dir);
```

The first prototype represents locating a file position measured in bytes from the beginning of a file. The second prototype represents locating a file position measured, in bytes, as an offset from a file location specified by the second argument.

Let's take a look at the arguments to `seekg()`. The `streampos` and `streamoff` types are integer types defined by `typedef` statements in the `iostream.h` header file.

The `streampos` argument represents the file position measured in bytes from the beginning of the file. File byte numbering begins with zero. So the statement

```
fin.seekg(112);
```

locates the file pointer at byte 112, which would be the 113th byte in the file. The `streamoff` argument represents the file position in bytes measured as an offset from one of three locations. The `seek_dir` argument is another integer type defined in `iostream.h`. It can be set to one of three constants defined in the *ios* class. The constant `ios::beg` means measure the offset from the beginning of the file. The constant `ios::cur` means measure the offset from the current position. The constant `ios::end` means measure the offset from the end of the file. Here are some sample calls, assuming `fin` is an *ifstream* object:

```
fin.seekg(30);                  // go to byte number 30 in the file
fin.seekg(30, ios::beg);        // same as the above
fin.seekg(-1, ios::cur);        // back up one byte
fin.seekg(0, ios::end);         // go to the end of the file
fin.seekg(0);                   // go to the start of the file
```

If you want to check the current position of a file pointer, you can use the `tellg()` method for input streams and the `tellp()` methods for output streams. Each returns a `streampos` value representing the current position, in bytes, measured from the beginning of the file. When you create an *fstream* object, the input and output pointers move in tandem, so `tellg()` and `tellp()` return the same value. But if you use an *istream* object to manage the input stream and an *ostream* object to manage the output stream to the same file, then the input and output pointers move independently of one another, and `tellg()` and `tellp()` can return different values.

Because our example uses the `ios::ate` mode, the program goes to the end of the file when opening it. (Using `ios::ate`, recall, prevents the program from truncating the file.) But you can then use `seekg()` to go to the file beginning. Here is a section of code that opens the file, goes to the beginning, and displays the file contents:

```
fstream finout;                 // read and write streams
finout.open(file,ios::in | ios::out | ios::ate | ios::binary);
//NOTE: Some UNIX systems require omitting | ios::binary
if (finout.good())
{
    finout.seekg(0, ios::beg); // go to beginning
    cout << "Here are the current contents of the "
            << file << " file:\n";
    int ct = 0;
    while (finout.read((char *) &pl, sizeof pl))
    {
            cout << ct++ << ": " << setw(20) << pl.name << ": "
```

```
                    << setprecision(0) << setw(12) << pl.population
                    << setprecision(2) << setw(6) << pl.g << "\n";
    }
    if (finout.eof())
            finout.clear();      // clear eof flag
    else
    {
            cerr << "Error in reading " << file << ".\n";
            exit(1);
    }
}
else
{
    cerr << file << " could not be opened-- bye.\n";
    exit(2);
}
```

This is similar to the start of Listing 12-17, but there are some changes and additions. First, as just described, the program uses an *fstream* object with a read/write mode, and it uses `seekg()` to position the file pointer at the start of the file. Next, the program makes the minor change of numbering the records as they are displayed. Then it makes the following important addition:

```
if (finout.eof())
            finout.clear();      // clear eof flag
else
{
    cerr << "Error in reading " << file << ".\n";
    exit(1);
}
```

The problem is that once the program reads and displays the entire file, it sets the `eof` flag on. This convinces the program that it's finished with the file and disables any further reading of or writing to the file. Using the `clear()` method resets the stream state, turning off the `eof` flag. Now the program can once again access the file. The `else` part handles the possibility that the program quit reading the file for some reason other than reaching the end-of-file, such as a hardware failure.

The next step is to identify the record to be changed, then change it. To do this, the program asks the user to enter a record number. Multiplying the number by the number of bytes in a record yields the byte number for the beginning of the record. If `record` is the record number, the desired byte number is `record * sizeof pl`:

```
char ch;
cout << "Enter the record number you wish to change: ";
streampos record;
cin >> record;
```

```
cin.ignore(80,'\n');    // get rid of newline
if (record < 0 || record >= ct)
{
    cerr << "Invalid record number -- bye\n";
    exit(3);
}
finout.seekg(record * sizeof pl);
```

The variable *ct* represents the number of records; the program exits if you try to go beyond the limits of the file.

Next, the program displays the current record:

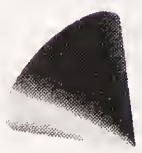
```
finout.read((char *) &pl, sizeof pl);
cout << "Your selection:\n";
cout << record << ": " << setw(20) << pl.name << ": "
     << setprecision(0) << setw(12) << pl.population
     << setprecision(2) << setw(6) << pl.g << "\n";
if (finout.eof())
    finout.clear();                    // clear eof flag
finout.seekp(record * sizeof pl);      // go back
```

After displaying the record, the program resets the file pointer to the beginning of the record. Then it lets you change the record:

```
cout << "Enter planet name: ";
cin.get(pl.name, 20).get(ch);
cout << "Enter planetary population: ";
cin >> pl.population;
cout << "Enter planet's acceleration of gravity: ";
cin >> pl.g;
finout.write((char *) &pl, sizeof pl) << flush;
if (finout.fail())
{
    cerr << "Error on attempted write\n";
    exit(5);
}
```

The program flushes the output to guarantee that the file is updated before proceeding to the next stage.

Finally, to display the revised file, the program uses `seekg()` to reset the file pointer to the beginning. Listing 12-18 shows the complete program.



LISTING 12-18 random.cpp

```
// random.cpp -- random access to a binary file
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>              // for exit()
```

```
struct planet
{
    char name[20];       // name of planet
    double population;   // its population
    double g;            // its acceleration of gravity
};

const char * file = "planets.dat";

int main(void)
{
    planet pl;
    cout.setf(ios::fixed, ios::floatfield);

// show initial contents
    fstream finout;                    // read and write streams
    finout.open(file,ios::in | ios::out | ios::ate |
                        ios::binary);
    //NOTE: Some UNIX systems require omitting | ios::binary
    int ct = 0;
    if (finout.good())
    {
            finout.seekg(0);           // go to beginning
            cout << "Here are the current contents of the "
                 << file << " file:\n";
            while (finout.read((char *) &pl, sizeof pl))
            {
                 cout << ct++ << ": " << setw(20) << pl.name << ": "
                      << setprecision(0) << setw(12) << pl.population
                      << setprecision(2) << setw(6) << pl.g << "\n";
            }
            if (finout.eof())
                 finout.clear();       // clear eof flag
            else
            {
                 cerr << "Error in reading " << file << ".\n";
                 exit(1);
            }
    }
    else
    {
            cerr << file << " could not be opened -- bye.\n";
            exit(2);
    }

// change a record
    char ch;
```

```
cout << "Enter the record number you wish to change: ";
streampos record;
cin >> record;
cin.ignore(80,'\n');                    // get rid of newline
if (record < 0 || record >= ct)
{
        cerr << "Invalid record number -- bye\n";
        exit(3);
}
finout.seekg(record * sizeof pl);
if (finout.fail())
{
        cerr << "Error on attempted seek\n";
        exit(4);
}

finout.read((char *) &pl, sizeof pl);
cout << "Your selection:\n";
cout << record << ": " << setw(20) << pl.name << ": "
     << setprecision(0) << setw(12) << pl.population
     << setprecision(2) << setw(6) << pl.g << "\n";
if (finout.eof())
        finout.clear();                 // clear eof flag
    finout.seekp(record * sizeof pl);  // go back

    cout << "Enter planet name: ";
    cin.get(pl.name, 20).get(ch);
    cout << "Enter planetary population: ";
    cin >> pl.population;
    cout << "Enter planet's acceleration of gravity: ";
    cin >> pl.g;
    finout.write((char *) &pl, sizeof pl) << flush;
    if (finout.fail())
    {
            cerr << "Error on attempted write\n";
            exit(5);
    }

// show revised file
    ct = 0;
    finout.seekg(0);                    // go to beginning of file
    cout << "Here are the new contents of the " << file
        << " file:\n";
```




```
    while (finout.read((char *) &pl, sizeof pl))
    {
            cout << ct++ << ": " << setw(20) << pl.name << ": "
                 << setprecision(0) << setw(12) << pl.population
                 << setprecision(2) << setw(6) << pl.g << "\n";
    }
    finout.close();

    return 0;
}
```

Here's a sample run based on a `planets.dat` file that has had a few more entries added since you last saw it:

```
Here are the current contents of the planets.dat File:
0:            Earth:     5333000000        9.81
1:   Bill's Planet:       23020020        8.82
2:           Tramtor:    58000000000       15.03
3:           Trellan:        4256000        9.62
4:        Freestone:     3845120000        8.68
5:        Taanagoot:      350000002       10.23
6:            Marin:         232000        9.79
Enter the record number you wish to change: 2
Your selection:
2:           Tramtor:    58000000000       15.03
Enter planet name: Trantor
Enter planetary population: 59500000000
Enter planet's acceleration of gravity: 10.53
Here are the new contents of the planets.dat file:
0:            Earth:     5333000000        9.81
1:   Bill's Planet:       23020020        8.82
2:           Trantor:    59500000000       10.53
3:           Trellan:        4256000        9.62
4:        Freestone:     3845120000        8.68
5:        Taanagoot:      350000002       10.23
6:            Marin:         232000        9.79
```

Using the techniques in this program, you can extend it to allow you to add new material and to delete records. If you were to expand the program, it would be a good idea to reorganize it by using classes and functions. For example, you could convert the `planet` structure to a class definition, then overload the `<<` insertion operator so that `cout << pl` displays the class data members formatted as in the example.

SUMMARY

A stream is a flow of bytes into or out of a program. A buffer is a temporary holding area in memory which acts as an intermediary between a program and a file or other I/O device. Information can be transferred between a buffer and a file using large chunks of data of the size most efficiently handled by devices like disk drives. And information can be transferred between a buffer and a program in a byte-by-byte flow that often is more convenient for the processing done in a program. C++ handles input by connecting a buffered stream to a program and to its source of input. Similarly, C++ handles output by connecting a buffered stream to a program and to its output target. The `iostream.h` and `fstream.h` files constitute an I/O class library that defines a rich set of classes for managing streams. C++ programs that include the `iostream.h` file automatically open four streams, managing them with four objects. The `cin` object manages the standard input stream, which, by default, connects to the standard input device, typically a keyboard. The `cout` object manages the standard output stream, which, by default, connects to the standard output device, typically a monitor. The `cerr` and `clog` objects manage unbuffered and buffered streams connected to the standard error device, typically a monitor.

The I/O class library provides a variety of useful methods. The *istream* class defines versions of the extraction operator (`>>`) that recognize all the basic C++ types and that convert character input to those types. The `get()` family of methods and the `getline()` method provide further support for single-character input and for string input. Similarly, the *ostream* class defines versions of the insertion operator (`<<`) that recognize all the basic C++ types and that convert them to suitable character output. The `put()` method provides further support for single-character output.

You can control how a program formats output by using *ios* class methods and by using manipulators (functions that can be concatenated with insertion) defined in the `iostream.h` and the `iomanip.h` files. These methods and manipulators let you control the number base, the field width, the number of decimal places displayed, the system used to display floating-point values, and other elements.

The `fstream.h` file provides class definitions that extend the `iostream.h` methods to file I/O. The *ifstream* class derives from the *istream* class. By associating an *ifstream* object with a file, you can use all the *istream* methods for reading the file. Similarly, associating an *ofstream* object with a file lets you use the `ostream` methods to write to a file. And associating an *fstream* object with a file lets you employ both input and output methods with the file.

To associate a file with a stream, you can provide the file name when initializing a file stream object or you can first create a file stream object, then use the `open()` method to associate the stream with a file. The `close()` method terminates the connection be-

tween a stream and a file. The class constructors and the `open()` method take an optional second argument that provides the file mode. The file mode determines such things as whether the file is to be read and/or written to, whether opening a file for writing truncates it or not, whether attempting to open a non-existing file is an error or not, and whether to use the binary or text mode.

A text file stores all information in character form. For instance, numeric values are converted to character representations. The usual insertion and extraction operators, along with `get()` and `getline()`, support this mode. A binary file stores all information using the same binary representation the computer uses internally. Binary files store data, particularly floating-point values, more accurately and compactly than text files, but they are less portable. The `read()` and `write()` methods support binary input and output.

The `seekg()` and `seekp()` functions provide random access for files. These class methods let you position a file pointer relative to the beginning of a file, relative to the end, or relative to the current position. The `tellg()` and `tellp()` methods report the current file position.

Review Questions

1. What role does the `iostream.h` file play in C++ I/O?
2. Why does typing a number such as *121* as input require a program to make a conversion?
3. What's the difference between the standard output and the standard error?
4. Why is `cout` able to display various C++ types without being provided explicit instructions for each type?
5. What feature of the output method definitions allows you to concatenate output?
6. Write a program that requests an integer and then displays it in decimal, octal, and hexadecimal form. Display each form on the same line in fields 15 characters wide and use the C++ number base prefixes.
7. Write a program that requests the information shown below and that formats it as shown:

```
Enter your name: Billy Gruff
Enter your hourly wages: 12
```

```
Enter number of hours worked: 7.5
First format:
               Billy Gruff: $    12.00:  7.5
Second format:
Billy Gruff                 : $12.00    :7.5
```

8. Consider the following program:

```
//rq12-8.cpp
#include <iostream.h>

int main(void)
{
 char ch;
 int ct1 = 0;

 cin >> ch;
 while (ch != 'q')
   {
        ct1++;
        cin >> ch;
   }
 int ct2 = 0;
 cin.get(ch);
 while (ch != 'q')
   {
        ct2++;
        cin.get(ch);
   }
 cout << "ct1 = " << ct1 << "; ct2 = " << ct2 << "\n";

 return 0;
}
```

What does it print, given the following input:

```
I see a q(RETURN)
I see a q(RETURN)
```

Here (RETURN) signifies pressing the return or enter key.

9. Both of the following statements read and discard characters up to and including the end of a line. In what way does the behavior of one differ from that of the other?

```
while (cin.get() != "\n")
 continue;
cin.ignore(80, "\n");
```

10. Write a program that counts the number of characters up to the first `$` in input and that leaves the `$` in the input stream.

11. Write a program that copies your keyboard input (up to simulated end-of-file) to a file named on the command line.

12. Write a program that copies one file to another. Have the program take the file names from the command line. Have the program refuse to copy if there already is a file having the target name.

Number Bases

Appendix A

Our method for writing numbers is based on powers of ten. For instance, consider the number 2468. The 2 represents 2 thousands, the 4 represents 4 hundreds, the 6 represents 6 tens, and the 8 represents 8 ones:

$$2468 = 2 \times 1000 + 4 \times 100 + 6 \times 10 + 8 \times 1$$

One thousand is $10 \times 10 \times 10$, which can be written as 10^3, or 10 to the third power. Using this notation, we can write the preceding relationship this way:

$$2468 = 2 \times 10^3 + 4 \times 10^2 + 6 \times 10^1 + 8 \times 10^0$$

Because our number notation is based on powers of ten, we refer to it as *base ten*, or *decimal*, notation. One can just as easily pick another number as a base. C++ lets you use *base eight* (*octal*) and *base sixteen* (*hexadecimal*) notation for writing integer numbers. (Note: 10^0 is 1, as is any nonzero number to the zero power.)

Octal Integers

Octal numbers are based on powers of eight, so base eight notation uses the digits 0-7 in writing numbers. C++ uses a 0 prefix to indicate octal notation. Thus, 0177 is an octal value. You can use powers of eight to find the equivalent base ten value:

$$0177 \text{ (octal)} = 1 \times 8^2 + 7 \times 8^1 + 7 \times 8^0 = 1 \times 64 + 7 \times 8 + 7 \times 1 = 127 \text{ (decimal)}$$

The UNIX operating system often uses octal representation of values, which is why C++ and C provide octal notation.

Hexadecimal Numbers

Hexadecimal numbers are based on powers of sixteen. That means 10 in hexadecimal represents the value 16 + 0, or 16. To represent the values between 9 and hexadecimal 16, we need a few more digits. Standard hexadecimal notation uses the letters a-f for that purpose. C++ accepts either lowercase or uppercase versions of these characters, as shown in Table A-1.

TABLE A-1 Hexadecimal Digits

Hexadecimal Digits	Decimal Value	Hexadecimal Digits	Decimal Value
a or A	10	d or D	13
b or B	11	e or E	14
c or C	12	f or F	15

C++ uses a 0x or a 0X notation to indicate hexadecimal notation. Thus 0x2B3 is a hexadecimal value. To find its decimal equivalent, you can evaluate the powers of sixteen:

$$0x2B3 \text{ (hex)} = 2 \times 16^2 + 11 \times 16^1 + 3 \times 16^0 = 2 \times 256 + 11 \times 16 + 3 = 691 \text{ (decimal)}$$

Hardware documentation often uses hexadecimal notation to represent values such as memory locations and port numbers.

Binary Numbers

Whether you use decimal, octal, or hexadecimal notation for writing an integer, the computer stores it as a *binary*, or base two, value. Binary notation uses just two digits, 0 and 1. As an example, 10011011 is a *binary* number. Note, however, that C++ doesn't provide for writing a number in binary notation. Binary numbers are based on powers of two:

$$\begin{aligned} 10011011 &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 128 + 0 + 0 + 16 + 8 + 0 + 2 + 0 \\ &= 154 \end{aligned}$$

Binary notation makes a nice match to computer memory, in which each individual unit, called a *bit*, can be set to off or on. Just identify the off setting with 0 and the on setting with 1. Memory commonly is organized in units called *bytes*, with each *byte* being eight bits. The bits in a byte are numbered corresponding to the associated power of two. Thus the rightmost bit is bit number 0, the next bit is bit 1, and so on. Figure A-1, for example, represents a two-byte integer.

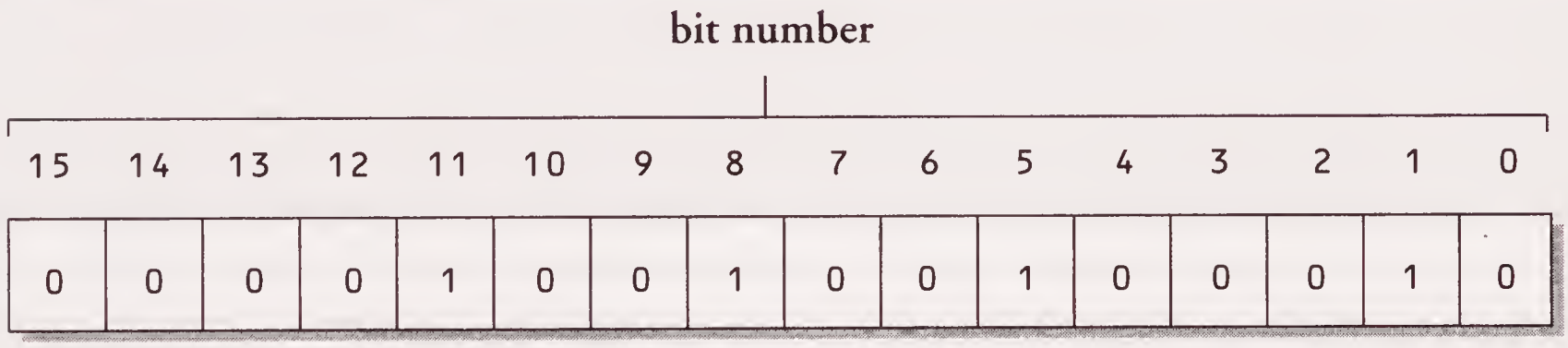


Figure A-1 *A two-byte integer value*

C++ Keywords

Appendix B

Keywords are identifiers that form the vocabulary of a programming language. They may not be used for other purposes, such as serving as a variable name. The following list shows C++'s keywords; not all of them are currently implemented. Keywords shown in **boldface** are also keywords in ANSI C.

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

The ASCII Character Set

Appendix C

Computers store characters using a numeric code. The ASCII code is the most commonly used code in the United States C++ lets you represent most single characters directly, by including the character in single quotation marks, as in '*A*' for the A character. You can also represent a single character by using the octal or hex code preceded by a backslash; for instance, '*\012*' and '*\0xa*' both represent the linefeed (LF) character. Such *escape sequences* can also be part of a string, as in "*Hello,\012my dear*".

When used as a prefix in the following table, the ^ character denotes using a Control key.

Decimal	Octal	Hex	Binary	Character	ASCII Name
0	0	0	00000000	^@	NUL
1	01	0x1	00000001	^A	SOH
2	02	0x2	00000010	^B	STX
3	03	0x3	00000011	^C	ETX
4	04	0x4	00000100	^D	EOT
5	05	0x5	00000101	^E	ENQ
6	06	0x6	00000110	^F	ACK
7	07	0x7	00000111	^G	BEL
8	010	0x8	00001000	^H	BS
9	011	0x9	00001001	^I	HT
10	012	0xa	00001010	^J	LF
11	013	0xb	00001011	^K	VT

Decimal	Octal	Hex	Binary	Character	ASCII Name
12	014	0xc	00001100	^L	FF
13	015	0xd	00001101	^M	CR
14	016	0xe	00001110	^N	SO
15	017	0xf	00001111	^O	SI
16	020	0x10	00010000	^P	DLE
17	021	0x11	00010001	^Q	DC1
18	022	0x12	00010010	^R	DC2
19	023	0x13	00010011	^S	DC3
20	024	0x14	00010100	^T	DC4
21	025	0x15	00010101	^U	NAK
22	026	0x16	00010110	^V	SYN
23	027	0x17	00010111	^W	ETB
24	030	0x18	00011000	^X	CAN
25	031	0x19	00011001	^Y	EM
26	032	0x1a	00011010	^Z	SUB
27	033	0x1b	00011011	^esc	ESC
28	034	0x1c	00011100	^\	FS
29	035	0x1d	00011101	^]	GS
30	036	0x1e	00011110	^^	RS
31	037	0x1f	00011111	^_	US
32	040	0x20	00100000	space	SP
33	041	0x21	00100001	!	
34	042	0x22	00100010	“	
35	043	0x23	00100011	#	
36	044	0x24	00100100	$	
37	045	0x25	00100101	%	
38	046	0x26	00100110	&	
39	047	0x27	00100111	‘	
40	050	0x28	00101000	(	
41	051	0x29	00101001	)	
42	052	0x2a	00101010	*	

Decimal	Octal	Hex	Binary	Character	ASCII Name
43	053	0x2b	00101011	+	
44	054	0x2c	00101100	,	
45	055	0x2d	00101101	-	
46	056	0x2e	00101110	.	
47	057	0x2f	00101111	/	
48	060	0x30	00110000	0	
49	061	0x31	00110001	1	
50	062	0x32	00110010	2	
51	063	0x33	00110011	3	
52	064	0x34	00110100	4	
53	065	0x35	00110101	5	
54	066	0x36	00110110	6	
55	067	0x37	00110111	7	
56	070	0x38	00111000	8	
57	071	0x39	00111001	9	
58	072	0x3a	00111010	:	
59	073	0x3b	00111011	;	
60	074	0x3c	00111100	<	
61	075	0x3d	00111101	=	
62	076	0x3e	00111110	>	
63	077	0x3f	00111111	?	
64	0100	0x40	01000000	@	
65	0101	0x41	01000001	A	
66	0102	0x42	01000010	B	
67	0103	0x43	01000011	C	
68	0104	0x44	01000100	D	
69	0105	0x45	01000101	E	
70	0106	0x46	01000110	F	
71	0107	0x47	01000111	G	
72	0110	0x48	01001000	H	
73	0111	0x49	01001001	I	

Decimal	Octal	Hex	Binary	Character	ASCII Name
74	0112	0x4a	01001010	J	
75	0113	0x4b	01001011	K	
76	0114	0x4c	01001100	L	
77	0115	0x4d	01001101	M	
78	0116	0x4e	01001110	N	
79	0117	0x4f	01001111	O	
80	0120	0x50	01010000	P	
81	0121	0x51	01010001	Q	
82	0122	0x52	01010010	R	
83	0123	0x53	01010011	S	
84	0124	0x54	01010100	T	
85	0125	0x55	01010101	U	
86	0126	0x56	01010110	V	
87	0127	0x57	01010111	W	
88	0130	0x58	01011000	X	
89	0131	0x59	01011001	Y	
90	0132	0x5a	01011010	Z	
91	0133	0x5b	01011011	[	
92	0134	0x5c	01011100	\	
93	0135	0x5d	01011101	]	
94	0136	0x5e	01011110	^	
95	0137	0x5f	01011111	_	
96	0140	0x60	01100000	‘	
97	0141	0x61	01100001	a	
98	0142	0x62	01100010	b	
99	0143	0x63	01100011	c	
100	0144	0x64	01100100	d	
101	0145	0x65	01100101	e	
102	0146	0x66	01100110	f	
103	0147	0x67	01100111	g	
104	0150	0x68	01101000	h	

Decimal	Octal	Hex	Binary	Character	ASCII Name
105	0151	0x69	01101001	i	
106	0152	0x6a	01101010	j	
107	0153	0x6b	01101011	k	
108	0154	0x6c	01101100	l	
109	0155	0x6d	01101101	m	
110	0156	0x6e	01101110	n	
111	0157	0x6f	01101111	o	
112	0160	0x70	01110000	p	
113	0161	0x71	01110001	q	
114	0162	0x72	01110010	r	
115	0163	0x73	01110011	s	
116	0164	0x74	01110100	t	
117	0165	0x75	01110101	u	
118	0166	0x76	01110110	v	
119	0167	0x77	01110111	w	
120	0170	0x78	01111000	x	
121	0171	0x79	01111001	y	
122	0172	0x7a	01111010	z	
123	0173	0x7b	01111011	{	
124	0174	0x7c	01111100	|	
125	0175	0x7d	01111101	}	
126	0176	0x7e	01111110	~	
127	0177	0x7f	01111111	del, rubout	

Operator Precedence

Appendix D

Operator precedence determines the order in which operators are applied to a value. C++ operators come in sixteen precedence groups, which are presented in the table below. Those in group 1 have the highest precedence, and so on. If two operators apply to the same operand (something upon which an operator operates), the operator with the higher precedence applies first. If the two operators have the same precedence, C++ uses associativity rules to determine which operator binds more tightly. All operators in the same group have the same precedence and the same associativity, which is either left to right (L-R in the table) or right to left (R-L in the table). Left-to-right associativity means to apply the lefthand operator first, while right-to-left associativity means to apply the righthand operator first.

Some symbols, such as * and &, are used for more than one operator. In such cases, one form is unary (one operand) and the other form is binary (two operands), and the compiler uses the context to determine which is meant. The table labels operator groups unary or binary for those cases in which the same symbol is used two ways.

Here are some examples of precedence and associativity:

```
3 + 5 * 6
```

The * operator has higher precedence than the + operator, so it is applied to the 5 first, making the expression 3 + 30, or 33.

```
120 / 6 * 5
```

Both / and * have the same precedence, but these operators associate from left to right. That means the operator to the left of the shared operand (6) is applied first, so the expression becomes 20 * 5, or 100.

```
char * str = "Whoa";
char ch = *str++;
```

Both the unary * and the ++ operators have the same precedence, but they associate right to left. This means the increment operator operates upon *str* and not **str*. That is, the operation increments the pointer, making it point to the next character, rather than altering the character pointed to. However, because ++ is the postfix form, the pointer is incremented after the original value of **str* is assigned to *ch*. Therefore this expression assigns the character '*W*' to *ch*, then moves *str* to point to the '*h*' character.

C++ Operator Precedence and Associativity

Precedence	Operator	Assoc.	Meaning
1	()	L-R	function call, grouping
	[]		array subscript
	->		indirect membership operator
	::		scope resolution operator
	.		direct membership operator
2 (all unary)	!	R-L	logical negation
	~		bitwise negation (one's complement)
	+		unary plus (positive sign)
	-		unary minus (negative sign)
	++		increment operator, prefix and postfix
	--		decrement operator, prefix and postfix
	&		address
	*		dereference (indirect value)
	sizeof		size in bytes
	new		dynamically allocate storage
	delete		dynamically free storage
3 (all binary)	*	L-R	multiply
	/		divide
	%		modulus (remainder)
4	.*	L-R	member dereference
	->*		indirect member dereference
5 (all binary)	+	L-R	addition
	-		subtraction

Precedence	Operator	Assoc.	Meaning
6	<<	L-R	left shift
	>>		right shift
7	<	L-R	less than
	<=		less than or equal to
	>=		greater than or equal to
	>		greater than
8	==	L-R	equal to
	!=		not equal to
9 (binary)	&	L-R	bitwise AND
10	^	L-R	bitwise XOR (exclusive OR)
11	|	L-R	bitwise OR
12	&&	L-R	logical AND
13	||	L-R	logical OR
14	?:	R-L	conditional
15	=	R-L	simple assignment
	*=		multiply and assign
	/=		divide and assign
	%=		take remainder and assign
	+=		add and assign
	-=		subtract and assign
	&=		bitwise AND and assign
	^=		bitwise XOR and assign
	|=		bitwise OR and assign
	<<=		left shift and assign
	>>=		right shift and assign
16	,	L-R	combine two expressions into one

Other Operators

Appendix E

In order to avoid terminal obesity, the main text of this book doesn't cover two groups of operators. The first group consists of the bitwise operators, which let you manipulate individual bits in a value; these operators were inherited from C. The second group consists of two-member dereferencing operators; they are C++ additions. This appendix briefly summarizes these operators.

Bitwise Operators

The bitwise operators operate upon the bits of integer values. For example, the left-shift operator moves bits to the left and the bitwise negation operator turns each one to a zero, and each zero to a one. Altogether, C++ has six such operators: <<, >>, ~, &, |, and ^.

The Shift Operators

The left-shift operator has the following syntax:

```
value << shift
```

Here *value* is the integer value to be shifted, and *shift* is the number of bits to shift. For example, *13 << 3* means shift all the bits in the value *13* three places to the left. The vacated places are filled with zeros, and bits shifted past the end are discarded. See Figure E-1.

Since each bit position represents a value twice that of the bit to the right (see Appendix A), shifting one bit position is equivalent to multiplying the value by 2. Similarly, shifting two bit positions is equivalent to multiplying by 2^2, and shifting n positions is equivalent to multiplying by 2^n.

The left-shift operator provides a capability often found in assembly languages. However, an assembly left-shift operator directly alters the contents of a register, while the C++ operator produces a new value without altering existing values. For instance, consider the following:

```
int x = 20;
int y = x << 3;
```

This code doesn't change the value of *x*. The expression *x* << *3* uses the value of *x* to produce a new value, much as `x + 3` produces a new value without altering *x*.

If you want to use the left-shift operator to change the value of a variable, you also have to use assignment. You can use regular assignment or the <<= operator, which combines shifting with assignment.

```
x = x << 4;         // regular assignment
y <<= 2;            // shift and assign
```

The right-shift operator (>>), as you might expect, shifts bits to the right. It has the following syntax:

```
value >> shift
```

Here *value* is the integer value to be shifted, and *shift* is the number of bits to shift. For example, *17* >> *2* means shift all the bits in the value *17* two places to the right. For unsigned integers, the vacated places are filled with zeros, and bits shifted past the end are discarded. For signed integers, vacated places may be filled with zeros or else with the

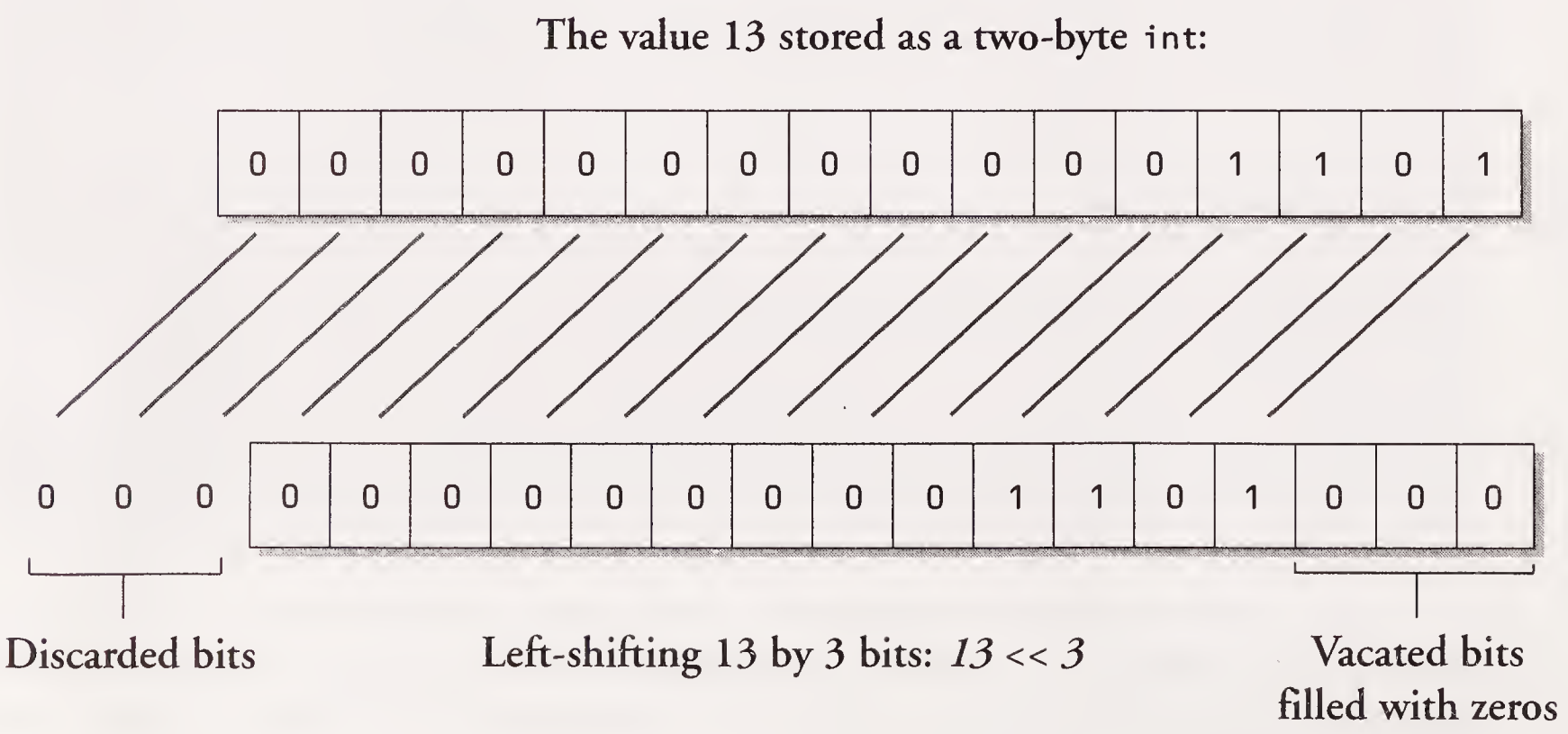


Figure E-1 *The left-shift operator*

value of the original leftmost bit. The choice depends upon the implementation. See Figure E-2.

Shifting one place to the right is equivalent to integer division by 2. In general, shifting *n* places to the right is equivalent to integer division by 2^n.

C++ also defines a shift-and-assign operator if you wish to replace the value of a variable by the shifted value:

```
int q = 43;
q >>= 2;   // replace 43 by 43 >> 2, or 10
```

On some systems, using left- and right-shift operators may produce faster integer multiplication and division by two than using the division operator, but as compilers get better at optimizing code, such differences are fading.

THE LOGICAL BITWISE OPERATORS

The logical bitwise operators are analogous to the regular logical operators, except they apply to a value on a bit-by-bit basis rather than to the whole. For instance, consider the regular negation operator (*!*) and the bitwise negation operator (*~*). The *!* operator converts a true value (nonzero) to false (zero) and a false value (zero) to true (1). The *~* operator converts each individual bit to its opposite (1 to 0 and 0 to 1). For instance, consider the `unsigned char` value of 3:

```
unsigned char x = 3;
```

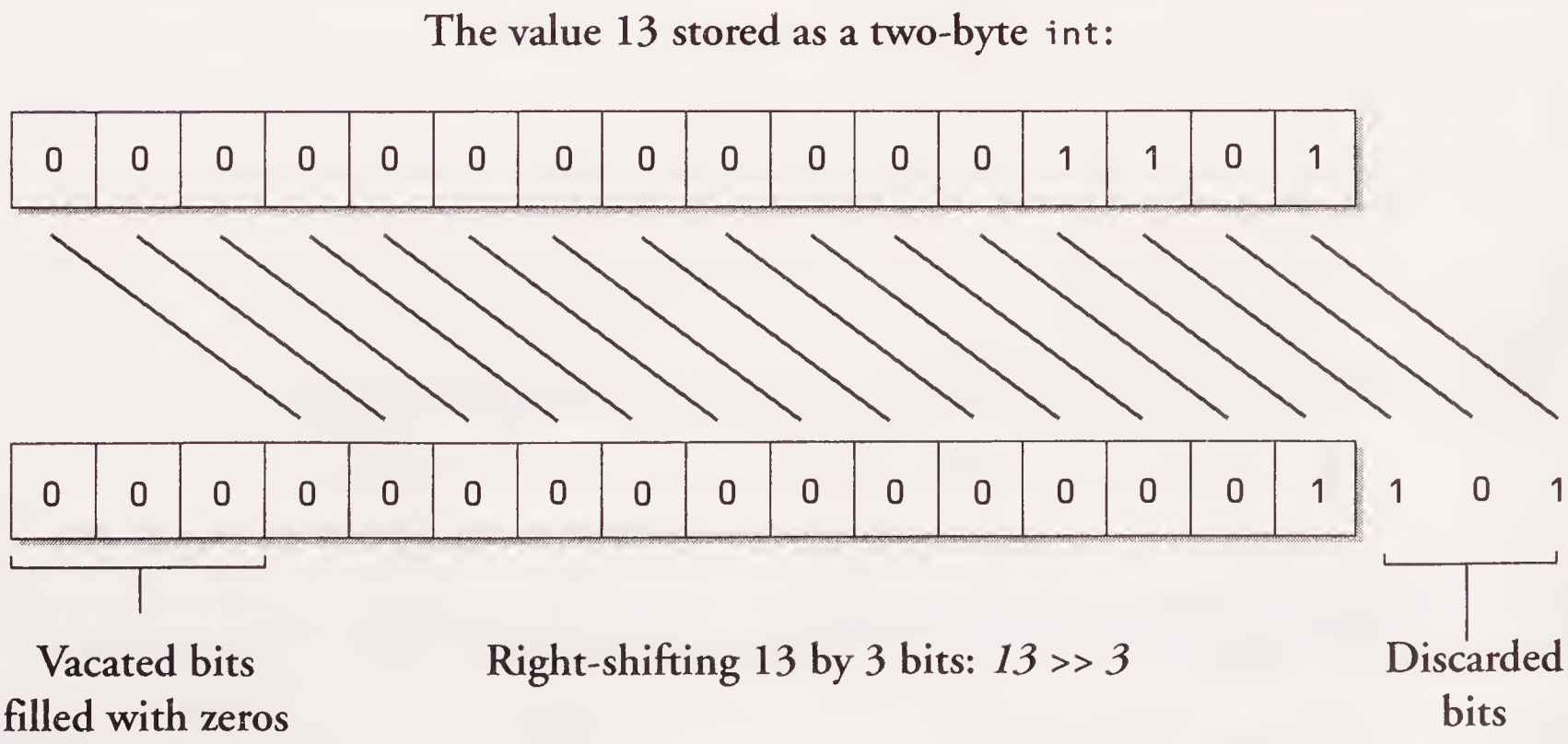


Figure E-2 *The right-shift operator*

The expression *!x* has the value *0*. To see the value of *~x*, write it in binary form: 00000011. Then convert each 0 to 1, each 1 to 0; this produces the value 11111100, or in base 10, the value 252. See Figure E-3.

The bitwise OR operator (|) combines two integer values to produce a new integer value. Each bit in the new value is set to 1 if one or the other, or both, of the corresponding bits in the original values is set to 1. If both corresponding bits are 0, then the final bit is set to 0. See Figure E-4.

Table E-1 summarizes how the | operator combines bits.

TABLE E-1 Value of *b1* | *b2*

BIT VALUES	B1 = 0	B1 = 1
b2 = 0	0	1
b2 = 1	1	1

The bitwise XOR operator (^) combines two integer values to produce a new integer value. Each bit in the new value is set to 1 if one or the other, but not both, of the corresponding bits in the original values is set to 1. If both corresponding bits are 0 or both are 1, then the final bit is set to 0. See Figure E-5.

Table E-2 summarizes how the ^ operator combines bits.

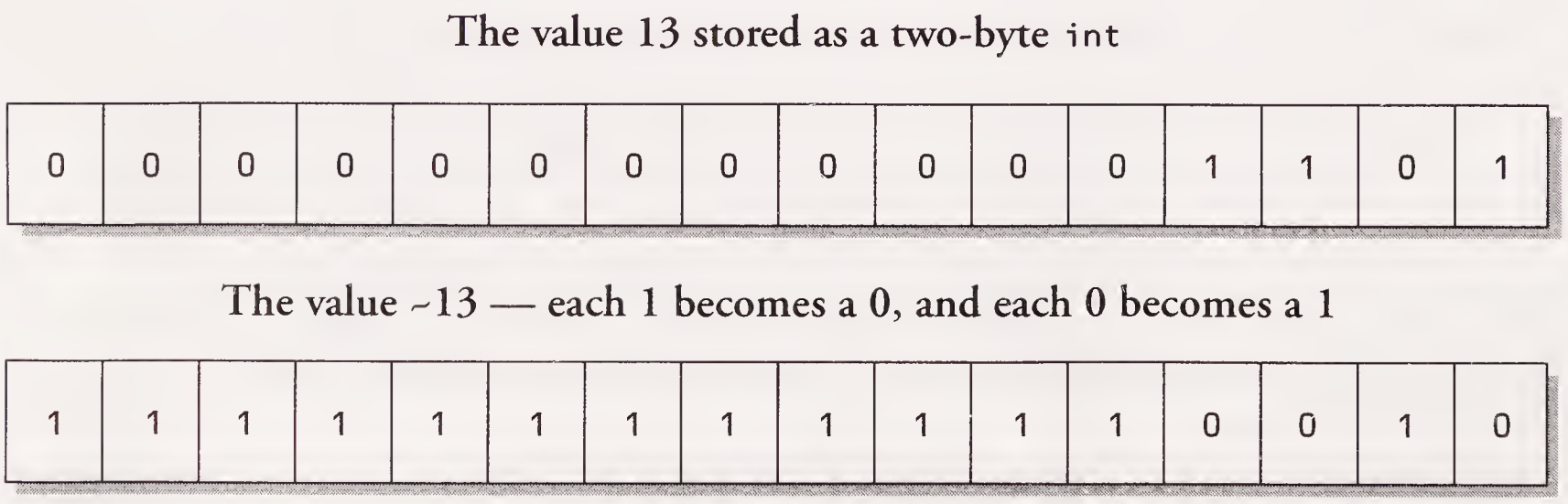


Figure E-3 *The bitwise negation operator*

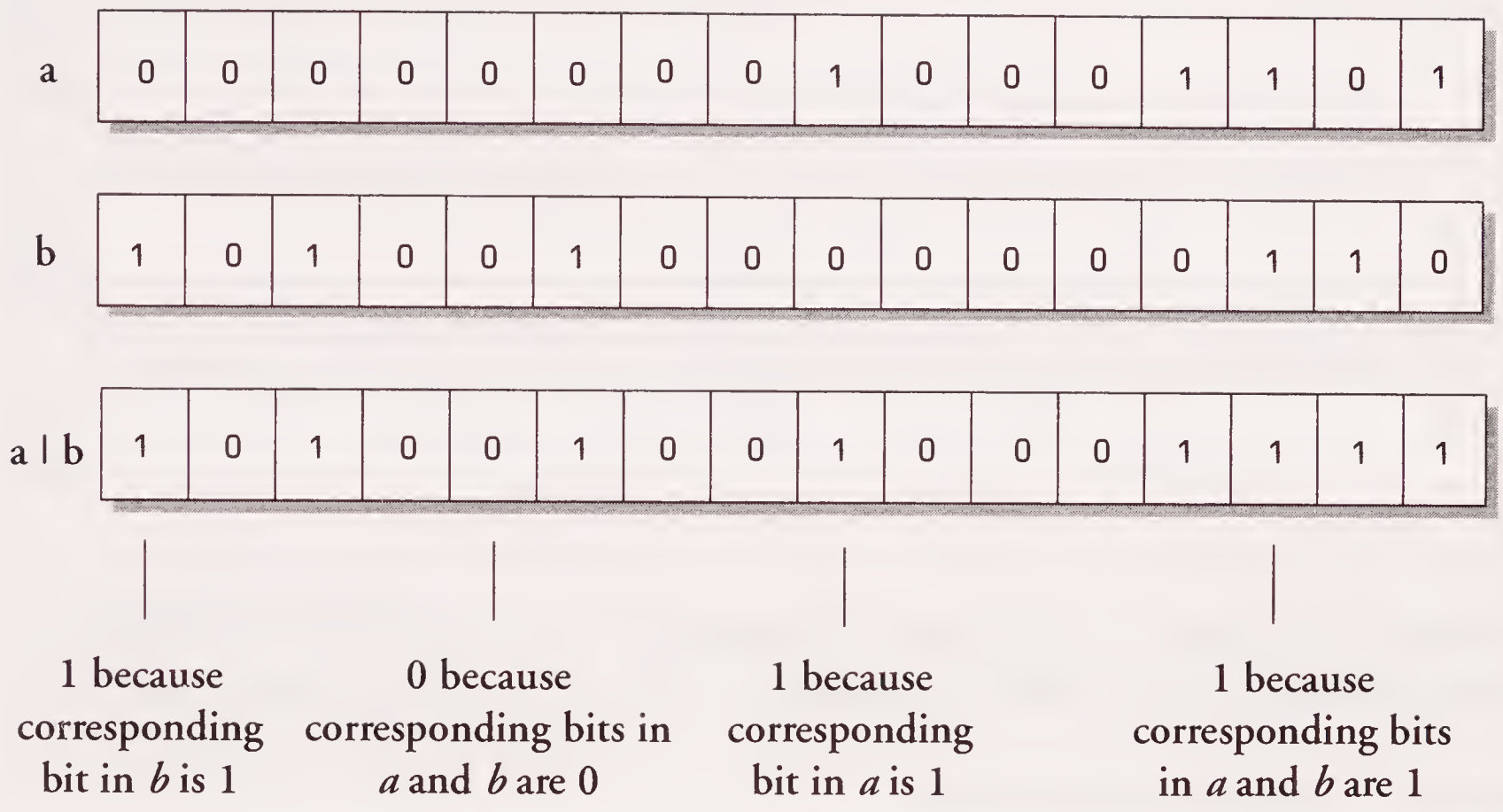


Figure E-4 *The bitwise OR operator*

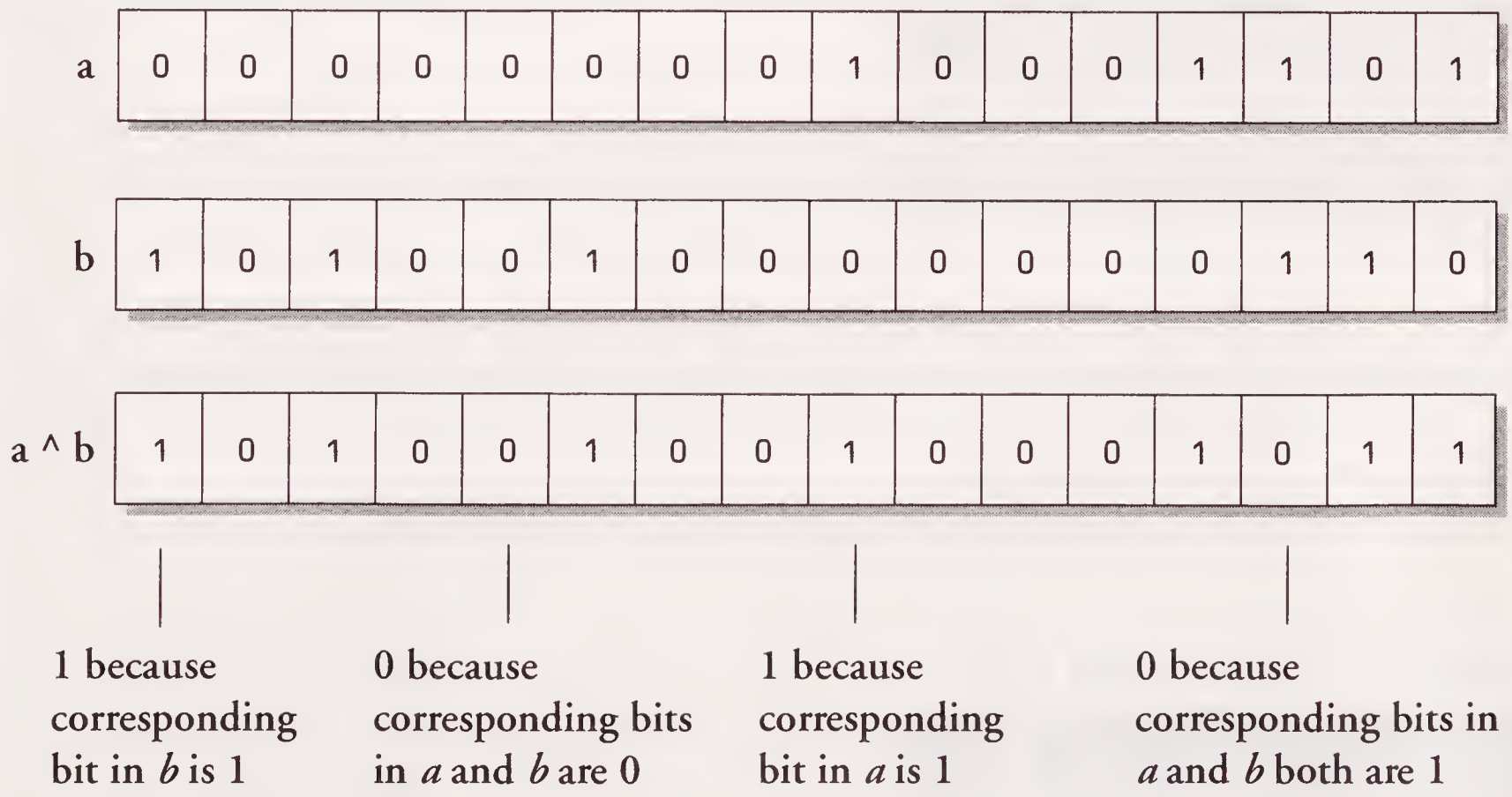


Figure E-5 *The bitwise XOR operator*

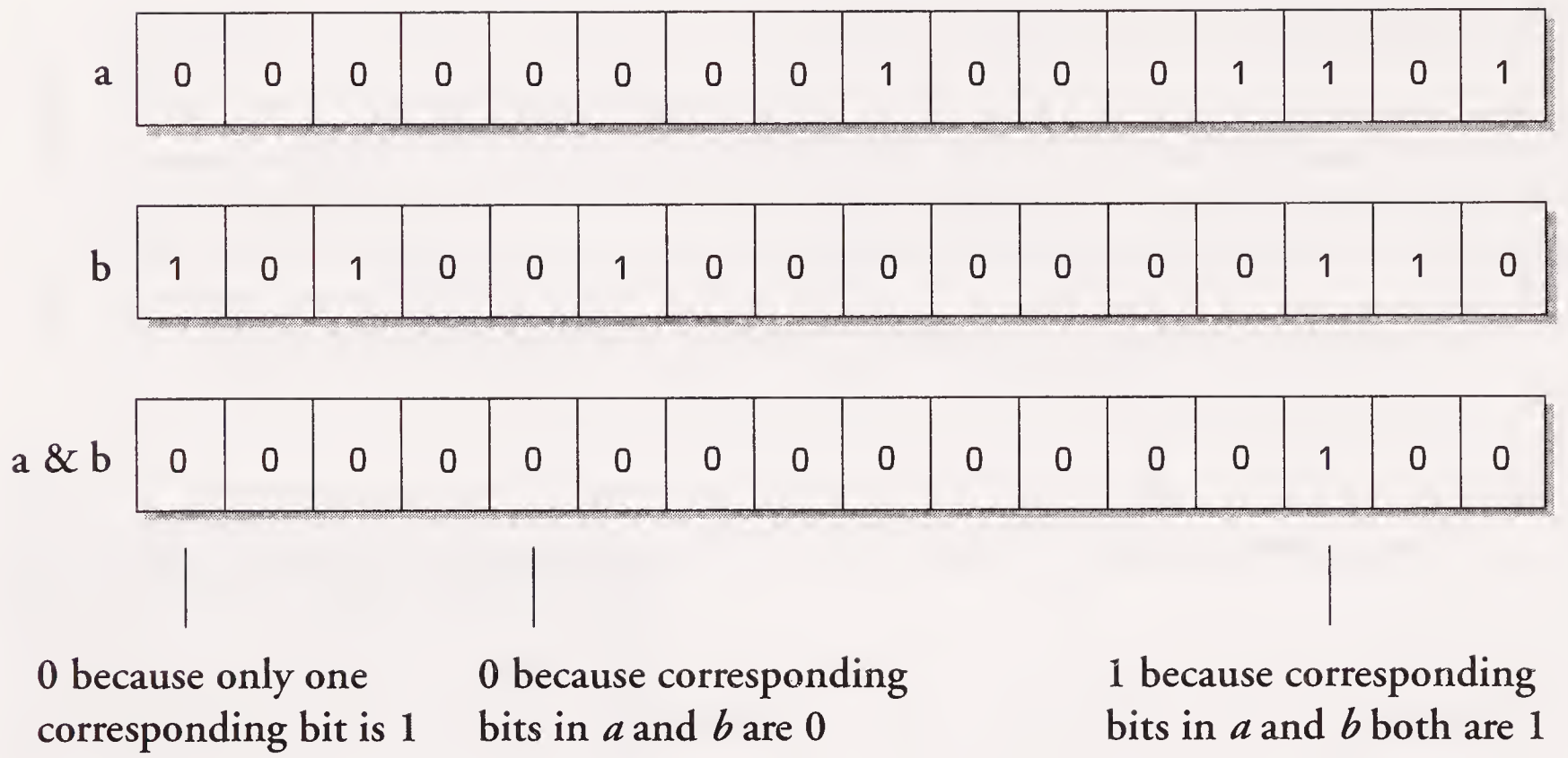


Figure E-6 *The bitwise AND operator*

TABLE E-2 Value of *b1 ^ b2*

BIT VALUES	B1 = 0	B1 = 1
b2 = 0	0	1
b2 = 1	1	0

The bitwise AND operator (&) combines two integer values to produce a new integer value. Each bit in the new value is set to 1 only if both of the corresponding bits in the original values are set to 1. If either or both corresponding bits are 0, then the final bit is set to 0. See Figure E-6.

Table E-3 summarizes how the *&* operator combines bits.

TABLE E-3 Value of *b1 & b2*

BIT VALUES	B1 = 0	B1 = 1
b2 = 0	0	0
b2 = 1	0	1

A FEW COMMON BITWISE TECHNIQUES

Often, controlling hardware involves turning particular bits on or off or checking their status. The bitwise operators provide the means to perform such actions. We'll go through the methods quickly.

In the following, *lottabits* represents a general value, and *bit* represents the value corresponding to a particular bit. Bits are numbered from right to left, beginning with bit 0, so the value corresponding to bit position n is 2^n. For example, an integer with only bit number 3 set to 1 has the value 2^3, or 8. In general, each individual bit corresponds to a power of two, as described for binary numbers in Appendix A. So *bit* will be a power of two corresponding to a particular bit being set to 1 and all other bits set to 0.

Turning a Bit On

The following two operations each turn on the bit in *lottabits* corresponding to the bit represented by *bit*:

```
lottabits = lottabits | bit;
lottabits |= bit;
```

Each sets the corresponding bit to 1 regardless of the former value of the bit. That's because ORing 1 with either 0 or 1 produces a 1. All other bits in *lottabits* remain unaltered. That's because ORing 0 with 0 produces a 0, and ORing 0 with 1 produces a 1.

Toggling a Bit

The following two operations each toggle the bit in *lottabits* corresponding to the bit represented by *bit*. That is, they turn the bit on if it was off, and they turn it off if it was on:

```
lottabits = lottabits ^ bit;
lottabits ^= bit;
```

XORing 1 with 0 produces 1, turning an off bit on, and XORing 1 with 1 produces 0, turning an on bit off. All other bits in *lottabits* remain unaltered. That's because XORing 0 with 0 produces a 0, and XORing 0 with 1 produces a 1.

Turning a Bit Off

The following two operations each turn off the bit in *lottabits* corresponding to the bit represented by *bit*:

```
lottabits = lottabits & ~bit;
lottabits &= ~bit;
```

These statements turn the bit off regardless of its prior state. First, the operator *~bit* produces an integer with all its bits set to 1 *except* the bit that originally was set to 1; that bit becomes a 0. ANDing a 0 with any bit results in 0, thus turning that bit off. All other bits in *lottabits* are unchanged. That's because ANDing a 1 with any bit produces the value that bit had before.

Testing a Bit Value

Suppose we wish to determine whether the bit corresponding to *bit* is set to 1 in *lottabits*. The following test does not necessarily work:

```
if (lottabits == bits)   // no good
```

That's because even if the corresponding bit in *lottabits* is set to 1, so might other bits be set to 1. The equality above is true only when *only* the corresponding bit is 1. The fix is to first AND *lottabits* with *bit*. That produces a value that is 0 in all the other bit positions, because 0 AND any value is 0. Only the bit corresponding to the *bit* value is left unchanged, since 1 AND any value is that value. Thus the proper test is this:

```
if (lottabits & bit == bits)          // testing a bit
```

Member Dereferencing Operators

Before discussing the member dereferencing operators, we need to provide a bit of background. C++ lets you define pointers to members of a class, but the process is not simple. To see what's involved, let's look at an example class. Listing E-1 presents a class definition:



LISTING E-1

```
class example
{
private:
    int feet;
    int inches;
public:
    example();
    example(int ft);
    ~example();
    void show_in();      // display inches member
    example operator+(example &ex);
};
```

Now suppose that you wish to define a pointer to the *inches* member of this class. The following attempt fails:

```
int * pi = &inches;       // not valid C++
```

It fails because *inches* is not type `int`. That's because the type for a class member must also specify the class to which the member belongs. To make the declaration valid, we have to use the scope operator to identify the class for the pointer and for the member:

```
int example::* pi = &example::inches; // valid C++
```

In this declaration the phrase `int example::*` is the type "pointer-to-`int` member of *example* class." The expression `&example::inches` means "the address of the *inches* member of the *example* class."

You can use this form of declaration in member functions or in friend functions. The *pi* pointer acts like a class member in that it must be invoked with a class object. This is where the member dereferencing operators come in. For example, suppose *ex* is an *example* object declared in a member function. To access the *inches* member of *ex*, you can use the standard `ex.inches` notation. But you can also use the .* operator with the *pi* pointer:

```
cout << ex.inches;        // display the inches member
cout << ex.*pi;           // ditto
```

That is, the . operator accesses a member using the member name, while the .* member accesses a member using a pointer to that member.

Similarly, suppose *px* is a pointer to an *example* object. Then you can use the -> operator to access the *inches* member by name or by using the ->* dereferencing operator to access *inches* via a pointer to a member:

```
px = &ex;                 // px a pointer to an example object
cout << px->inches;       // display the inches member
cout << px->*pi;          // ditto
```

Note that *px* is a pointer to an entire object, while *pi* is a pointer to a class member.

To see how these new operators work in practice, let's use them in a slightly roundabout way of implementing the `operator+()` function. The function adds two objects. One object is an argument to the function. Since it is an object, not a pointer, we can use the .* to access the *inches* member. The other object is the invoking object, which, you recall, is represented by the pointer `this`. Hence we can use the -> operator with it. Listing E-2 shows the result:

LISTING E-2

```
example example::operator+(example &ex)
{
    example sum;

    int example::*pi = &example::inches;
    // point to an inches member of example class

    sum.inches = ex.*pi + this->*pi;
    sum.feet = 12 * sum.inches;
    return sum;
}
```

Here ex.*pi represents the *inches* member of *ex* and `this->*pi` represents the *inches* member of the object to which `this` points. Note that `*pi` is used like a member name.

Listing E-3 provides the rest of the method definitions and a `main()` function that uses the class.

LISTING E-3 memb_pt.cpp

```
// memb_pt.cpp -- dereferencing pointers to class members
#include <iostream.h>

class example
{
private:
    int feet;
    int inches;
public:
    example();
    example(int ft);
    ~example();
    void show_in();
   example operator+(example &ex);
};

example::example()
{
    feet = 0;
    inches = 0;
}

example::example(int ft)
```

```
{
    feet = ft;
    inches = 12 * feet;
}

example::~example()
{
}

void example::show_in()
{
    cout << inches << " inches\n";
}

example example::operator+(example &ex)
{
   example sum;

    int example::*pi = &example::inches;
    // point to an inches member of example class

    sum.inches = ex.*pi + this->*pi;
    sum.feet = 12 * sum.inches;
    return sum;
}

int main(void)
{
    example car(15);
    example van(20);
    example garage;

    garage = car + van;
    car.show_in();
    van.show_in();
    garage.show_in();

    return 0;
}
```

Here is a sample run:

```
180 inches
240 inches
420 inches
```

Enumerations

Appendix F

The C++ `enum` facility provides an alternative means to `const` for creating symbolic constants. It also lets you define new types, but in a fairly restricted fashion. The syntax for using `enum` resembles structure syntax. For instance, consider the following statement:

```
enum spectrum {red, orange, yellow, green, blue, purple, indigo};
```

This statement does two things:

- It makes `spectrum` the name of a new type; `spectrum` is termed an *enumeration*, much as a `struct` variable is called a structure.
- It establishes *red*, *orange*, *yellow*, and so on as symbolic constants for the integer values 0-6. These constants are called *enumerators*.

By default, enumerators are assigned integer values starting with *0* for the first enumerator, *1* for the second enumerator, and so forth. You can override the default by explicitly assigning integer values. We'll show you how later.

Meanwhile, you can then use the enumeration name to declare a variable of that type:

```
spectrum band;
```

An enumeration variable has some special properties, which we'll examine now.

The only valid values that can be assigned to an enumeration variable are the enumerator values used in defining the type. Thus we have the following:

```
band = blue;      // valid, blue is an enumerator
band = 2000;      // invalid, 2000 not an enumerator
```

Thus a *spectrum* variable is limited to just seven possible values.

Only the assignment operator is defined for enumerations. In particular, arithmetic and relation operations are not defined:

```
band = orange;          // valid
++band;                 // not valid
if (band > red)         // not valid
...
```

However, some implementations, including Turbo C++, do not honor this restriction. That makes enumerations a bit more useful but also may make it possible to violate the type limits. For example, if *band* has the value *violet*, or 6, then `++band` increments *band* to *7*, which is not a valid value for a *spectrum* type.

Enumerators are of integer type and can be promoted to type `int`, but `int` types are not automatically converted to the enumeration type:

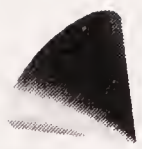
```
int color = blue;       // valid, spectrum type promoted to int
band = 3;               // invalid, int not converted to spectrum
if (blue > 3)           // valid, blue converted to int value of 4
...
```

Note that even though *3* corresponds to the enumerator *green*, assigning *3* to *band* is a type error. But assigning *green* to *band* is fine, for they both are type `spectrum`. Again, some implementations, including Turbo C++, do not enforce this restriction.

As you can see, the rules governing enumerations are fairly restrictive. In practice, they have been used more often as a way of defining related symbolic constants than as a means of defining a new type. For instance, you might use an enumeration to define symbolic constants for a `switch` statement. If you plan to use just the constants and not create variables of the enumeration type, you can omit an enumeration type name:

```
enum {red, orange, yellow, green, blue, purple, indigo};
```

Listing F-1 illustrates using `enum` to define a set of related constants and then using the constants in a `switch`.



LISTING F-1 enum.cpp

```
// enum.cpp -- use enum
#include <iostream.h>

// create named constants for 0 - 6
enum {red, orange, yellow, green, blue, purple, indigo};

  int main(void)
  {
  cout << "Enter color code: ";
  int code;
  cin >> code;
  while (code >= red && code <= indigo)
```

```
    {
            switch (code)
            {
               case red    : cout << "Her lips were red.\n"; break;
               case orange : cout << "Her hair was orange.\n"; break;
               case yellow  : cout << "Her shoes were yellow.\n"; break;
               case green   : cout << "Her nails were green.\n"; break;
               case blue    : cout << "Her sweatsuit was blue.\n"; break;
               case purple  : cout << "Her eyes were purple.\n"; break;
               case indigo : cout << "Her mood was indigo.\n"; break;
            }
            cout << "Enter color code: ";
            cin >> code;
    }
    cout << "Bye\n";
    return 0;
}
```

Here's a sample output:

```
Enter color code: 3
Her nails were green.
Enter color code: 5
Her eyes were purple.
Enter color code: 2
Her shoes were yellow.
Enter color code: 8
Bye
```

Setting Enumerator Values

You can set enumerator values explicitly by using the assignment operator:

```
enum {one = 1, two = 2, four = 4, eight = 8};
```

The assigned values must be integers. You also can define just some of the enumerators explicitly:

```
enum {first, second = 100, third};
```

In this case *first* is *0* by default. Subsequent uninitialized enumerators are larger by one than their predecessors. So *third* would have the value *101*. Finally, you can create more than one enumerator with the same value:

```
enum {zero, null = 0, one, numero_uno = 1};
```

Here, both *zero* and *null* are *0*, and both *one* and *numero_uno* are *1*.

Enumerations and Classes

An enumeration defined within a class has class scope. That means you can use it in the class data members and in the class methods, but not directly outside of the class. However, you can use the scope resolution operator to access class enumerations from without. We did so in Chapter 12, for example, to format the output stream. For instance, we used *ios::left* to indicate left justification. Here, *ios* is the class and *left* is an enumerator defined in that class. Turbo C++, for example, defines the enumerators as follows:

```
// formatting flags as defined in the ios class (Turbo C++)
    enum   {
         skipws     = 0x0001,  // skip whitespace on input
         left       = 0x0002,  // left-adjust output
         right      = 0x0004,  // right-adjust output
         internal   = 0x0008,  // padding after sign or base indicator
         dec        = 0x0010,  // decimal conversion
         oct        = 0x0020,  // octal conversion
         hex        = 0x0040,  // hexidecimal conversion
         showbase   = 0x0080,  // use base indicator on output
         showpoint  = 0x0100,  // force decimal point (floating output)
         uppercase  = 0x0200,  // upper-case hex output
         showpos    = 0x0400,  // add '+' to positive integers
         scientific= 0x0800,  // use 1.2345E2 floating notation
         fixed      = 0x1000,  // use 123.45 floating notation
         unitbuf    = 0x2000,  // flush all streams after insertion
         stdio      = 0x4000   // flush stdout, stderr after insertion
         };
```

Here, the enumeration defined each enumerator explicitly.

Answers to Review Questions

Appendix G

Chapter 2

1. They are called functions.
2. It causes the contents of the `iostream.h` file to be substituted for this directive before final compilation.
3. `cout << "Hello, world\n";`
4. `int cheeses;`
5. `cheeses = 32;`
6. `cin >> cheeses;`
7. `cout << "We have " << cheeses << " varieties of cheese\n";`
8. It tells us that the function `froop()` expects to be called with one argument, which will be type `double`, and that the function will return a type `int` value.
9. You don't use `return` in a function when the function has return type `void`.
10.
```
#include <iostream.h>
int main(void)
{
     int furlongs;
     cout << "Please enter a distance in furlongs:\n";
     cin >> furlongs;
     int yards = 220 * furlongs;
     cout << furlongs << " furlongs are ";
     cout << yards << " yards.\n";
     return 0;
}
```

11.
```
#include <iostream.h>
int fur2yards(int fur);                    // provide prototype
int main(void)
{
 int furlongs;
 cout << "Please enter a distance in furlongs:\n";
 cin >> furlongs;
 int yards = fur2yards(furlongs);
 cout << furlongs << furlongs << " are ";
 cout << yards << " yards.\n";
 return 0;
}

int fur2yards(int fur)                     // provide definition
{
 return 220 * fur;
}
```

CHAPTER 3

1. Having more than one integer type lets you choose the type best suited to a particular need.

2.
```
short rbis = 80;                    // or short int rbis = 80;
unsigned int q = 42110;             // or unsigned q = 42110;
unsigned long ants = 3000000000;
```
   Note: Don't count on `int` being large enough to hold 3000000000.

3. C++ provides no safeguards to keep you from exceeding integer limits.

4. The constant `33L` is type `long`, while the constant `33` is type `int`.

5. The two statements are not really equivalent. Most important, the first statement assigns the letter *A* to `grade` only on a system using the ASCII code, while the second statement also works for other codes. Second, `65` is a type `int` constant, while `'A'` is a type `char` constant.

6. Here are three ways:
```
char c = 88;
cout << c << "\n";    // char type prints as character

cout.put(88);         // put() prints char or int as character

cout << char(88) << "\n";  // type cast value to char
```

7. The answer depends upon how large the two types are. If `long` is four bytes, there is no loss. That's because the largest `long` value would be about 2 billion, which is 10 digits. Since `double` provides at least 15 significant figures, no rounding would be needed.

8. a. `8 * 9 + 2` is `72 + 2` is `74`

   b. `6 * 3 / 4` is `18 / 4 is 4`

   c. `3 / 4 * 6` is `0 * 6` is `0`

   d. `6.0 * 3 / 4` is `18.0 / 4` is `4.5`

   e. `15 % 4` is `3`

9. Either of the following work:

```
int pos = (int) x1 + (int) x2;
int pos = int(x1) + int(x2);
```

10.
```
#include <iostream.h>
int main (void)
{
 int inches;
 const int Inchperfoot = 12;
 cout << "What is your height in inches? ___ \b\b\b";
 cin >> inches;
 cout << "that is" << inches / Inchperfoot << "feet and ";
 cout << inches % Inchperfoot << " inch(es).\n";
 return 0;
}
```

CHAPTER 4

1. a. `char actors[30];`

   b. `short betsie[100];`

   c. `float chuck[13];`

   d. `long double dipsea[64];`

2. `int oddly[5] = {1, 3, 5, 7, 9};`

3. `int even = oddly[0] + oddly[4];`

4. `cout << ideas[1] << "\n";`

5.
```
char lunch[13] = "cheeseburger"; // number of characters + 1
```
or
```
char lunch[] = "cheeseburger";  // let the compiler count elements
```

6.
```
struct fish {
 char kind[20];
 int weight;
 float length;
};
```

7.
```
fish petes =
{
 "trout",
 13,
 12.25
};
```

8.
```
double * pd = &ted;
cout << *pd << "\n";
```

9.
```
float * pf = treakle;  // or = &treakle[0]
cout << pf[0] << " " << pf[9] << "\n";
                        // or use *pf and *(pf + 9)
```

10.
```
unsigned int size;
cout << "Enter a positive integer: ";
cin >> size;
int * dyn = new int [size];
```

11. Yes, it is valid. The expression `"Home of the jolly bytes"` is a string constant, hence it evaluates as the address of the beginning of the string. The `cout` object interprets the address of a `char` as an invitation to print a string, but the type cast `(int *)` converts the address to type pointer-to-`int`, which is then printed as an address. In short, the statement prints the address of the string.

12.
```
struct fish
{
 char kind[20];
 int weight;
 float length;
};

fish * pole = new fish;
cout << "Enter kind of fish: ";
cin >> pole->kind;
```

CHAPTER 5

1. An entry-condition loop evaluates a test expression before entering the body of the loop. If the condition initially is false, the loop never executes its body. An exit-condition loop evaluates a test expression after processing the body of the loop. Thus the loop body is executed once even if the test expression initially is false. The `for` and `while` loops are entry-condition loops, while the `do while` loop is an exit-condition loop.

2. It would print the following:

   ```
   01234
   ```

   Note that the `cout << "\n";` is not part of the loop body (no braces).

3. It would print the following:

   ```
   0369
   12
   ```

4. It would print the following:

   ```
   6
   8
   ```

5. It would print the following:

   ```
   k = 8
   ```

6. It's simplest to use the `*=` operator:

   ```
   for (int num = 1; num <= 64; num *= 2)
    cout << num << " ";
   ```

7. You enclose the statements within paired braces to form a single compound statement, or block.

8. Yes, the statement is valid. The expression `1,024` consists of two expressions—`1` and `024`—joined by a comma operator. The value is the value of the right-hand expression. This is `024`, which is octal for 20, so the declaration assigns the value `20` to `x`.

9. The `cin >> ch` form skips over spaces, newlines, and tabs when it encounters them. The other two forms read these characters.

CHAPTER 6

1. Both versions give the same answers, but the `if else` version is more efficient. Consider what happens, for instance, when `ch` is a space. Version 1, after incrementing `spaces`, then tests to see if the character is a newline. This wastes time because the program already has established that `ch` is a space and hence could not be a newline. Version 2, in the same situation, skips the newline test.
2. Both `++ch` and `ch + 1` have the same numerical value. But `++ch` is type `char` and prints as a character, while `ch + 1`, because it adds a `char` to an `int`, is type `int` and prints as a number.
3. Because the program uses `ch = '$'` instead of `ch == '$'`, the combined input and output looks like this:

```
Hi!
H$i$!$
$Send $10 or $20 now!
S$e$n$d$ $ct1 = 9, ct2 = 9
```

   Each character is converted to the `'$'` character before being printed the second time. Also, the value of the expression `ch = '$'` is the code for the `'$'` character, hence nonzero, hence true, so `ct2` is incremented each time.

4. a. `weight >= 115 && weight < 125`

   b. `ch == 'q' || ch == 'Q'`

   c. `ch % 2 == 0 && ch != 26`

   d. `donation >= 1000 && donation <= 2000 || guest == 1`

   e. `(ch >= 'a' && ch <= 'z') ||(ch >= 'A' && ch <= 'Z')`

5. Not necessarily. For instance, if `x` is `10`, then `!x` is `0` and `!!x` is `1`.

6. `(x < 0)? -x : x`

7.
```
switch (ch)
{
 case 'A':     a_grade++;
               break;
 case 'B':     b_grade++;
               break;
 case 'C':     c_grade++;
```

```
                    break;
 case 'D':          d_grade++;
                    break;
 default  :         f_grade++;
                    break;
}
```

8. If we use integer labels and the user types a noninteger such as `'q'`, the program hangs up because integer input can't process a character. But if we use character labels and the user types an integer such as `5`, character input will process *5* as a character. Then the `default` part of the `switch` can suggest entering another character.

9. Here is one version:

```
int line = 0;
char ch;
while (cin.get(ch) && ch != 'Q')
{
 if (ch == '\n')
        line++;
}
```

10. The easiest approach is to use the `ctype.h` functions.

```
#include <iostream.h>
#include <ctype.h>
int main(void)
{
 char ch;
 while (cin.get(ch))
 {
        if (islower(ch))
                ch = toupper(ch);
        else if (isupper(ch))
                ch = tolower(ch);
        else if (isdigit(ch))
                continue;
        cout << ch;
 }
 return 0;
}
```

Chapter 7

1. The three steps are defining the function, providing a prototype, and calling the function.

2. a. `void igor(void);`

   b. `float tofu(int n);` or `float tofu(int);`

   c. `double mpg(double miles, double gallons);`

   d. `long summation(long harray[], int size);`

   e. `double doctor(char * str);`

   f. `void ofcourse(boss dude);`

   g. `char * plot(map *pmap);`

3.
```
void set_array(int arr[], int size, int value)
{
 for (int i = 0; i < size; i++)
        arr[i] = value;
}
```

4.
```
double biggest (const double foot[], int size)
{
 double max;
 if (size < 1)
 {
        cout << "Invalid array size of " << size << "\n";
        cout << "Returning a value of 0\n";
        return 0;
 }
 else   // not necessary because return terminates program
 {
        max = foot[0];
        for (int i = 1; i < size; i++)
               if (foot[i] > max)
                      max = foot[i];
        return max;
 }
}
```

5. We use the `const` qualifier with pointers to protect the original pointed-to data from being altered. When a program passes a fundamental type such as an `int` or `double`, it passes it by value so that the function works with a copy. Thus the original data is already protected.

6. A string can be stored in a `char` array, it can be represented by a string constant in double quotation marks, and it can be represented by a pointer pointing to the first character of a string.

7.
```
int replace(char * str, char c1, char c2)
{
 int count = 0;
 while (*str)   // while not at end of string
 {
 if (*str == c1)
        {
                *str = c2;
                count++;
        }
        str++;          // advance to next character
 }
 return count;
}
```

8. Because C++ interprets `"pizza"` as the address of its first element, applying the `*` operator yields the value of that first element, which is the character `'p'`. Because C++ interprets `"taco"` as the address of its first element, it interprets `"taco"[2]` as the value of the element two positions down the line, that is, as the character `'c'`. In other words, the string constant acts the same as an array name.

9. To pass it by value, just pass the structure name `glitz`. To pass its address, use the address operator: `&glitz`. Passing by value automatically protects the original data, but it takes time and memory. Passing by address saves time and memory but doesn't protect the original data unless you use the `const` modifier for the function parameter. Also, passing by value means you can use ordinary structure member notation, but passing a pointer means you have to remember to use the indirect membership operator.

10. a.
```
void show_box(box container)
   {
      cout << "Made by " << container. maker << "\n";
      cout << "Height = " << container.height << "\n";
      cout << "Width = " << container.width << "\n";
      cout << "Length = " << container.length << "\n";
      cout << "Volume = " << container.volume << "\n";
   }
```

    b.
```
void set_volume(box * pb)
   {
      pb->volume = pb->height * pb->width * pb->length;
   }
```

CHAPTER 8

1. Short functions that can fit in one line of code.

2. a. `void song(char * name, int times = 1);`

   b. None. Only prototypes contain the default value information.

   c. Yes, providing you retain the default value for *times*:

   ```
   void song(char * name = "O, My Papa", int times = 1);
   ```

3. You can use either the string `"\""` or the character `'"'` to print a quotation mark. The following functions show both methods.

```
#include <iostream.h>
void iquote(int n)
{
 cout << "\"" << n << "\"";
}

void iquote(double x)
{
 cout << '"' << x << '"';
}

void iquote(const char * str)
{
 cout << "\"" << str << "\"";
}
```

4. a. This function shouldn't alter the structure members, so use the `const` qualifier.

```
void show_box(const box & container)
{
   cout << "Made by " << container. maker << "\n";
   cout << "Height = " << container.height << "\n";
   cout << "Width = " << container.width << "\n";
   cout << "Length = " << container.length << "\n";
   cout << "Volume = " << container.volume << "\n";
}
```

   b.
```
void set_volume(box & crate)
{
 crate.volume = crate.height * crate.width * crate.length;
}
```

5. a. This can be done using a default value for the second argument:

```
double mass(double d, double v = 1.0);
```

It can also be done by overloading:

```
double mass(double d, double v);
double mass(double d);
```

b. You can't use a default for the repeat value because you have to provide default values from right to left. You can use overloading:

```
void repeat(int times, char * str);
void repeat(char * str);
```

c. You can use function overloading:

```
int average(int a, int b);
double average(double x, double y);
```

d. You can't do this one because both versions would have the same signature.

e. At least one version must be defined as a `static` function in one of the files:

```
static int average(int a, int b);      // definition in file 1
static double average(int a, int b);   // definition in file 2
```

6 a. `homer` is automatically an automatic variable.

b. `secret` should be defined as an external variable in one file and declared using `extern` in the second file.

c. `topsecret` should be defined as a static external variable by prefacing the external definition with the keyword `static`.

d. `beencalled` should be defined as a local static variable by prefacing a declaration in the function with the keyword `static`.

7. The function should use a default second argument along with a static local variable.

```
// prototype provides default
void printstr(const char * str, int more = 0);
// definition uses static local variable
void printstr(const char * str, int more)
{
 static int times = 0;// number of times function is called
 times++;
 int repeats = (more) ? times : 1;
// if more is nonzero, repeats = times; otherwise repeats = 1
```

```
    for (int i = 0; i < repeats; i++)
            cout << str << "\n";
}
```

8.
```
double calculate(double x, double y, double (*pf)(double, double))
{
 return (*pf)(x, y);
}
```

CHAPTER 9

1. A class is a definition of a user-defined type. A class definition specifies how data is to be stored, and it specifies the methods (class member functions) that can be used to access and manipulate that data.

2. A class bundles the data representation (class data members) and the rules for using the data (class methods) into a single unit; this is encapsulation. The class can use `private` visibility (the default) for data members, meaning that the data can be accessed only through the member functions. This is data hiding.

3. The class defines a type, including how it can be used. An object is a variable or other data object, such as that produced by `new`, created and used according to the class definition. The relationship is much the same as that between a standard type and a variable of that type.

4. If you create several objects of a given class, each object comes with storage for its own set of data. But all the objects use the one set of member functions.

5. Note: The program uses `cin.get(char *, int)` instead of `cin >>` to read names because `cin.get()` reads a whole line instead of just one word (see Chapter 4).

```
#include <iostream.h>

// class definition
class bankaccount
{
private:
 char name[40];
 char acctnum[25];
 double balance;
public:
 void set(void);
 void show(void);
```

```
 void deposit(double cash);
 void withdraw(double cash);
};

// member function definitions
void bankaccount::set(void)
{
 cout << "Enter depositor's name: ";
 cin.get(name, sizeof(name)).get();
 cout << "Enter depositor's account number: ";
 cin.get(acctnum, sizeof(acctnum)).get();
 cout << "Enter the starting balance: ";
 cin >> balance;
}

void bankaccount::show(void)
{
 cout << "Depositor: " << name << "   Account Number: ";
 cout << acctnum << "   Balance: " << balance << "\n";
}

void bankaccount::deposit(double cash)
{
 balance += cash;
}

void bankaccount::withdraw(double cash)
{
 balance -= cash;
}

// test program
int main(void)
{
 bankaccount keating;
 keating.set();
 keating.show();
 keating.deposit(2000.00);
 keating.show();
 keating.withdraw(20000000.00);
 keating.show();
}
 keating.withdraw(20000000.00);
 return 0;
}
```

6. A class constructor is called when you create an object of that class or when you explicitly call the constructor. The class destructor is called when the object expires.

7. First, add a class constructor prototype to the class definition:

```
class bankaccount
{
private:
 char name[40];
 char acctnum[25];
 double balance;
public:
 bankaccount(char * client, char * num, double bal = 0.0);
 void set(void);
 void show(void);
 void deposit(double cash);
 void withdraw(double cash);
};
```

Then add the function definition to the methods. Note that you need to include `string.h` in order to use `strcpy()`.

```
bankaccount::bankaccount(char * client, char * num, double bal)
{
 strcpy(name, client);
 strcpy(acctnum, num);
 balance = bal;
}
```

Keep in mind that default arguments go in the prototype, not in the function definition.

8. A default constructor is one with no arguments or else with defaults for all the arguments. Having one enables you to declare objects without initializing them, even if you've already defined an initializing constructor.

9. First, modify the class definition to add new data members and a new private method that calculates the slugging average. Note: We'll have the user supply total hits, doubles, triples, and home runs. The number of singles can be calculated from that data, so the user needn't supply that value.

```
// softbl3a.h -- header file for softball class
class softball
{
private:
 char firstname[15];
 char lastname[15];
 unsigned atbats;
 unsigned hits;
 unsigned doubles;          // new
```

```
 unsigned triples;           // new
 unsigned homeruns;          // new
 float slugging;             // new
 unsigned rbis;
 float ave;
 float calc_ave(void);
 float calc_slug(void);      // new
public:
 softball(char * fn, char * ln, unsigned ab = 0,
    unsigned h = 0, unsigned db = 0, unsigned tri = 0,
    unsigned hr = 0, unsigned rbi = 0);
  softball(void);
 ~softball(void);
 void setall(void);
 void update(void);
 void show_stats(void);
 softball & topbat(softball & player); };
```

Next, we have to add a definition for `calc_slug()` to the methods file. Also, we need to modify the constructor function, `setall()`, `update()`, and `show_stats()` so that they handle the new data items. The other functions are unchanged, so we don't show them here. To calculate the slugging average, we use the expression

```
int bases =  hits + doubles + triples * 2 + homeruns * 3;
```

At first, this may look as if we're counting a double as one base instead of two, a triple as two bases instead of three, and a home run as three bases instead of four. But the expression already provides one base credit for every hit, so the rest of the expression provides the extra bases beyond the first for each category of hit.

```
// softbl3a.cpp -- methods for softball class
#include <iostream.h>
#include <string.h>
#include "softbl3a.h"

softball:: softball(char * fn, char * ln, unsigned ab,
        unsigned h, unsigned db, unsigned tri,
        unsigned hr, unsigned rbi)
{
 strcpy(firstname, fn);
 strcpy(lastname, ln);
 atbats = ab;
 hits = h;
 doubles = db;
 triples = tri;
```

```
 homeruns = hr;
 rbis = rbi;
 ave = calc_ave();
 slugging = calc_slug();
}

float softball::calc_slug(void)
{
 if (atbats == 0)
        return 0;
 else
 {
        int bases =  hits + doubles + triples * 2 + homeruns * 3;
        return float(bases) / atbats;
 }
}

void softball::setall(void)
{
 cout << "Enter player's first name: ";
 cin >> firstname;
 cout << "Enter player's last name: ";
 cin >> lastname;
 cout << "Enter number of at bats: ";
 cin >> atbats;
 cout << "Enter number of hits: ";
 cin >> hits;
 cout << "Enter number of doubles: ";
 cin >> doubles;
 cout << "Enter number of triples: ";
 cin >> triples;
 cout << "Enter number of homeruns: ";
 cin >> homeruns;
 cout << "Enter number of RBIs: ";
 cin >> rbis;
 ave = calc_ave();
 slugging = calc_slug();
 cout << "\n";
}

void softball::update(void)
{
 cout << "Enter stats for " << firstname << " ";
 cout << lastname << ":\n";
 cout << "Enter additional number of at bats: ";
 int temp;
 cin >> temp;
```

```
 atbats = atbats + temp;
 cout << "Enter additional number of hits: ";
 cin >> temp;
 hits = hits + temp;
 cout << "Enter additional number of doubles: ";
 cin >> temp;
 doubles = doubles + temp;
 cout << "Enter additional number of triples: ";
 cin >> temp;
 triples = triples + temp;
 cout << "Enter additional number of home runs: ";
 cin >> temp;
 homeruns = homeruns + temp;
 cout << "Enter additional number of RBIs: ";
 cin >> temp;
 rbis = rbis + temp;
 ave = calc_ave();
 slugging = calc_slug();
 cout << "\n";
}

void softball::show_stats(void)
{
 cout << "Player: " << firstname << " " << lastname << "\n";
 cout << "At bats: " << atbats << "   Hits: " << hits;
 cout << "   Doubles: " << doubles << "   Triples: ";
 cout << triples << "\nHome Runs: " << homeruns;
 cout << "   RBIS: " << rbis << "\nBA: " << average;
 cout << ave << "   Slugging Ave: " << slugging << "\n\n";
}
```

10. The `this` pointer is a pointer available to class methods. It points to the object used to invoke the method. Thus `this` is the address of the object, and `*this` represents the object itself.

Chapter 10

1. A member function is part of a class definition and is invoked by a particular object. The member function can access members of the invoking object implicitly, without using the membership operator. A friend function is not part of a class, so it's called as a straight function call. It can't access class members implicitly, so it has to use the membership operator applied to an object passed as an argument.

2. It has to be a friend to access `private` members, but it doesn't have to be a friend to access `public` members.

3. Here's a prototype for the class definition file and a function definition for the methods file:

```
// prototype
diva operator*(double mult);
// definition
diva diva::operator*(double mult)
{
 diva temp(firstname, lastname, range, annual_take);
 temp.annual_take = annual_take * mult;
 return temp;
}
```

4. Here's a prototype for the class definition file and a function definition for the methods file:

```
// prototype
friend diva operator*(double mult, const diva & d);
// definition
diva operator*(double mult, const diva & d)
{
 diva temp(firstname, lastname, range, annual_take);
 temp.annual_take = d.annual_take * mult;
 return temp;
}
```

   Or you can define the function this way:

```
// definition
diva operator*(double mult, const diva & d)
{
 return d * mult;
}
```

5. The following five operators cannot be overloaded: `sizezof  .  .*  :: ?:`

6. These operators must be defined by using a member function.

7. Here is a possible prototype and definition:

```
// prototype
friend ostream & operator<<(ostream & os, const diver & dv);
// definition
ostream & operator<<(ostream & os, const diver & dv)
{
```

```
    os << "The reliable " << dv.firstname << " " << dv.lastname
       << " makes $" << dv.annual_take << " per annum\nand can"
       << " dive to a depth of " << dv.max_depth << " feet.\n";
    return os;
}
```

8. We've allowed for three states:

   `'i'` for `int`

   `'d'` for `double`

   `'s'` for stone, pounds

   However, the *cout* object automatically recognizes types `double` and `int`, so the display function doesn't have to explicitly distinguish between the two. Thus only two states are really needed, but defining three leaves the class open for further development.

```
// stonewt.h -- definition for stonewt class
const int Lbs_per_stn = 14;  // pounds per stone
class stonewt
{
private:
    char state;              // display state

    int stone;               // whole stones
    double pds_left;         // fractional pounds
    double pounds;           // entire weight in pounds
public:
    stonewt(int lbs);        // construct from int pounds
    stonewt(double lbs);     // construct from double pounds
    stonewt(int stn, double lbs); // construct from stone, lbs
    stonewt();               // default constructor
    ~stonewt();
// friend operator
    friend ostream & operator<<(ostream & os, stonewt & sw);
};


// stonewt.cpp -- stonewt class methods
#include <iostream.h>
#include "stonewt.h"

stonewt::stonewt(int lbs)
{
    stone = lbs / Lbs_per_stn;              // integer division
```

```
    pds_left = lbs % Lbs_per_stn;
    pounds = lbs;
    state = 'i';
}

stonewt::stonewt(double lbs)
{
    stone = int (lbs) / Lbs_per_stn;        // int division
    pds_left = int (lbs) % Lbs_per_stn + lbs - int(lbs);
    pounds = lbs;
   state = 'd';
}

stonewt::stonewt(int stn, double lbs)
{
    stone = stn;
    pds_left = lbs;
    pounds =  stn * Lbs_per_stn +lbs;
   state = 's';
}

stonewt::stonewt()
{
    stone = pounds = pds_left = 0;
    state = 's';
}

stonewt::~stonewt()
{
}

ostream & operator<<(ostream & os, stonewt & sw)
{
    if (sw.state == 's')
        os << sw.stone << " stone,"
           << sw.pds_left << " pounds\n";
    else
        os << sw.pounds << " pounds\n";
    return os;
}
```

9. The `mag` member equals the magnitude of the vector, so all the function has to do is return the value of that member. Here's a prototype and function definition:

```
operator double();                  // prototype
vector::operator double()           // definition
```

```
{
 return mag;
}
```

10. We've taken the liberty of adding a function that returns the current number of objects. We made `playerct()` a friend function so that it can be called independently of any one object. Note that you have to use the scope operator with the member name in that function.

```
// softrq10.h -- revised header file for softball class
class softball
{
private:
    static int numplay;    // counts objects
    char * firstname;
    char * lastname;
    unsigned atbats;
    unsigned hits;
    unsigned rbis;
    float ave;
    float calc_ave();
public:
    softball(const softball &player);
    softball(char * fn, char * ln, unsigned ab = 0,
        unsigned h = 0, unsigned rbi = 0);
    softball(void);
    ~softball();
    void setall();
    void update();
    softball& topbat(softball& player);
// overloaded operators
    softball & operator=(softball & player);
// friend function
    friend ostream & operator<<(ostream & os,
         softball & player);
    friend int playerct();
};
```

Here we've just listed the revised and new functions:

```
/ softbal5.cpp -- revised methods for softball class
#include <iostream.h>
#include <string.h>
#include "softrq10.h"

int softball::numplay = 0; // initialize count
```

```
softball:: softball(void)
{
    firstname = NULL;
    lastname = NULL;
    numplay++;
}

// special constructor initializes object to an object
softball:: softball(const softball & player)
{
    firstname = new char[strlen(player.firstname) + 1];
    strcpy(firstname, player.firstname);
    lastname = new char[strlen(player.lastname) + 1];
    strcpy(lastname, player.lastname);
    atbats = player.atbats;
    hits = player.hits;
    rbis = player.rbis;
    ave = calc_ave();
    numplay++;
}

// revised constructor allocates string space
softball:: softball(char * fn, char * ln, unsigned ab,
                    unsigned h, unsigned rbi)
{
    firstname = new char[strlen(fn) + 1];
    strcpy(firstname, fn);
    lastname = new char[strlen(ln) + 1];
    strcpy(lastname, ln);
    atbats = ab;
    hits = h;
    rbis = rbi;
    ave = calc_ave();
    numplay++;
}

softball::~softball()
{
    cout << "Bye, " << firstname << "\n";
    delete firstname;
    delete lastname;
    numplay--;
}

int playerct()
{
```

```
    return softball::numplay;      // must use scope operator
}
```

11. First, when an object of that type expires, the data pointed to by the object's member pointer remains in memory, using space and inaccessible because the pointer has been lost. That can be fixed by having the class destructor delete memory allocated by `new` in the constructor functions. Second, once the destructor deletes such memory, it may wind up trying to delete it twice if a program initialized one such object to another. That's because the default initialization of one object to another copies pointer values but does not copy the pointed-to data, producing two pointers to the same data. The solution is to define a class constructor that causes initialization to copy the pointed-to data. Third, assigning one object to another can produce the same situation of two pointers pointing to the same data. The solution is to overload the assignment operator so that it copies the data, not the pointers.

12. Here is a possible class definition:

```
// strngs1.h -- string class definition
class string
{
private:
 char * str;                        // pointer to string
 int len;                           // length of string
   static int num_strings;          // number of objects
public:
 string(char * s);                  // constructor
 string();                          // default constructor
 string(const string & so);         // copy constructor
 ~string();                         // destructor
 int length();                      // return string length
// operators
 operator char*();                  // type cast to string
 string & operator=(string & so);
 friend ostream & operator<<(ostream & os, string & so);
};
```

And here are some possible implementations:

```
// strngs.cpp -- string class methods
#include <iostream.h>
#include <string.h>
#include "strngs1.h"

// initializing static class member
int string::num_strings = 0;
```

```
// class methods

string::string(char * s)
{
 str = new char[strlen(s) + 1];      // allot storage
 strcpy(str, s);                     // initialize pointer
 len = strlen(str);
 num_strings++;                      // set object count
}

string::string()
{
 str = NULL;
 len = strlen(str);
 num_strings++;
}

string::string(const string & so)
{
 str = new char[so.len + 1];         // allot storage
 strcpy(str, so.str);                // initialize pointer
 len = so.len;
 num_strings++;                      // set object count
}

string::~string()
{
 delete str;                         // required
}

int string::length()
{
 return len;
}

string::operator char*()
{
 return str;
}

string & string::operator=(string & so)
{
 delete str;
 len = so.len;
 str = new char[len + 1];
 strcpy(str, so.str);
 return *this;
```

```
}

ostream & operator<<(ostream & os, string & so)
{
 os << so.str;
 return os;
}
```

CHAPTER 11

1. A derived class inherits the `public` and the `protected` members of the base class.

2. You would still be able use the value of an element, but you no longer could assign to it:

```
darray tosanjose[4];
double n = tosanjose[2];      // ok
tosanjose[1] = 68.8;          // no longer valid
```

3. Constructors are called in the order of derivation, with the most ancestral constructor called first. Destructors are called in the opposite order.

4. Yes, every class requires its own constructors. If the derived class adds no new members, the constructor may have an empty body, but it must exist.

5. Only the derived class method is called. It supersedes the base class definition. A base class method is called only if the derived class does not redefine the method.

6. The *bound_dar* class inherits the *darray* assignment operator function. Because the *bound_dar* class uses the same data representation as the *darray* class, it can use the *darray* assignment operator.

7. Yes, you can assign an object of a derived class to an object of the base class. Any data members new to the derived type are not passed to the base type, however. Assignment in the opposite direction (base to derived) is possible only if the derived class defines a conversion operator, which is a constructor having a reference to the base type as its sole argument.

8. A function taking a base class object as an argument can take a derived class object as an argument because passing by value is equivalent to initializing the function argument to the passed value, and C++ allows you to assign a derived type to its base type.

9. It can do so because C++ allows a reference to a base type to refer to any type derived from that base.

10. If `head()` is a regular function, then `ph->head()` invokes `corporation::head()`. If `head()` is a virtual function, then `ph->head()` invokes `department::head()`.

11. Passing an object by reference instead of by value enables the function to avail itself of virtual functions. Also, passing an object by reference instead of value may use less memory and time, particularly for large objects. The main advantage of passing by value is that it protects the original data, but you can accomplish the same end by passing the reference as a `const` type.

12. The constructor for the member class is called first.

13. A class based on multiple inheritance inherits the class data members and the class methods of its ancestors, but a class with member classes does not make the data members and methods of the member classes available publicly.

14. If two lines of inheritance for a class share a common ancestor, the class winds up with two copies of the ancestor's members. Making the ancestor class a virtual base class to its immediate descendants solves that problem.

15. Here's a sample class definition:

```
// pers_age.h -- person_age class

#ifndef __PERS_AGE__
#define __PERS_AGE__

#include "person.h"

class person_age: public person
{
protected:
 double age;
public:
 person_age();
 person_age(char * s1, char * s2, double a = 0.0);
 person_age(person & p, double a = 0);
 void set_age(double a);
 friend ostream & operator<<(ostream & os, person_age & p);
};

#endif
```

And here are the methods:

```
/ pers_age.cpp -- person_age methods

#include "pers_age.h"
person_age::person_age()
{
 age = 0;
}


person_age::person_age(char * s1, char * s2, double a)
 : person(s1, s2), age(a)
{
}

person_age::person_age(person & p, double a) : person(p),
age(a)
{
}

void person_age::set_age(double a)
{
 age = a;
}


ostream & operator<<(ostream & os, person_age & p)
{
 os << person(p);
 os << "; age = " << p.age;
 return os;
}
```

CHAPTER 12

1. The `iostream.h` file defines the classes, constants, and manipulators used to manage input and output. These objects manage the streams and buffers used to handle I/O. The file also creates standard objects (`cin`, `cout`, `cerr`, and `clog`) used to handle the standard input and output streams connected to every program.

2. Keyboard entry generates a series of characters. Typing `121` generates three characters, each represented by a one-byte binary code. If the value is to be

stored as type `int`, these three characters have to be converted to a single binary representation of the value `121`.

3. By default, both the standard output and the standard error send output to the standard output device, typically a monitor. If you have the operating system redirect output to a file, however, the standard output connects to the file instead of to the screen, but the standard error continues to be connected to the screen.

4. The *ostream* class defines a version of the `operator<<()` function for each basic C++ type. The compiler interprets an expression like

   ```
   cout << spot
   ```

   as the following:

   ```
   cout.operator<<(spot)
   ```

   It then can match this method call to the function prototype having the same argument type.

5. You can concatenate output methods that return type *ostream* `&`. This causes the invoking of a method with an object to return that object. The returned object can then invoke the next method in a sequence.

6.
```
//rq12-6.cpp
#include <iostream.h>
#include <iomanip.h>

int main(void)
{
 cout << "Enter an integer: ";
 int n;
 cin >> n;
 cout << setw(15) << "base ten" << setw(15)
      << "base sixteen" << setw(15) << "base eight" << "\n";
 cout.setf(ios::showbase);
 cout << setw(15) << n << hex << setw(15) << n
      << oct << setw(15) << n << "\n";

 return 0;
}
```

7.
```
//rq12-7.cpp
#include <iostream.h>
#include <iomanip.h>

int main(void)
{
```

```
    char name[20];
    float hourly;
    float hours;

    cout << "Enter your name: ";
    cin.get(name, 20).get();
    cout << "Enter your hourly wages: ";
    cin >> hourly;
    cout << "Enter number of hours worked: ";
    cin >> hours;

    cout.setf(ios::showpoint);
    cout << "First format:\n";
    cout << setw(30) << name << ": $" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";
    cout << "Second format:\n";
    cout.setf(ios::left, ios::adjustfield);
    cout << setw(30) << name << ": $" << setprecision(2)
         << setw(10) << hourly << ":" << setprecision(1)
         << setw(5) << hours << "\n";

    return 0;
}
```

8. Here is the output:

```
ct1 = 5; ct2 = 9
```

   The first part of the program ignores spaces and newlines, the second part doesn't. Note that the second part of the program begins reading at the newline character following the first `q`, and it counts that newline as part of its total.

9. The `ignore()` form falters if the input line exceeds 80 characters. In that case it only skips the first 80 characters.

10.
```
//rq12-10.cpp
#include <iostream.h>

int main(void)
{
 int ct = 0;

 while (cin.peek() != '$')
 {
        ct++;
```

```
        cin.get();
 }

 cout << ct << "\n";

 return 0;
}
```

11.
```
//rq12-11.cpp
#include <fstream.h>
#include <stdlib.h>

int main(int argc,char *argv[])
{
 if (argc != 2)
 {
        cerr << "Usage: " << argv[0] << " filename\n";
        exit(1);
 }

 ofstream fout(argv[1]);
 if (fout.fail())
 {
        cerr << "Tough luck\n";
        exit(2);
 }

 char ch;
 while (cin.get(ch))
        fout << ch;

 fout.close();

 return 0;
}
```

12.
```
//rq12-12.cpp
#include <fstream.h>
#include <stdlib.h>

int main(int argc,char *argv[])
{
 if (argc != 3)
 {
        cerr << "Usage: " << argv[0] << " sourcefile"
            << " targetfile\n";
        exit(1);
```

```
}

ifstream fin(argv[1]);
if (fin.fail())
{
        cerr << "Can't open source file " << argv[1] << "\n";
        exit(2);
}

ofstream fout(argv[2], ios::out | ios::noreplace);
if (fout.fail())
{
        cerr << "Can't open target file " << argv[2] << "\n";
        exit(3);
}

char ch;
while (fin.get(ch))
        fout << ch;

fin.close();
fout.close();
```

```
 return 0;
}
```

Index

& operator, *26*, *303*, *312*
&& operator, *214*, *216-21*
  setting up ranges with, *218-21*
. operator, *427*
.* operator, *427*
! operator, *214*, *221-22*
!= operator, *175*
// operator, *23*, *214-15*
[] operator, *427*, *491*, *497*
() operator, *427*
% operator, *275*
define directive, *60*, *74*
#ifndef directive, *516*
#include facility, *316*
\n, *27*
? operator, *225-28*
?: operator, *427*
:: operator, *330*, *358*, *427*
+ operator, overloading, *415-17*
++ operator, *156*, *167-68*
+= operator, *169*
-- operator, *166*, *167-78*
-lm option, *422*
= operator, *32*, *176*, *427*
== operator, *175*, *176*
* operator, *27*, *122*, *137*, *150*, *387*, *409*
> operator, *175*, *385*
< operator, *175*, *193*
>= operator, *166*, *175*
<= operator, *175*
-> operator, *144*, *151*, *387*, *427*, *455*, *482*
<< operator, *26*, *50*, *620*, *632*, *646*
  defining, *481*
  overloaded, *457*, *480*, *585-88*
  overloading, *427-31*, *475*, *563*
>> operator, *35*, *50*, *606*, *613*, *620*, *646*

A

Access
  of members in dynamic structure, *143-44*
Actual argument, *249*
Addition, *80*
  pointer, *135 fig*
Address
  of function, *339*, *340*
  memory to hold, *126*
  passing structure, *282-84*
  stored by pointers, *125 fig*
Alert character, *71*
Algorithms, *5*
Alternatives, deciding between, *209-11*
Ambiguous conversion, *446*
American National Standards Institute. *See* ANSI.
Ancestor, multiple inheritance with common, *563-72*
Anonymous variable, *311*
ANSI, *9*, *247*, *578*
Append, *628*, *630-32*
Argument, *22*, *36*, *394*
  array name, *286*
  command-line, *624*
  default, *295-97*, *344*
  formal, *249*
  function, *248*, *249-56*
  function with two, *252-56*
  multiple, *250-53*
  passing function as, *340*
  passing object as, *546*
  passing with a stack, *325 fig*
  sending directly to virtual ancestor class, *569*
  type of, *248*
  using arrays as, *259-61*
Argument list, *22*, *344*, *383*, *481*
  function with, *295*
  using different, *298*
Argument passing, *89*, *507*, *509*
Arithmetic, pointer, *138*
Arithmetic operators, *80-91*
Array, *97-102*, *150*
  accessing elements in, *132*
  allocating memory for, *131*
  applying sizeof to name, *136*
  of arrays, *199 fig*
  binding for, *138*
  character, *286*
  creating, *99 fig*
  dynamic, *138*
  equivalence to pointer, *132*
  external, *219*
  filling, *262*

Array (continued)
functions and, *256-69*
initializing, *101-02*
initializing two-dimensional, *200-202*
of integers, *63*
modifying, *264*
nested loops and two dimensional, *198-202*
of objects, *391-93*
and pointer arithmetic, *133-39*
and pointers, *258-59*
protecting, *263-64*
showing, *263-64*
using as argument, *259-61*
using a dynamic, *131-33*
using new to create dynamic, *130-33*
using strings in, *104-06*
Array base class, *488-502*
Array contents, displaying, *500*
Array declaration, *138*
Array functions, *261-67*
Array multiplication, *498*
Array name, *137, 192*
Array name argument, *286*
Array notation, *138, 496*
Array of pointers-to-char, *218*
Arraysize, *98*
Arrow membership operator, *144, 477*
ASCII, *67*
Assembly language, *5*
Assigning, to referred-to variable, *498*
Assignment, *59, 446, 483*
in constructors, *552-53*
conversion on, *86-87*
fixing, *468-75*
to a function, *315-16*
memberwise, *119*
pointer problems in, *464-67*
structure, *118-19*
Assignment definition, *479*
Assignment expressions, *160*
Assignment operator, *32, 176, 427, 531*
combination of, *169*
overloading, *468, 499*
redefining, *524*
Assignment statement, *29, 32, 39, 49, 479*
Associativity, *81-82*
AT&T Version 2.0, *9*
Auto keyword, *323, 324*
Automatic conversions, for classes, *439-47*
Automatic storage, *149-50*
Automatic storage class, *327*
Automatic variables, *149, 151, 321-26, 344*
and stack, *324-26*

B

Bad (), *626*
Bang, *221*
Base class, *487*
array, *488-502*
modifying to meet specific needs, *517*
virtual, *564-72*
Base class method, derived class and, *509*
Base eight, *64, 602*
Base object, assigning to derived object, *535 fig*
Base sixteen, *64, 602*
Base ten, *64, 602*
Base types, and derived types, *532*
BASIC, *5*
Bell Laboratories, *5, 8*
Binary files, *632-38, 647*
Binary operator, *417*
Bits, *56*
Bitwise left-shift operator, *26*
Black boxes, *49*
Blocks, *170-71, 323 fig*
defining, *184*
variable defined inside, *171*
Block scope, *320, 393*
Block statement, *211*
Body
for loop, *159*
loop, *156, 157*
Borland C++, *14*
Bottom-up programming, *7, 267*
Bounds checking, derived class with, *517-21*
Braces, *184, 211*
Brackets, *317*
Branching, limited, *6*
Branching statements, *5, 207*
Break statement, *229, 233-35*
Buffers, *578-83, 646*
flushing, *581, 591-92*
Bytes, *56*

C

C language, *5-7, 10*
Called function, *36*
Calling, *22*
constructor, *372*
Calling function, *36, 243*
Carriage return, *28*
Case label value, *232*
Case-sensitive, *12, 19*
CC command, *13*
Cerr object, *582, 584, 646*
Cfront, *12*
Char, *66-73, 91*
Character array, *286*
Character constant, *103*
Character form, *647*
Character functions, ctype.h library of, *224-25*
Character output, *19*
Characters
displaying, *588*
storing, *66-73*
Cin, *25, 34-35, 50, 581, 605, 606, 608-10, 646*
to control while loop, *281*
input with, *189-91, 605-07*
Cin.get(), *191-93, 195-98, 613 table*
Class constructor, *383, 447, 493*
base, *527*
derived, *527*
new and, *400, 456 fig*
new operator in, *400*
Class conversions, *530-33*
Class data, manipulating, *69*
Class definition, *354, 365, 368-69, 394, 448, 489-500*

changing, *431*
completing, *553-56*
incorporating constructor and destructor into, *373-78*
vector, *428*
Class destructors, *384, 465, 483, 494*
Classes, *7, 37-39, 49, 353-69, 394*
adding state member to, *431-32*
array base, *488-502*
base, *487*
defining derived, *503-05*
derived, *488, 546*
derived with bounds checking, *517-21*
deriving, *502-21*
deriving from derived, *517*
deriving from multiple base classes, *557*
disadvantages of member, *556-57*
dynamic memory and ,*447-80*
friend, *401*
function to access separate classes, *401*
implementing the derived, *506-10*
member, *547-57*
multiple representations and, *414*
and structures, *358*
type conversions for, *399, 447*
using, *364-68, 500-502*
using derived, *511-15*
virtual base, *564-72*
whose constructors use new, *467*
Class inheritance, *487*
Class libraries, *38, 487, 578*
Class member access by pointer operator, *427*
Class member function, *355*
concatenating, *111*
friends versus, *421*
implementing, *358-63, 369*
invoked via pointers, *540*
Class members, *357-58*
accessing, *455*
how functions refer to, *405*
static, *448-57*
visibility of inherited, *504 table*
Class method, *38, 355, 358, 491-94, 505*
accessing with pointers, *480*
applying to object, *363*
defining, *354, 553*
Class object, *383, 573*
converting derived to base, *509*
declaring, *365*
Class properties, combining, *547*
Class-related problems, *475-77*
Class scope, *359, 393, 407*
Class types, converting, *482*
Clearing the bit, *601*
Clear ( ) method, *641*
Client-server model, *368*
Clock ( ) function, *185*
Clog object, *582, 584, 646*
Close ( ) method, *622, 646*
Code
executable, *11*
object, *11*
reusable, *7, 487*
source, *11*
Combination assignment operators, *169*
Command-line processing, *624-26*
Comma operator, *171-74*
Comments, *19, 20, 23-24*
Compatibility, *20, 58, 72, 83, 100, 109, 112, 116, 121, 148*
long double, *255*
static storage class, *326*
Compilation, *12-15*
separate, *316-20*
Compile menu, *14*
Compilers, *5*
function interface to, *246-47*
Compile-time decision, *122*
Compiling, *13, 14, 318 fig*
Compound statements, *170-71*
Concatenate, *36, 611*
class member functions, *111*
input, *607*
output, *587-88*
string, *104*
Conditional operator, *207, 225-28, 235, 427*
Const, *73-74, 165, 263-64, 269 fig*
pointers and, *267-69*
storage classes and, *336-37*
Constants
char, *70-73*
character, *103*
file mode, *628 table*
floating-point, *79*
formatting, *600*
symbolic, *185*
symbolic names for, *73-74*
type of, *66, 336*
Constructor, *394, 494-96, 573*
assignment in, *552-53*
calling, *372*
class, *369-84*
copy, *467, 483, 495*
declaring and defining, *370-71*
default, *378-83, 384, 494-96, 552*
definition, *551-52*
for derived class, *506, 507*
and derivation, *504*
designing, *568*
incorporating into class definition, *373-78*
initialization section, *561*
to initialize array elements, *391*
invoking, *383*
providing a new, *524*
and sequence of calls, *530*
using, *372*
using with new, *372*
Constructor chains, *534 fig*
Continue statement, *233-35*
Controlled access, *401*
Control section, loop, *157*
Conventions, typographic, *15-16*
Conversion functions, *443-47, 476, 482*
Conversions, *532*
ambiguous, *446*
on assignment, *86-87*
automatic for classes, *439-47*
class, *530-33*
in expressions, *88*
in passing arguments, *89*
potential problems with, *87*
type, *86-91*
Copy constants, *467*
Copy constructor, *467, 483, 495*

Copying, objects containing pointers, *464-67*
Cout facility, *19, 21, 25, 50, 584*
  formatting with, *592-605*
  how it operates, *429*
  output with, *25-28, 584-620*
  and printf(), *33-34*
  and variable integer, *33*
Cout object, *581, 646*
Cout.put(), *69-70*
Ctype.h library, *224-25, 235*

D

Darray class, *496-500, 502*
Darray object, *494-95*
Data, *5*
  how to represent, *489*
  memory to hold, *126*
  storing in file, *632*
  strategy for handling stored, *122*
  types of, *31*
Data abstraction, *353-69*
Data form, *7*
Data hiding, *356, 369, 394, 401*
Data members, *355, 531*
Data object, *128*
Data structure, *7*
Debuggers, *15*
Decimal integers, *66*
Decimal-point notation, *75*
Decisions, implementing, *207*
Declaration, *31-32, 59, 207*
  array, *97-98*
  defining, *329*
  forward, *402*
  referencing, *329*
Declaration statement, *29, 39, 49, 162*
  to initialize pointer, *126*
  and variables, *30-32*
Declaration-statement expression, *163*
Declaring, pointers, *137*
Decrement operator, *166, 167-68*
Default arguments, *295-97, 344*
Default behavior, modifying, *614*
Default constructor, *378-83, 384, 494-96, 552*
  redefine, *470*
Default initialization, *467*
Default values, *628*
Defining declaration, *329, 333 fig*
Definition, *32, 329*
  assignment, *479*
  class, *365, 368-69, 394, 448, 489-500*
  friend function, *406*
  function, *241*
Delete operator, *151, 338, 448, 452, 457, 458, 476, 483*
  with classes, *399*
  class objects, *482*
  freeing memory with, *146-49*
  and memory, *150*
  new and, *146-49*
Dereferencing operator, *122, 137, 150, 151, 387*
Derivation
  of class, *502-21*
  private, *503*
  public, *503*
Derived class, *488, 546, 573*
  and base class method, *509*
  with bounds checking, *517-21*
  constructor for, *507*
  implementing, *506-10*
  using, *511-15*
Derived object, assigning to base object, *535*
Derived type, *53, 97*
  array as, *98*
  and base types, *532*
Derived type name, *124*
Destructor, *369-84, 372-73, 452, 464*
  for base class, *544, 546*
  class, *384, 465, 483, 494*
  and derivation, *504*
  incorporating into class definition, *373*
  and sequence of calls, *530*
  special name of, *373*
Destructor method, *394*
DIP switches, *600*
Direction, *411*
Displacement vector, *411, 412 fig*
Division, *80, 82-84*
DOS, *11, 584, 634*
Dot membership operator, *144, 369, 394, 477*
Double, *41, 76, 91, 584*
Do while loop, *6, 187-89, 202*
Dynamic allocation, storage classes and, *338-39*
Dynamic array, *131*
  allocated with new, *489*
  creating and deleting, *461*
  of objects, *463*
  using, *131-33*
  using new to create, *130-33*
Dynamic binding, *131, 138, 543*
Dynamic linking, *546*
Dynamic memory, *338, 399*
  and classes, *447-80*
Dynamic structure, using new to create, *143-45*
Dynamic structure members, identifying, *144 fig*

E

Edit menu, *14*
Elements, accessing in array, *98, 132, 199 fig*
Encapsulation, *355, 369*
Endl manipulator, *591*
End-of-file condition, *193-96*
E notation, *75*
Entry-condition loop, *158, 202*
Enumerated constant, *599*
EOF, *195, 612, 626, 641*
Error stream, standard, *582, 584*
Escape sequences, *71-72*
Executable code, *11*
Executable file, *13*
Exit-condition loop, *187, 202*
Exit values, *46*
Exponent, *75, 76*
Expression, *80, 171*
  assignment, *160*
  conversions in, *88*

logical, *214-23*
relational, *174-80*
and statements, *160-62*
Extension, *12, 19*
External array, *219*
External definition, *116*
External identifiers, *55*
External linkage, *321, 332, 344*
External modifiers, *332-36*
External storage class, *327*
External variables, *116, 327-29, 332-36*
defining, *334*
Extern declaration, *332*
Extern keyword, *334, 337*
Extraction operator, *50, 606, 620, 646*
with cin, *608-10*

F

Factorials, *163, 164*
Fail( ), *626*
Field widths, *595-97, 604*
File-end connection, *578*
File input, *620-45*
File I/O, simple, *621-23*
File menu, *14*
File mode argument, *628*
File mode constants, *628 table*
File modes, *627-45, 647*
File output, *620-45*
File pointer, *647*
checking position of, *640*
Files
binary, *632-38, 647*
moving through, *639*
opening multiple, *623-26*
processing sequentially, *623*
File scope, *321, 344, 393, 407*
Fill characters, *597*
Fill( ) member function, *597*
Fixed-point notation, *602*
Float, *76, 91*
Floating-point
advantages of, *79-80*
assigning to integer, *86*
Floating-point constants, *79*
Floating-point format, *91*
Floating-point numbers, *53, 75-80*
Floating-point precision, setting, *598*
Floating-point types, *76-79*
Floating-point value, *633 fig*
Flushing the buffer, *581, 591-92*
Flush manipulator, *591*
For, vs. while, *183-85*
For loop, *6, 155-74, 202*
parts of, *157-63*
strings with, *166-67*
within a for loop, *198-202*
Formal argument, *249*
reference as, *302*
Formal parameters, *256, 286*
Formatting
with cout, *592-605*
source-code, *28-29*
Formatting constants, *600*
Formatting flags, *602*
For statements, *162*
FORTRAN, *5*
Forward declaration, *402, 404 fig*
Fractional parts, *83*
Free store, *149-50*
Friend class, *401*
Friend function, *401, 403 fig, 481*
members of other classes as, *408*
to overload operator, *428*
to reverse operand order, *420*
Friend function definition, *406*
Friend keyword, *407*
Friends, *399, 401-09, 434*
declaring as function from another class, *409*
designing, *403-09*
operator overloading and, *419-20*
versus class member functions, *421*
Fstream class, *620*
Fstream.h file, *646*
Fstream object, *638*
Function arguments, *248, 249-56*
Function body, *21, 49*
Function call, *39, 49, 245-49*
alternative syntax for, *416*
that pass constants and expressions, *312*
Function call argument, and reference parameter, *312*
Function call operator, *427*
Function call statements, *35-36*
Function code, *317*
Function definition, *21, 41, 241, 286, 292*
friend, *406*
location of, *338*
Function heading, *21, 44-46, 49*
as interface, *22*
Function interface, *49, 256*
Function library, *35, 487*
Function members, *355*
Function overloading, *193, 298-302, 344, 409*
to define second constructor, *378*
Function parameters, references as, *306-09*
Function pointers, *339-44*
Function polymorphism, *298-302, 344*
Function prototype, *41, 241, 243-44, 245-49, 262, 286, 295, 394*
in header file, *316-17*
Function prototype statement, *40*
Functions, *7, 11, 20, 39-49, 241-49*
address of, *339, 340*
with argument list, *295*
and arrays, *256-69*
assigning to, *315-16*
called, *36*
calling, *36, 243*
calling itself, *284-85*
clock( ), *185*
conversion, *443-47, 476, 482*
declaring pointer to, *340-41*
defining, *242-45*
family of, *298*
fill( ) member, *597*
form of, *44*
friend, *401, 403 fig, 481*
inline, *291-95*
isupper( ), *226*
isxdigit( ), *226*
library, *35, 42, 241, 578*
main( ), *21-23*

Functions (continued)
member, *69-70*, *364 fig*, *394*, *403 fig*
obtaining address of, *340*
operator, *410*, *468*
passing as argument, *340*
pointer to, *339-44*
public member, *394*
puts(), *35*
redefining in derived classes, *546*
returning, *243*
returning references, *315*
that return strings, *272-73*
with a return value, *39-43*
seekg(), *639*, *647*
seekp(), *639*, *647*
setf(), *599*
sharing same name, *298*
storage classes and, *337-38*
strcmp(), *178*
strcpy(), *140*, *143*, *151*, *452*, *458*
and strings, *369-73*
strlen(), *105*, *140*, *143*, *151*, *166*, *241*, *452*
and structures, *274-84*
telling about an array, *259 fig*
tolower(), *226*
toupper(), *226*
with two arguments, *252-56*
user-defined, *43-47*
user-defined with a return value, *47-49*
user-supplied, *35*
using pointer to invoke, *342*
virtual, *535-47*
void, *242-43*
Function signature, *298*, *344*
Fundamental types, *53*

G

Gcount(), *617-20*
Get(), *108-13*, *610-13*, *614-16*, *620*, *646*
Getchar(), *195*, *613*
Getline(), *108-13*, *610-13*, *614-16*, *646*
Get(void), *613*
Global search and replace, *60*
Global variable, *330*
Good(), *626*
Greater than or equal to relational operator, *166*

H

Hardware-specific, *9*
Header file, *316-17*
library, *241*
Hexadecimal integer, *66*
Hex manipulator, *601*
High-level language, *5*

I

If else statement, *6*, *207*, *209-11*, *235*
construction of, *212-14*
formatting, *211-12*
and switch, *232*
If statement, *207-14*, *235*
Ifstream class, *620*
Ifstream object, *621*, *628*
Ignore(), *614-16*
Include files, managing, *515-17*
Increment operator, *156*, *167-68*
Index, *98*, *150*, *638*
Index operator, *491*, *497*
Indirect value, *122*
Infinite loop, *183*
Information hiding, *8*
Inheritance, *8*, *408*, *488*, *503*, *529-30*, *573*
class, *487*
extending, *521-35*
multiple, *547*, *557-63*
multiple with common ancestor, *563-72*
Initialization, *43*, *57*, *59*, *479*
of array, *101-02*
in constructors, *552-53*
default, *467*
loop, *156*, *157*
member class, *551-52*
memberwise, *483*
redefined, *469 fig*
responsibility for, *527*
of two-dimensional array, *200-02*
Initialization list, *101*, *551*
Initialization string, checking fit of, *371*
Initialization value, *295*
Initializing, objects to objects, *508-10*
Inline functions, *291-95*, *338*, *362*
versus macros, *294-95*
versus regular functions, *293 fig*
Inline keyword, *292*, *344*
Input, *25*, *577-84*
changing associations for standard, *583*
cin, *189-91*, *605-07*
cin.get(ch), *197*
file, *620-45*
form of single-character, *613-14*
single-character, *610-13*
string, *614-16*
Input redirection, *583*
Input stream, *578*
standard, *646*
In scope, *321*
Insertion operator, *26*, *50*, *620*, *632*, *646*
ostream, *592*
overloaded, *585-88*
Instance variable, *37*
Instances, *355*
Integer, *31*, *53*
array of, *63*
char, *66-73*
cout with, *33*
decimal, *66*
hexadecimal, *66*
int, *56-60*
long, *63*
octal, *66*
overflow behavior for, *63 fig*
pointers and, *127-28*
short, *56-60*, *63*
signed, *55*
suffix on, *66*
unsigned, *55*, *61-62*
Integer constants, *64-66*
Integer types, *55-60*, *91*
exceeding limits with, *62*
Integral promotion, *88*
Interface, function, *22*, *49*

Int integer, *31*, *56-60*, *91*, *584*
Internal justification, *602*
Internal linkage, *321*, *332*, *337*, *344*
Iomanip.h header file, *604-05*, *646*
Ios::adjustfield, *602*
Ios::app, *628*
Ios::ate, *628*
Ios::basefield, *602*
Ios::binary, *628*
Ios class, *581*, *594*, *602*, *646*
Ios::floatfield, *602*
Ios::in, *628*
Ios::nocreate, *628*
Ios::out, *628*
Ios::replace, *628*
Ios::showbase, *600*
Ios::showpoint, *600*
Ios::showpos, *600*
Iostream class, *581*
Iostream.h file, *24-25*, *581-83*, *646*
Ios.trunc, *628*
Ios::uppercase, *600*
Isalnum() function, *226*
Isalpha() function, *226*
Isalpha(ch) function, *224*
Iscntrl() function, *226*
Isdigits() function, *224*, *226*
Is-equal-to operator, *176*
Isgraph() function, *226*
Islower() function, *226*
Isprint() function, *226*
Ispunct() function, *224*, *226*
Isspace() function, *224*, *226*
Istream class, *50*, *191*, *581*, *584*, *606*, *646*
Istream methods, *610-16*, *617-20*
Isupper() function, *226*
Isxdigit() function, *226*

J

Justification, *602*

K

Keyboard, *583*, *605*, *646*
Keyboard input, *581*
Keywords, *46*
  auto, *323*, *324*
  extern, *334*, *337*
  friend, *407*
  inline, *292*, *344*
  private, *356*, *357-58*, *490*
  public, *356*, *357-58*
  register, *326*
  static, *337*
  virtual, *543*, *564*, *572*, *573*

L

Language, *5*, *11*
Language divergence, *9*
Left justification, *602*
Left-shift operator, *585*
Libraries, *8*, *11*
  C function, *487*
  class, *38*, *487*, *578*
  function, *35*, *487*
  math, *422*
  standard C, *10*
  string, *140*, *178*
Library files, *42*
Library functions, *35*, *42*, *241*, *578*
LIFO, *324*
Limits.h header file, *57*, *58*
Line-oriented input, *108-13*
Linkage, *320-31*, *332-36*, *344*
Linking, *12-15*
  Borland C++, *14*
  dynamic, *543*
  static, *543*
  turbo C++, *14*
  UNIX, *13*
  Zortech C++, *14-15*
Local variables, *249*, *251 fig*, *256*, *330*
Logical AND, *214*, *216-21*
Logical expressions, *214-23*
Logical NOT, *214*, *221-22*
Logical operators, *207*, *214-23*, *235*
  precedence of, *222-23*
Logical OR, *214-15*
Long double, *76*, *91*, *255*
Long integer, *56-60*, *63*, *91*, *600*
  assigning to float, *86*
Loop
  do while, *6*, *187-89*, *202*
  entry-condition, *158*, *202*
  exit-condition, *187*, *202*
  for, *6*, *155-74*
  infinite, *183*
  nested and two-dimensional arrays, *198-202*
  parts of for, *157-63*
  terminating early, *262-63*
  and text input, *189-98*
  while, *6*, *181-87*
Loop body, *156*
  for, *159*
Loop counter, *165*
Loop initialization, *156*
Loop test, *156*
Loop update, *156*
Low-level language, *5*
Lvalue, *74*

M

Machine language, *11*
Macintosh, *634*
Macros, inline versus, *294-95*
Magnitude, *411*
Main() function, *21-23*
Make program, *316*
Manipulator, *646*
Mantissa, *75*
Math.h header file, *41*
Math library, *422*
Member
  accessing, *143-44*
  accessing individual, *114*
  adding a new, *522*
  of class, *354*
  as object, *553*
Member classes, *550-51*
  classes with, *547-57*
  disadvantages of, *556-57*
  multiple inheritance versus, *558 fig*
Member function, *69-70*, *364 fig*, *394*, *403 fig*
  accessing objects with, *385 fig*
  calling, *363*
  cin.get(), *195*
  declaring, *365*

Member functions (continued)
to overload, *427*
providing objects to, *384*
public, *394*
regular versus virtual, *547 fig*
using with object, *363*
Member function call, *202*
Member function definitions, *358, 365*
Membership operator, *69, 114, 150, 363, 427*
Memberwise assignment, *119, 531*
Memberwise initialization, *465, 466 fig, 483*
Memory
allocating for an array, *131*
allocating with new, *128-30*
allocation during run time, *447*
dynamic, *338, 399*
freeing, *146*
freeing with delete, *146-49*
improved management of, *457-64*
penalty, *292*
required, *30*
sufficient, *130*
treated as stack, *324-26*
used as temporary storage, *579*
for variables, *338*
Memory position, *121*
Menu, *14*
Messages, *38*
handling series of, *218-21*
Message statements, *39*
Method, defining within class definition, *361-62*
Method definition, *373-78*
Methods, which get called, *529-30*
Methods file, *451*
Mode, adding as third argument, *432-33*
Modifiers, *332-36*
Modulus, taking, *80*
Modulus operator, *84-85, 275, 500*
Monitor, *646*
MS DOS, *583*
Multiple inheritance, *557-63, 565 fig, 573*
with common ancestor, *563-72*
from virtual base class, *567 fig*
versus member classes, *558 fig*
Multiplication, *80, 498*

N

Name
array, *137, 192*
for variables, *54-55*
Negative exponent, *76*
Nested loops, and two-dimensional arrays, *198-202*
Newline character, *27-28, 591, 611, 632, 634*
New operator, *138, 143, 151, 338, 448, 452, 458*
allocating memory with, *128-30, 482, 493*
to allocate an object, *457*
to allocate storage space for name strings, *457*
in class constructor, *400, 456 fig*
with classes, *399*
classes whose constructors use, *476*
to create class objects, *482*
to create dynamic arrays, *461*
to create objects, *463*
and delete, *146-49*
dynamic array allocated with, *489*
and memory, *150*
using constructor with, *372*
using to create dynamic arrays, *130-33*
using to create dynamic structures, *143-45*
Nonexpressions, and statements, *162-63*
Null character, *102, 286, 590*
Null pointer, *130*
assignment message, *466*
Number base, changing, *593-95*
Numbers, pointers and, *127-28*
Numeric data, write() method with, *591*
Numeric types, converting, *585*

O

Object, *7, 37, 49, 355, 364 fig, 394*
accessing with member function, *385 fig*
applying class method to, *363*
array of, *391-93*
assigning one to another, *383*
cerr, *582, 584, 646*
class, *383, 573*
clog, *582, 584, 646*
cout, *581, 646*
create, *369*
darray, *494-95*
data, *128*
fstream, *638*
how functions choose which to use, *404*
ifstream, *621, 628*
Object code, *11, 13*
Object initialization
fixing problems with, *467-68*
pointer problems in, *464-67*
Object instance, *37*
Object members
initialization for classes with, *551-52*
initializing to objects, *508-10*
initializing those created with new, *477*
new to create, *463*
passing by reference, *405*
pointers and references to, *533-35*
predefined, *25*
providing to member function, *384*
represented by references, *541*
using constructor to initialize, *372*
using pointer to, *477-80*
using pointer or reference to identify, *540*
Object message statements, *49*
Object-oriented programming (OOP), *3, 4, 7-8, 351-53*
Object type, *37*
Octal integer, *66*
Offsets, *121*
Ofstream class, *620, 628*
Ofstream object, *621*

Opening, multiple files, *623-26*
Open() method, *628*, , *646*
Operand order, using friend function to reverse, *420*
Operands, *80*
  user-defined, *426*
Operating system, *5*, *46*
Operator, *91*
  arithmetic *80-91*
  arrow membership, *144*, *477*
  assignment, *32*, *176*, *427*, *531*
  binary, *417*
  bitwise left shift, *26*
  that can be overloaded, *427 table*
  conditional, *207*, *225-28*, *235*, *427*
  decrement, *166*, *167-68*
  delete, *151*, *338*, *448*, *452*, *457*, *458*, *476*, *483*
  dereferencing, *122*, *137*, *150*, *151*, *387*
  dot membership, *144*, *369*, *394*, *477*
  extraction, *50*, *606*, *620*, *646*
  increment, *156*, *167-68*
  index, *491*, *497*
  insertion, *26*, *50*, *620*, *632*, *646*
  left-shift, *585*
  logical, *207*, *214-23*, *235*
  membership, *69*, *114*, *150*, *363*, *427*
  modulus, *84-85*, *275*, *500*
  new, *138*, *143*, *151*, *338*, *448*, *452*, *458*
  overloading, *481*
  overloading an overloaded, *417-18*
  redefine, *39*
  reference, *312*
  relational, *175*
  scope resolution, *330*, *409*, *427*, *600*
  sizeof, *57*, *58*, *101*, *105*, *427*, *635*
  subscripting, *427*
  unary, *417*
Operator function, *410*, *468*
  defining special, *481*
Operator *() function, *499*
Operator overloading, *26*, *84*, *399*, *409-39*
  and friends, *419-20*
  vector class with, *410-26*
Operator precedence, *81-82*, *341*, *427*
Operator redefinition, *316*
Ostream class, *37*, *50*, *581*, *584*, *585*, *586*, *588-91*, *646*
Ostream insertion operators, *592*
Ostream methods, *620*
Ostream reference, *429*
Output, *25*, *577-84*
  changing associations for standard, *583*
  character, *19*
  with cout, *25-28*, *584-620*
  file, *620-45*
  redirecting standard, *584*
Output buffer, flushing, *591-92*
Output concatenation, *587-88*
Output redirection, *583*
Output statements, combining, *36*
Output stream, standard, *646*
Out of scope, *321*
Overloaded operators, *496*
  overloading, *417-18*
Overloading
  classes containing pointers, *499*
  operator, *409-39*, *468*, *499*
  restrictions on, *426-27*

P

Parameter, *249*
Parentheses, *341*
Passed values, receiving, *249*
Passing an object, *546*
Passing arguments
  with a stack, *325 fig*
  to base class constructor, *507 fig*
Passing by reference, *306*, *307 fig*
Passing by value, *249-56*, *307 fig*
Passing structures, *274-82*
Peek(), *617-20*
Pointers, *97*, *150*, *449*
  accessing class methods with, *480*
  address stored by, *125 fig*
  applying sizeof to, *136*
  arrays and, *258-59*
  assigning values to, *137*
  changing the value of, *268*
  checking positino of file, *640*
  class member functions invoked by, *540*
  and const, *267-69*
  copying objects containing, *464*
  declaration statement to initialize, *126*
  declaring, *124-27*, *137*
  declaring to function, *340-41*
  decrementing, *168*
  dereferencing, *137*
  equivalence to arrays, *132*
  file, *640*, *647*
  and free store, *120-50*
  to functions, *339-44*
  to identify an object, *540*
  incrementing, *168*
  initializing, *124-27*
  null, *130*
  and numbers, *127-28*
  to objects, *477-80*, *533-35*
  and pointed-to value, *137*
  and pointer arithmetic, *133-39*
  that points to a constant, *267*
  problems with in object initialization, *464-67*
  and references, *304*
  storing addresses, *125 fig*
  and strings, *140-43*
  using to invoke function, *342*
Pointer addition, *135 fig*
Pointer argument, *546*
Pointer arithmetic, *133-39*
Pointer dereferencing operator, *151*
Pointer membership operator, *482*
Pointer notation, *138*
Pointer-to-const, *268*, *269 fig*
Pointer-to-member operator, *427*
Pointer to a string, *286*
Pointer types, *607*
Polar coordinates, *278*, *279 fig*, *415*
Polymorphism, *8*, *409*
  function, *298-302*, *344*, *546*
Portability, *9-10*

Postfix, *167*
Precedence
  of logical operators, *222-23*
  operator, *341*, *427*
  rules of, *82*
Precision, setting, *604*
Precision() member function, *598*
Predefined object, *25*
Prefix, *167*
Preprocessor, *24-25*
  symbolic constants in, *60*
Preprocessor directive, *60*
Printf(), *20*
  cout and, *33-34*
Private data members, *394*
Private derivation, *503*
Private keyword, *356*, *357-58*, *490*
Private members, *503*
  accessing, *359*, *401*, *408*
Procedural language, *3*, *5*
Procedural programming, *351-53*
Program, mechanics of creating, *11-15*
Program-end connection, *578*
Program management, *515*
Programming
  bottom-up, *7*
  object-oriented, *3*, *4*, *7-8*
  spaghetti, *5*
  structured, *4*, *6*
Project menu, *14*
Protected keyword, *356*, *490*
Protected members, *503*, *546*, *573*
Protected section, of class definition, *490*
Prototype, *294*, *344*
  ANSI C, *247*
  features of, *248-49*
  function, *41*, *241*, *243-44*, *245-49*, *262*, *286*, *295*, *394*
  syntax of, *247*
  why C++ requires them, *246-47*
Prototyping, function, *245-49*
Public derivation, *503*
Public keyword, *356*, *357-58*
Public member functions, *394*
Public members, *503*
Public section, of class definition, *490*
Punctuation, bad, *184*
Putback(), *617-20*
Putchar(), *195*
Put() method, *588*, *646*
Puts() function, *35*

Q

Quotation marks
  double, *70*, *103*, *317*
  single, *68*, *103*

R

Random access, *638-45*, *647*
Ranges, setting up with &&, *218-21*
Range tests, *220-21*
Rdstate(), *626*
Read, a file, *621*
Read() method, *617-20*, *647*
Records, *638*
Rectangular coordinates, *277*, *415*
Recursion, *284-86*, *287*
Redirection, *193*, *583-84*, *620*
Reference, *192*, *404*
  as function parameters, *306-09*
  function returning, *315*
  to identify an object, *540*
  to an object, *468*, *533-35*
  objects represented by, *541*
  passing objects by reference, *405*
  properties of, *309-12*
  returning, *384*
  as return value, *314-15*
  using with structure, *312-16*
Reference argument, *546*, *606*
Reference declarations, *32*, *329*, *333 fig*
Reference operator, *312*
Reference parameter, and function call argument, *312*
Reference variables, *302-16*
Register keyword, *326*
Relational expressions, *174-80*, *235*
  joining with AND, *220-21*
Relational operators, *175*, *385*
Remainder, *84*
Repetitive tasks, performing, *155*
Representations, multiple, *414*
Return a reference, *384*
Returning function, *243*
Returning structures, *274-82*
Return statement, *21*, *47-48*, *49*, *162*
Return type, *22*
Return value, *22*, *242-45*, *248*
  function with, *39-43*
  reference as, *314-15*
Reusable code, *7*, *521*
Right-justification, *595*, *602*
Ritchie, Dennis, *5*
Run menu, *14*
Run-time decision, *122*, *543*

S

Scaling factor, *75*
Scientific notation, *602*
Scope, *320-21*, *323 fig*, *344*, *358*, *369*, *407*, *451*
  block, *320*, *393*
  class, *359*, *393*, *407*
  file, *321*, *344*, *393*, *407*
Scope resolution operator, *330*, *409*, *427*, *600*
Scoping rules, *74*, *456*
Screen, *583*
Seek-dir argument, *640*
Seekg() function, *639*, *647*
Seekp() function, *639*, *647*
Segment value, *121*
Semicolon, *28*, *185*, *247*
  statements and, *21-22*
Sending a message, *363*
Sentinel character, *190*
Setf() function, *599*
Setf() method, *600-604*
Setf() prototype, *601*
Setprecision(), *604*
Setw(), *604*
Short integer, *56-60*, *63*, *91*
  assigning to long, *86*
Showpoint, *599*
Side effects, *161*
Signature, *418*
Signed char, *73*
Signed types, *55*, *57*
Significant figures, *76*, *78*
Simula *67*, *8*

Single-character input, *610-13, 646*
  form of, *613-14*
Size, of array, *192*
Sizeof operator, *57, 58, 101, 105, 427, 635*
  and array name, *136, 138*
  and pointer, *136*
Source code, *11-12, 29, 317*
Space, *28*
Spaghetti programming, *5*
Stack
  automatic variables and, *324-26*
  passing arguments with, *325 fig*
Standard Input/Output package, *578*
Standard input stream, *581*
Standard output stream, *581, 584*
Standards, *9-10*
State member, *431-32*
Statement, *21, 29-34*
  assignment, *29, 32, 39, 49, 479*
  block, *211*
  body of for, *160*
  branching, *5, 207*
  break, *229, 233-35*
  C++, *34-39*
  compound, *170-71*
  continue, *233-35*
  declaration, *29, 39, 49, 162*
  expressions and, *160-62*
  for, *162*
  function call, *35-36*
  if, *207-14, 235*
  if else, *6, 207, 209-11, 235*
  message, *39*
  nonexpressions and, *162-63*
  object message, *49*
  output, *36*
  return, *21, 47-48, 49, 162*
  and semicolons, *21-22*
  summary of, *39*
  switch, *207, 228-32, 235*
Statement portion, *207*
Static, *100*
Static binding, *131, 138*
Static class members, *448-57*
Static external storage class, *327*
Static keyword, *337*
Static linking, *543*
Static modifier, *331-36*
Static storage, *149-50*
Static storage class, *326-30, 449*
Static variable, *151, 344*
Step size, user-selected, *165*
Storage classes, *320-39, 448*
  automatic, *149-50, 327*
  and const, *336-37*
  default, *336*
  and dynamic allocation, *338-39*
  external, *327*
  and functions, *337-38*
  run-time, *400*
  static, *149-50, 326-30, 337, 449*
  static external, *327*
Storage location, *30*
Strcmp() function, *178*
Strcpy() function, *140, 143, 151, 452, 458*
Stream, *26, 578-83, 646*
  associating file with, *646*
Streambuf class, *581*
Stream checking, *626-27*
Stream-state methods, *626 table*
String, *25, 97, 102-13, 151*
  comparing, *178-80*
  cout with, *33*
  displaying, *588*
  with for loop, *166-67*
  functions and, *269-73*
  functions that return, *272-73*
  locating end of, *107*
  passing as argument to a function, *270*
  pointer and, *140-43*
  problems with, *106-13*
  reversing, *173 fig*
  setting size of new, *297*
  using in an array, *104-06*
  ways to represent, *286*
String concatenations, *104*
String constant, *103, 286*
String.h, *241*
String input, *614-16, 646*
String library, *140, 178*
String literal, *103*
Strlen() function, *105, 140, 143, 151, 166, 241, 452*
Str member, *452*
Strncpy() function, *371*
Stroustrup, Bjarne, *8, 12, 400*
Structure addresses, passing, *282-84*
Structure assignment, *118-19*
Structure data object, *113*
Structured programming, *4, 6*
Structure member, *114*
Structures, *97, 113-20, 150, 286*
  assignment, *118-19*
  classes and, *358*
  creating with no type name, *119-20*
  functions and, *274-84*
  passing and returning, *274-82*
  properties of, *118-20*
  returning, *274-82*
  using references with, *312-16*
Structure template, *113, 115 fig, 117 fig, 150*
  where to place, *116*
Structure variable, *113, 274*
Subscript, *98, 150*
  accessing array elements with, *199 fig*
Subscripting operator, *427*
Subtraction, *80*
Suffix
  on integer, *6*
  proper, *11*
Swapping functions, *306*
Switch statement, *207, 228-32, 235*
  and if else, *232*
Symbolic constants, *59, 185, 195*
  in preprocessor, *60*
Syntax, prototype, *247*
Syntax rules, *426*

T

Tab, *28*
TCK_CLK, *185*
Tellg() method, *640, 647*
Tellp() method, *640, 647*
Temp, *217*
Templates, in header files, *316*
Temporary variable, *217-18*
Terminating character, built-in, *270*
Terminating null character, *587*
Termination character, *614*
Terminator, *21*

Test
loop, *156, 157*
for more than one condition, *214*
Test-condition expression, *181, 207, 208, 209*
Test-expression, *157*
Text editor, *11*
Text files, *634, 647*
Text input, loops and, *189-98*
This pointer, *384-91, 394*
Tilde, *373*
Time delay, *185*
Time.h header file, *185, 186*
Tokens, *28, 30 fig*
Tolower() function, *226*
Top-down design, *6*
Toupper() function, *226*
Trailing zeros, printing, *599*
Translator program, *24*
Truncate, *628*
Truth value, reversing, *221*
Two-dimensional array
initializing, *200-202*
nested loops and, *198-202*
Type aliases, *186-87*
Type casts, *89-91, 563*
for classes, *439-47*
Type conversions, *86-91*
applying automatically, *446-47*
automatic, *249, 399*
for classes, *447*
Typedef, *187*
Type name, derived, *124*
Types, *353-54*
basic, *606*
of data, *31*
derived, *53*
fundamental, *53*

U

Unary operator, *417*
UNIX, *578, 583, 584, 634, 635*
C++ on, *11*
compiling on, *318 fig*
compiling and linking, *13*
development of, *5*
Unsigned char, *73*
Unsigned types, *55, 61-62*
Update, loop, *156, 157*
Update-expression, *158, 165*
User-defined functions, *43-47*
User-defined types, *317*
operator overloading with, *409*
User-supplied functions, *35*

V

Variable declarations, *32*
Variables, *29, 31*
anonymous, *311*
assigning to referred-to, *498*
automatic, *149, 151, 321-26, 344*
creating reference, *303-06*
declaration statements and, *30-32*
declared with function, *249*
declaring, *54, 355*
defined inside a bloc, *171*
defining external, *334*
external, *116, 327-29, 332-36*
global, *330*
initializing in declaration, *329*
instance, *37*
local, *249, 251 fig, 256, 330*
names for, *54-55*
reference, *302-16*
register, *326*
simple, *53-73*
static, *151, 344*
structure, *113, 274*
temporary, *217-18*
undefined, *60*
Vector, *411*
adding in rectangular coordinates, *415*
describing displacement with, *412 fig*
displacement, *411*
for operator overloading, *412*
Vector class
defining, *421*
with operator overloading, *410-26*
Vector class definition, *428*
Virtual base classes, *564-72*
Virtual functions, *535-47*
Virtual keyword, *543, 564, 572, 573*
Void functions, *242-43*

W

While, for versus, *183-85*
While loop, *6, 181-87, 202*
terminating, *216*
using cin to control, *281*
White space, *29, 30 fig, 107, 608, 612, 613*
Width() method, *595*
Write, to a file, *620, 621*
Write() method, *588, 620, 635, 647*

X

X component, *412, 413 fig*
X_flags, *600*

Y

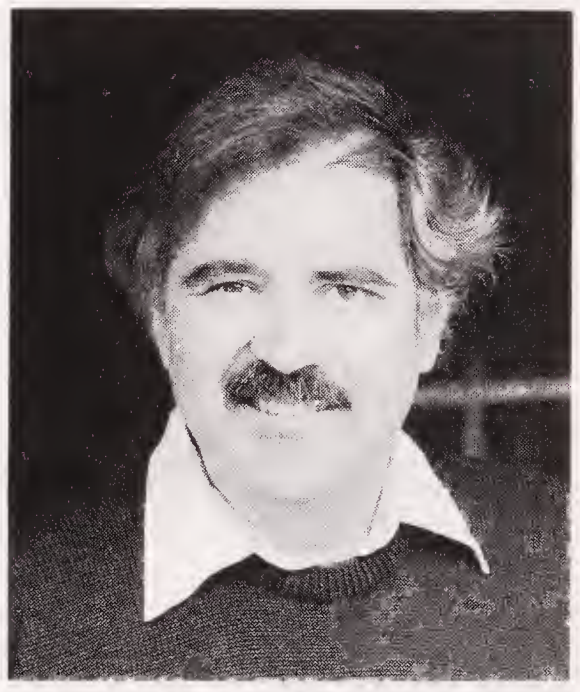
Y component, *412, 413 fig*

Z

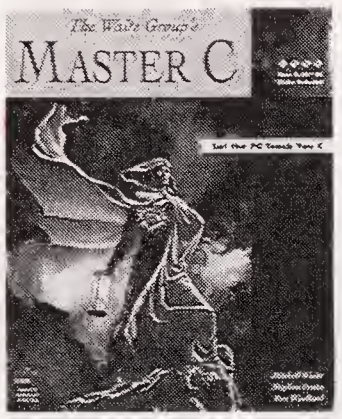
Zeros, printing trailing, *599*
Zortech C++, *11, 14-15*

About the Author

Stephen Prata teaches astronomy, physics, and computer science at the College of Marin in Kentfield, California. He received his B.S. from the California Institute of Technology and his Ph.D. from the University of California, Berkeley. He has been the author or coauthor of more than a dozen books for The Waite Group, including *UNIX Primer Plus*, *Microsoft QuickC Programming*, *Microsoft QuickBASIC Primer Plus*, and *The Waite Group's New C Primer Plus*, which received the Computer Press Association's 1990 Best How-To Computer Book Award.



TITLES FROM

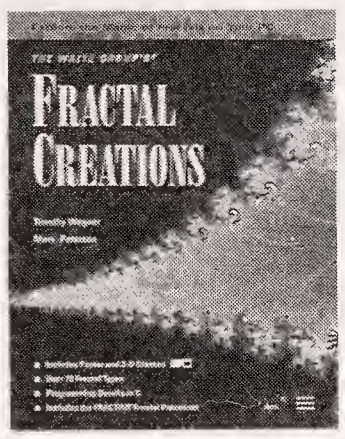


The Waite Group's Master C: *Let the PC Teach You C*

Mitchell Waite, Stephen Prata, Rex Woollard

The programming language known as C has become one of the languages of choice among programmers, due to its enormous range and flexibility which allows it to be used in simple applications as well as advanced operating systems. *Master C: Let the PC Teach You C* is a revolutionary new book/disk package designed to use the power and speed of the IBM PC to provide the fastest, most effective way to learn C. The Master C CBT has features not found in books. It automatically guides the user through C topics, notices problems, and recommends action. Sophisticated answer judgment accepts rough responses (even misspellings). "Remediation paths" send the user to review material not understood. All in all, there is no better or faster way to learn C today than with Master C.

256 Pages, 7 × 9, 3 5.25" disks, Softbound **ISBN: 1-878739-00-X, $44.95**

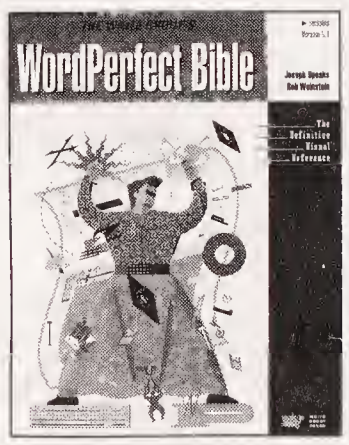


Fractal Creations

Tim Wegner, Mark Peterson

If the magic place between science and art strikes your fancy—if you enjoy mixing special effects, meditation, and, say, a little mathematics—then take a look at *The Waite Group's Fractal Creations*, a creative new book/disk package. Fractals are the mathematical patterns that underlie many futuristic ideas in science. FRACTINT, the program that comes with the book, turns an IBM PC into an exciting "Fractal Processor" for exploring fractal images. You can zoom in on any of FRACTINT's 68 fractal images; then you can rotate, colorize, or color cycle the images for the most beautiful animated patterns imaginable. The program also converts them to 3-D, and creates 3-D mountain ranges, clouds, ferns, or landscapes—a pair of 3-D glasses is included. The book that accompanies the software is an illustrated tutorial on fractals, a reference manual for FRACTINT, and a bible of all the built-in fractals and how they work. A large fold-out color poster features some of the most beautiful fractals created by FRACTINT users. Check out the future with *The Waite Group's Fractal Creations.*

350 Pages, 1 360K 5.25" disk, 7 × 9, Softbound **ISBN: 1-878739-05-0, $34.95**



The Waite Group's WordPerfect Bible

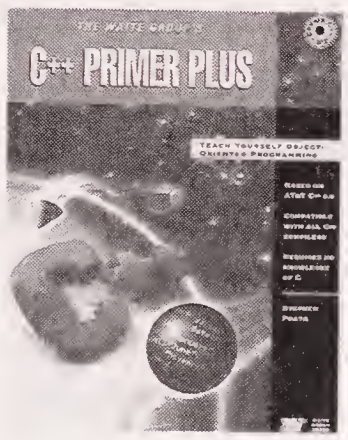
Joseph Speaks, Rob Weinstein

Until now, WordPerfect has been a difficult product to learn, let alone master: its over 165 powerful text control commands are non-intuitive and difficult to remember. Indeed, WordPerfect is especially hard on people new to computers.

The Waite Group's WordPerfect Bible brings an end to this confusion. It's an exciting reference and tutorial to all WordPerfect commands with a specially designed "idea-in-a-block" format that makes it accessible to first-time users as well as power users. Every one of the beautifully designed command spreads in *WordPerfect Bible* contains blocks for purpose, usage techniques, keyboard shortcuts, mouse selection, an example, deeper details, tips, warning, variations, macros, related commands, and a compatibility box. You waste no time reading large tracts of irrelevant text. This book is The Waite Group at its best—for fast and easy lookup, quick and complete alphabetical and by-task "jump tables" of all WordPerfect commands and concepts are featured on the inside covers. Everyone needs *The Waite Group's WordPerfect Bible* next to their computer. Toss out all of your WordPerfect books—this is the only one you'll ever need.

546 Pages, 7 × 9, Softbound **ISBN: 1-878739-01-8, $26.95**

WAITE GROUP® PRESS



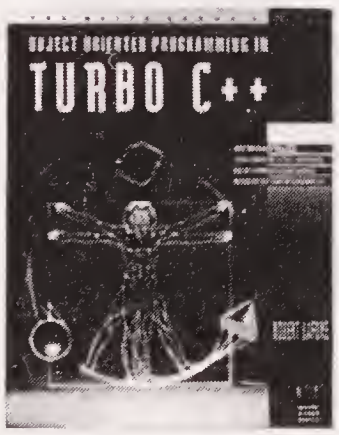
The Waite Group's C++ Primer Plus

Stephen Prata

Computer mavens expect the new programming language C++ to eventually displace C from its position of near-universal preference. An extension of C, C++ treats data and functions as objects (hence the term object-oriented programming or OOP), making it possible to clone, modify, and build upon them. Unlike existing books on the subject, C++ Primer Plus is addressed to those without an extensive programming background. It provides a simple introduction to the standard 2.0 AT&T implementations of C++ while also covering the essential concepts of C. It teaches the basics of OOP and shows how to build programs that are flexible and readily modified.

In the friendly, easy-to-follow style of *C Primer Plus* (the author's best-selling work), *C++ Primer Plus* illustrates the language's fundamentals with short sample programs that are easy to key in and experiment with. A companion disk (available through the order card included with the book) contains all the sample programs and projects. The book is compatible with any AT&T Version 2.0 compliant compiler and works under Unix, Borland, Turbo, Zortec, and Sun C++.

720 Pages, 7 × 9, Softbound **ISBN: 1-878739-02-6, $26.95**

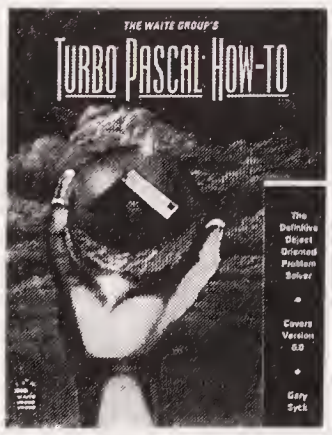


The Waite Group's Object-Oriented Programming in Turbo C++

Robert Lafore

Object-oriented programming (OOP) is the most dramatic—and potentially confusing—innovation in software development since the dawn of the computer age. Based on the idea of treating functions and data as objects, OOP results in programs that are more flexible, more easily maintained, and on the whole, more powerful. *Object-Oriented Programming in Turbo C++* focuses on C as a separate language, distinct from C, and assumes no prior experience with C. Step by step lessons teach Turbo C++, Borland C++, and the basics of OOP at the same time. Suitable for students, hackers, and enthusiasts, *Object-Oriented Programming in Turbo C++* is written by Robert Lafore, author of the best-selling *The Waite Group's Turbo C Programming for the IBM* (over 200,000 copies in print). You'll find details on Turbo C++ graphics and the DOS file system, project chapters enhance your understanding of object-oriented design with a game of life, a program that simulates a town's water distribution system, and a fractal generator, and more. No experience required for this hands-on book.

640 Pages, 7 × 9, Softbound **ISBN: 1-878739-06-9, $26.95**



The Waite Group's Turbo Pascal How-To

Gary Syck

Turbo Pascal How-To is a working programmer's dream: hundreds of typical programming problems, with creative ways to solve them, in an easy-to-use reference format. The solutions provided are designed to work with the latest, "object-oriented" Version 6.0 of Turbo Pascal, as well as earlier versions. Experienced programmers will appreciate the ease of incorporating these solutions, which will allow them to concentrate on their program's unique characteristics and not waste time solving problems. At the same time, novice programmers will find they can quickly begin creating functional programs just by building on the modular examples provided here. Borland's new Turbo Vision interface system as well as the new Windows 3.0 extensions are presented for the first time.

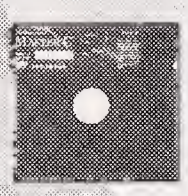
The Waite Group's *Turbo Pascal How-To* answers the most practical questions any programmer has, such as: "How do I put information in a window? How do I build a fancy pull-down menu system? How do I scroll the screen? Create a directory? Save memory?" And more. In short, this book has everything readers need to know to write professional Turbo Pascal programs.

500 Pages, 7 × 9, Softbound **ISBN: 1-878739-04-2, $24.95**

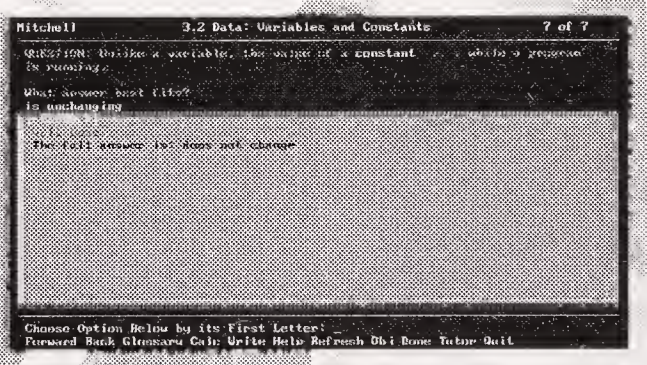
The Waite Group's

MASTER C

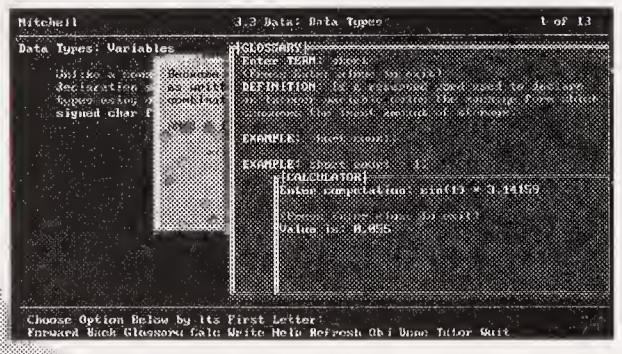
Computer Based C Training for the IBM PC



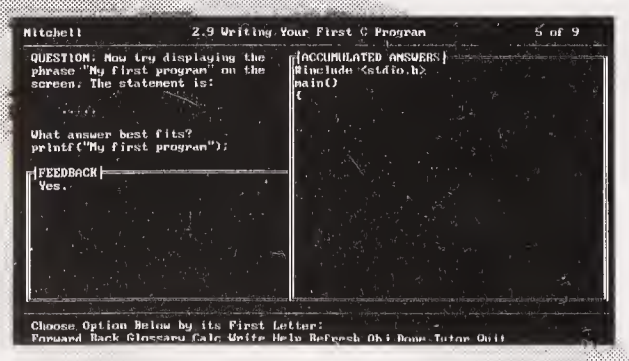
Master C is a revolutionary book/disk package designed to turn your PC into an instructor that teaches you to write powerful C programs.



The Master C disk guides you through all the essential C topics—from data types to pointers and data structures. It teaches you, quizzes you, notices problems, and recommends action. A sophisticated answer judgment technique accepts rough responses and even misspellings.



Master C features windows, an on-line glossary, a calculator, electronic bookmarks, and it keeps track of student scores for classroom use. Answer collection can be turned on and off. Even screen colors can be changed.



The Master C book provides lessons you can use when you're away from your computer, and it features a reference section that explains the use and syntax of every keyword and function in C. Accumulated answer "blackboards" let you learn C by building the program in a window.

PACKAGE CONTAINS: 360K 5.25" disks, companion book, and registration card.

SYSTEM REQUIREMENTS: An IBM PC or 100%-compatible computer, a 384K system memory, DOS 3.0 or later, a hard disk and 5.25" floppy-disk drive, and a monochrome or color adaptor.

PRICE: $44.95 U.S.A. ***See Order Form.***

Waite Group Press 200Tamal Plaza Corte Madera California 94925 (415) 924-2515

Production for this book was done using desktop publishing techniques and every phase of the book involved the use of computer technology. Never did production have to use traditional typesetting, stats, or photos, and everything for this book, from the illustrations to the formatted text, was saved on disk. Only the cover painting was created in the traditional manner.

The author wrote portions of the text in Microsoft Word 4 on a Macintosh SE-30, and portions in Word for Windows, running under Windows 3.0. The C++ code was originally generated using Turbo C++ on a PC 386. The source code files were imported into WinWord, and program output was transferred using cut and paste methods. The Word for Windows files were converted to RTF (rich text format) files, sent to the Macintosh using MacLink, and opened directly from Microsoft Word for the Mac. For the portions written using Microsoft Word Mac, C++ code was sent to the Macintosh using MacLink and pasted into the Microsoft Word text. After the book was edited and revised, the program code was copied from Macintosh Word, converted to ASCII, sent to the PC, and retested.

All page formatting and book design was done in PageMaker 4.01, using the imported Microsoft Word files. Adobe and Bitstream fonts were used.

Illustrations were done in Aldus Freehand and saved as EPS files for placement in PageMaker. Backmatter figures were done on a Macintosh IIfx using Freehand. The icons (globe, cube, cone) were created using Adobe Illustrator.

The cover was created as a traditional airbrush painting and was traditionally separated. Quark Express was used for cover design. For the chapter openers and the graphics that adorn the tops of pages, the cover art was scanned using a MicroTek color scanner, altered in ImageStudio, and imported into PageMaker.

Final files were sent on Syquest 44 Mb removable disk cartridges to the printer, R.R. Donnelley & Sons Co., where they were directly imposed to film through a Macintosh IIfx and Linotronic 530 phototypesetting machine, utilizing Adobe and Monotype fonts. Plates were then made from the film.

AS A PUBLISHER AND WRITER WITH OVER 360,000 BOOKS SOLD EACH YEAR, IT CAME AS A GREAT SHOCK TO DISCOVER THAT OUR RAIN FORESTS, HOME FOR HALF OF ALL LIVING THINGS ON EARTH, ARE BEING DESTROYED AT THE RATE OF 50 ACRES PER MINUTE & AT THIS RATE THE RAIN FORESTS WILL COMPLETELY DISAPPEAR IN JUST 50 YEARS & BOOKS HAVE A LARGE INFLUENCE ON THIS RAMPANT DESTRUCTION & FOR EXAMPLE, SINCE IT TAKES 17 TREES TO PRODUCE ONE TON OF PAPER, A FIRST PRINTING OF 30,000 COPIES OF A TYPICAL 480 PAGE BOOK CONSUMES 108,000 POUNDS OF PAPER WHICH WILL REQUIRE 918 TREES. TO HELP OFFSET THIS LOSS, WAITE GROUP PRESS WILL PLANT TWO TREES FOR EVERY TREE FELLED FOR PRODUCTION OF THIS BOOK & THE DONATION WILL BE MADE TO RAINFOREST ACTION NETWORK (THE BASIC FOUNDATION, P.O. BOX 47012, ST. PETERSBURG, FL 33743), WHICH CAN PLANT 1,000 TREES FOR $250.

YOU'VE READ THE BOOK. NOW GET:



The Waite Group's C++ Primer Plus: The Companion Disk

The Time of Your Life is pretty valuable. Why waste it typing in computer code? If you measure in mere dollars, consider that you're worth $40 an hour (not atypical for a programmer). If it takes you ten minutes to type in and debug a half page of code, then the listings in this book would take you about 16 hours to type, consuming $640 worth of your time!

But your time is more valuable than money, isn't it? Think of all the ways you could spend your time that would be more fulfilling than typing code. You could be creating 3-D fractals. You could be walking along the beach pondering the structure of a double helix. You could be practicing genetic engineering on your house pets.

How many times have you heard "Not tonight, honey, I have to type in the source code from *The Waite Group's C++ Primer Plus.*" Gets pretty tiresome after awhile, doesn't it?

Well, we've made it easy for you by creating *C++ Primer Plus: The Companion Disk.* This disk (available in 3.5-inch or 5.25-inch formats) contains all the program listings featured in *C++ Primer Plus*, organized in chapter directories, and named to match the book's listings. Each has been pre-tested and debugged. So just load these educational listings into your C++ compiler and watch the fun.

Thrill to:

- Potato Nose Muggins and Polly Morf in `strtref.cpp`
- Glorious Gloria and Audacious Arthur of `arrayone.cpp`
- The Amazing Accounto averaging sums in `block.cpp`
- The number guessing game `ifelseif.cpp`
- The inlaw IQ tracker `and.cpp`
- The Lotto odds calculator `lotto.cpp`
- The yam-counting capabilities of `arrayone.cpp`
- The object-oriented softball stats program `softbal3.cpp`

And so much more for a mere $19.95 (plus applicable taxes, and shipping fees).

There's more to life than typing programs. Use the postage-paid card in this book to order *The Waite Group's C++ Primer Plus: The Companion Disk* and have the Time of Your Life.

Disk Order Form

Filling out this order form can change your life, help you impress friends, and perhaps even influence people.

The companion disk for *The Waite Group's C++ Primer Plus* includes all of the C++ program listings contained in this book, organized by chapter, and pre-tested and debugged.

The Waite Group's Master C is a disk-based training system that turns your IBM PC into a C instructor. Master C uses modern computer-based techniques including sophisticated answer judgement to teach C, from fundamentals through advanced concepts. It presents information, gives quizzes, keeps score, notices problems, recommends action, and even lets you chart your own C course.

To order by phone call 800-368-9369 or 415-924-2576 (FAX)
or send to The Waite Group, 200 Tamal Plaza, Corte Madera, CA 94925

Name

Company

Address
Street Address Only, No P.O. Box

City State ZIP –

Daytime Phone

Quantity and Type

Name	Item #	Quantity	Price
C++ Primer Plus companion disk	WGP-CD10	☐	x $19.95 =
Master C training system	CBT-1	☐	x $44.95 =

Sales Tax—California addresses add 7.25% sales tax.

Shipping—Add $5 USA, $10 Canada, or $30 Foreign for shipping and handling. Standard shipping is UPS Ground. Allow 3 to 4 weeks. Prices subject to change. Purchase orders subject to credit approval, and verbal purchase orders will not be accepted.

Disk Type: ☐ 5.25-inch ☐ 3.5-inch

Sub Total

Sales Tax

Shipping

Total Due

Method of Payment

Checks or money orders, payable to The Waite Group. To pay by credit card, complete the following:

☐ Visa ☐ Mastercard Card Number

Cardholder's Name _____ Exp. Date

Cardholder's Signature _____

©1992 The Waite Group, Inc. All rights reserved. The Waite Group is a registered trademark of The Waite Group, Inc.

Disk Order Form

Filling out this order form can change your life, help you impress friends, and perhaps even influence people. The companion disk for *The Waite Group's C++ Primer Plus* includes all of the C++ program listings contained in this book, organized by chapter, and pre-tested and debugged.

The Waite Group's Master C is a disk-based training system that turns your IBM PC into a C instructor. Master C uses modern computer-based techniques including sophisticated answer judgement to teach C, from fundamentals through advanced concepts. It presents information, gives quizzes, keeps score, notices problems, recommends action, and even lets you chart your own C course.

To order by phone call 800-368-9369 or 415-924-2576 (FAX)
or send to The Waite Group, 200 Tamal Plaza, Corte Madera, CA 94925

Name

Company

Address
Street Address Only, No P.O. Box

City State ZIP –

Daytime Phone

Quantity and Type

Name	Item #	Quantity	Price
C++ Primer Plus companion disk	WGP-CD10	☐	x $19.95 =
Master C training system	CBT-1	☐	x $44.95 =

Sales Tax—California addresses add 7.25% sales tax.

Shipping—Add $5 USA, $10 Canada, or $30 Foreign for shipping and handling. Standard shipping is UPS Ground. Allow 3 to 4 weeks. Prices subject to change. Purchase orders subject to credit approval, and verbal purchase orders will not be accepted.

Disk Type: ☐ 5.25-inch ☐ 3.5-inch

Sub Total

Sales Tax

Shipping

Total Due

Method of Payment

Checks or money orders, payable to The Waite Group. To pay by credit card, complete the following:

☐ Visa ☐ Mastercard Card Number

Cardholder's Name _____ Exp. Date

Cardholder's Signature _____

©1992 The Waite Group, Inc. All rights reserved. The Waite Group is a registered trademark of The Waite Group, Inc.

$26.95 U.S.

Computer Programming Language

THE WAITE GROUP®

C++ PRIMER PLUS®

Teach Yourself Object-Oriented Programming

BY STEPHEN PRATA

Winner of the 1990 CPA Best How-To Computer Book Award

Finally, here is the first no-experience-required, fun-to-read book on the C++ programming language, *C++ Primer Plus*. Learn about this new language's full power without getting bogged down in technicalities.

C++ Primer Plus teaches "generic" C++ from the ground up and assumes no prior experience in the C language. You'll learn the basics of object-oriented programming (OOP) and how to use classes, inheritance, and polymorphism to build programs that are flexible and easy to modify. Following up on his immensely successful classic *C Primer Plus*, Stephen Prata presents gentle, humorous, and interesting explanations and examples.

C++ Primer Plus lessons work with any of the new AT&T C++ 2.0 and 2.1-compatible compilers, including Borland and Turbo C++, Sun cfront C++, Zortech C++, and AT&T Unix C++. Suitable for classroom use or self-directed study, the book contains end-of-chapter quizzes, exercises, chapter summaries, and informative notes.

C++ Primer Plus:

- Presents C and C++ at the same time
- Teaches Object-Oriented Programming (OOP) from the ground up
- Covers Objects, Classes, Overloading, Constructors, Inheritance, Virtual Functions
- Provides a fun, insightful, and practical approach to the C++ language

IF YOU WANT TO LEARN C++, WHY NOT LET A CLASS ACT HELP YOU OUT?

"Prata makes a relatively complex subject — professional programming — clear and enjoyable. He also manages to teach both people completely new to programming and programmers new to C without being too technical or condescending. Plus, he anticipates questions, prejudices, and real-world situations — almost like a live teacher."

Computer Press Association

The Waite Group was founded by Mitchell Waite, a computer book author since 1974. The Waite Group has produced over 70 titles, including such best-sellers as *C Primer Plus, Fractal Creations, Microsoft C Bible, MS-DOS Bible, Tricks of the MS-DOS Masters, and Master C: Let the PC Teach You C.*



WAITE GROUP PRESS™
200 Tamal Plaza, Corte Madera, CA 94925

