STARTING OUT WITH C++

Early Objects

seventh edition



TONY GADDIS • JUDY WALTERS • GODFREY MUGANDA

LOCATION OF VIDEONOTES IN THE TEXT



Chapter 1	Designing a Program with Pseudocode, p. 19 Designing the Account Balance Program, p. 24 Predicting the Output of Problem 30, p. 24 Solving the Candy Bar Sales Problem, p. 25
Chapter 2	Using `cout` to Display Output, p. 32 Assignment Statements, p. 59 Arithmetic Operators, p. 61 Solving the Restaurant Bill Problem, p. 72
Chapter 3	Using `cin` to Read Input, p. 75 Evaluating Mathematical Expressions, p. 81 Combined Assignment Operators, p. 102 Solving the Stadium Seating Problem, p. 151
Chapter 4	Using an `if` Statement, p. 162 Using an `if/else` Statement, p. 172 Using an `if/else if` Statement, p. 175 Solving the Time Calculator Problem, p. 236
Chapter 5	The `while` Loop, p. 249 The `for` Loop, p. 263 Nested Loops, p. 277 Solving the Ocean Levels Problem, p. 299
Chapter 6	Defining and Calling Functions, p. 306 Using Function Arguments, p. 316 Value-Returning Functions, p. 326 Solving the Markup Problem, p. 380
Chapter 7	Creating a Class, p. 391 Creating and Using Class Objects, p. 393 Creating and Using Structures, p. 436 Solving the `Car` Class Problem, p. 480
Chapter 8	Accessing Array Elements, p. 487 Passing an Array to a Function, p. 517 Two-Dimensional Arrays, p. 526 Solving the Chips and Salsa Problem, p. 567
Chapter 9	Performing a Binary Search, p. 580 Sorting a Set of Data, p. 587 Solving the Lottery Winners Problem, p. 616

(continued on next page)

LOCATION OF VIDEONOTES IN THE TEXT *(continued)*



Chapter 10	Pointer Variables, p. 621 Dynamically Allocating an Array, p. 645 Solving the Days in Current Month Problem, p. 676
Chapter 11	Operator Overloading, p. 704 Aggregation and Composition, p. 734 Overriding Base Class Functions, p. 755 Solving the Number of Days Worked Problem, p. 768
Chapter 12	Converting Strings to Numbers, p. 788 Writing a C-String Handling Function, p. 800 Solving the Backward String Problem, p. 823
Chapter 13	Passing File Stream Objects to Functions, p. 845 Rewinding a File, p. 858 Solving the File Encryption Filter Problem, p. 897
Chapter 14	Recursive Binary Search, p. 913 QuickSort, p. 915 Solving the Recursive Multiplication Problem, p. 933
Chapter 15	Polymorphism, p. 941 Composition Versus Inheritance, p. 952 Solving the Sequence Sum Problem, p. 970
Chapter 16	Throwing and Handling Exceptions, p. 975 Writing a Function Template, p. 986 Iterators, p. 1004 Solving the Arithmetic Exceptions Problem, p. 1020
Chapter 17	Adding an Element to a Linked List, p. 1031 Removing an Element from a Linked List, p. 1038 Solving the Member Insertion by Position Problem, p. 1069
Chapter 18	Storing Objects in an STL Stack, p. 1083 Storing Objects in an STL Queue, p. 1097 Solving the File Reverser Problem, p. 1109
Chapter 19	Inserting an Element into a Binary Tree, p. 1118 Removing an Element from a Binary Tree, p. 1122 Solving the Tree Size Problem, p. 1138

Seventh Edition

Starting Out with

C++ Early Objects

Tony Gaddis
Judy Walters
Godfrey Muganda

Addison-Wesley
Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editor-in-Chief: Michael Hirsch
Editorial Assistant: Stephanie Sellinger
Director of Marketing: Margaret Whaples
Marketing Coordinator: Kathryn Ferranti
Managing Editor: Jeffrey Holcomb
Production Project Manager: Heather McNally
Senior Manufacturing Buyer: Carol Melville
Media Manufacturing Buyer: Ginny Michaud
Art Director: Linda Knowles
Cover and Interior Designer: Joyce Cosentino Wells
Cover Art: © 2010 Dmitriy Ustyujanin/iStockphoto
Media Project Manager: Katelyn Boller
Full-Service Project Management: Peggy Kellar, Aptara®, Inc.
Composition: Aptara®, Inc.
Copyeditor: Evelyn Perricone

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on appropriate page within text.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. Screen shots and icons reprinted with permission from the Microsoft Corporation. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

Copyright © 2011, 2008, 2006, 2005. Pearson Education, Inc., publishing as Addison-Wesley, 501 Boylston Street, Suite 900, Boston, Massachusetts 02116. All rights reserved. Manufactured in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, 501 Boylston Street, Suite 900, Boston, Massachusetts 02116.

Many of the designations by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Library of Congress Cataloging-in-Publication Data

Gaddis, Tony.
Starting out with C++ : early objects / Tony Gaddis, Judy Walters, Godfrey Muganda.—Seventh ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-13-607774-9 (alk. paper)
1. C++ (Computer program language) I. Walters, Judy. II. Muganda, Godfrey. III. Title.
QA76.73.C153G33 2010
005.13'3—dc22 2010004498

10 9 8 7 6 5 4 3 2 1—EB—14 13 12 11 10

Addison-Wesley
is an imprint of

PEARSON

www.pearsonhighered.com

ISBN 10: 0-13-607774-9
ISBN 13: 978-0-13-607774-9

Contents at a Glance

Preface xv

CHAPTER 1 Introduction to Computers and Programming 1

CHAPTER 2 Introduction to C++ 27

CHAPTER 3 Expressions and Interactivity 75

CHAPTER 4 Making Decisions 157

CHAPTER 5 Looping 243

CHAPTER 6 Functions 305

CHAPTER 7 Introduction to Classes and Objects 387

CHAPTER 8 Arrays 485

CHAPTER 9 Searching, Sorting, and Algorithm Analysis 577

CHAPTER 10 Pointers 619

CHAPTER 11 More About Classes and Object-Oriented Programming 677

CHAPTER 12 More About Characters, Strings, and the `string` Class 771

CHAPTER 13 Advanced File and I/O Operations 829

CHAPTER 14 Recursion 901

CHAPTER 15 Polymorphism and Virtual Functions 935

CHAPTER 16 Exceptions, Templates, and the Standard Template Library (STL) 973

CHAPTER 17 Linked Lists 1023

CHAPTER 18 Stacks and Queues 1071

CHAPTER 19 Binary Trees 1111

Appendix A: The ASCII Character Set 1141

Appendix B: Operator Precedence and Associativity 1145

Index 1147

Student CD The following appendices are on the accompanying Student CD.

Appendix C: A Brief Introduction to Object-Oriented Programming
Appendix D: Using UML in Class Design
Appendix E: Namespaces
Appendix F: Passing Command Line Arguments
Appendix G: Header File and Library Function Reference
Appendix H: Binary Numbers and Bitwise Operations
Appendix I: C++ Casts and Run-Time Type Identification
Appendix J: Multi-Source File Programs
Appendix K: Multiple and Virtual Inheritance
Appendix L: Introduction to the MinGW C++ Compiler and the wxDev-C++ IDE
Appendix M: Introduction to Microsoft Visual C++ 2008 Express Edition
Appendix N: .NET and Managed C++
Appendix O: Introduction to Flowcharting
Appendix P: Answers to Checkpoints
Appendix Q: Answers to Odd-Numbered Review Questions

Contents

Preface xv

CHAPTER 1 Introduction to Computers and Programming 1

1.1 Why Program? 1
1.2 Computer Systems: Hardware and Software 2
1.3 Programs and Programming Languages 6
1.4 What Is a Program Made of? 12
1.5 Input, Processing, and Output 16
1.6 The Programming Process 17
1.7 Tying It All Together: *Hi! It's Me* 22

CHAPTER 2 Introduction to C++ 27

2.1 The Parts of a C++ Program 27
2.2 The `cout` Object 31
2.3 The `#include` Directive 35
2.4 Standard and Prestandard C++ 37
2.5 Variables, Constants, and the Assignment Statement 37
2.6 Identifiers 41
2.7 Integer Data Types 43
2.8 The `char` Data Type 48
2.9 The C++ `string` Class 52
2.10 Floating-Point Data Types 54
2.11 The `bool` Data Type 57
2.12 Determining the Size of a Data Type 58
2.13 More on Variable Assignments and Initialization 59
2.14 Scope 60
2.15 Arithmetic Operators 61
2.16 Comments 64
2.17 Focus on Software Engineering: *Programming Style* 65
2.18 Tying It All Together: *Smile!* 67

CHAPTER 3 Expressions and Interactivity 75

3.1 The `cin` Object 75
3.2 Mathematical Expressions 81
3.3 Implicit Type Conversion 89
3.4 Explicit Type Conversion 90
3.5 Overflow and Underflow 94
3.6 Named Constants 97
3.7 Multiple and Combined Assignment 101
3.8 Formatting Output 105
3.9 Working with Characters and String Objects 115
3.10 Using C-Strings 120
3.11 More Mathematical Library Functions 126
3.12 Introduction to Files 130
3.13 Focus on Debugging: *Hand Tracing a Program* 138
3.14 Green Fields Landscaping Case Study—Part 1 140
3.15 Tying It All Together: *Word Game* 142

CHAPTER 4 Making Decisions 157

4.1 Relational Operators 157
4.2 The `if` Statement 162
4.3 The `if/else` Statement 172
4.4 The `if/else if` Statement 175
4.5 Menu-Driven Programs 181
4.6 Nested `if` Statements 183
4.7 Logical Operators 187
4.8 Validating User Input 196
4.9 More About Variable Definitions and Scope 197
4.10 Comparing Characters and Strings 202
4.11 The Conditional Operator 206
4.12 The `switch` Statement 210
4.13 Enumerated Data Types 219
4.14 Testing for File Open Errors 222
4.15 Focus on Testing and Debugging: *Validating Output Results* 223
4.16 Green Fields Landscaping Case Study—Part 2 225
4.17 Tying It All Together: *Fortune Teller* 229

CHAPTER 5 Looping 243

5.1 The Increment and Decrement Operators 243
5.2 Introduction to Loops: The `while` Loop 249
5.3 Using the `while` Loop for Input Validation 253
5.4 Counters 256
5.5 The `do-while` Loop 258
5.6 The `for` Loop 263
5.7 Keeping a Running Total 269
5.8 Sentinels 271
5.9 Using a Loop to Read Data from a File 273
5.10 Focus on Software Engineering: *Deciding Which Loop to Use* 276
5.11 Nested Loops 277
5.12 Breaking Out of a Loop 279
5.13 The `continue` Statement 281

5.14 Focus on Testing and Debugging: *Creating Good Test Data* 284
5.15 Central Mountain Credit Union Case Study 287
5.16 Tying It All Together: *What a Colorful World* 291

CHAPTER 6 Functions 305

6.1 Modular Programming 305
6.2 Defining and Calling Functions 306
6.3 Function Prototypes 314
6.4 Sending Data into a Function 316
6.5 Passing Data by Value 321
6.6 The `return` Statement 325
6.7 Returning a Value from a Function 326
6.8 Returning a Boolean Value 332
6.9 Using Functions in a Menu-Driven Program 334
6.10 Local and Global Variables 337
6.11 Static Local Variables 344
6.12 Default Arguments 347
6.13 Using Reference Variables as Parameters 350
6.14 Overloading Functions 359
6.15 The `exit()` Function 363
6.16 Stubs and Drivers 366
6.17 Little Lotto Case Study 368
6.18 Tying It All Together: *Glowing Jack-o-lantern* 373

CHAPTER 7 Introduction to Classes and Objects 387

7.1 Abstract Data Types 387
7.2 Object-Oriented Programming 389
7.3 Introduction to Classes 391
7.4 Introduction to Objects 393
7.5 Defining Member Functions 395
7.6 Constructors 402
7.7 Destructors 407
7.8 Private Member Functions 410
7.9 Passing Objects to Functions 413
7.10 Object Composition 420
7.11 Focus on Software Engineering: *Separating Class Specification, Implementation, and Client Code* 424
7.12 Input Validation Objects 431
7.13 Structures 435
7.14 Home Software Company OOP Case Study 449
7.15 Introduction to Object-Oriented Analysis and Design 455
7.16 Screen Control 464
7.17 Tying It All Together: *Yoyo Animation* 469

CHAPTER 8 Arrays 485

8.1 Arrays Hold Multiple Values 485
8.2 Accessing Array Elements 487
8.3 Inputting and Displaying Array Contents 489
8.4 Array Initialization 496
8.5 Processing Array Contents 502

8.6 Using Parallel Arrays 513
8.7 The `typedef` Statement 516
8.8 Arrays as Function Arguments 517
8.9 Two-Dimensional Arrays 526
8.10 Arrays with Three or More Dimensions 534
8.11 Vectors 537
8.12 Arrays of Class Objects 549
8.13 National Commerce Bank Case Study 559
8.14 Tying It All Together: *Rock, Paper, Scissors* 561

CHAPTER 9 Searching, Sorting, and Algorithm Analysis 577

9.1 Introduction to Search Algorithms 577
9.2 Searching an Array of Objects 584
9.3 Introduction to Sorting Algorithms 587
9.4 Sorting an Array of Objects 596
9.5 Sorting and Searching Vectors 599
9.6 Introduction to Analysis of Algorithms 601
9.7 Case Studies 609
9.8 Tying It All Together: *Secret Messages* 610

CHAPTER 10 Pointers 619

10.1 Pointers and the Address Operator 619
10.2 Pointer Variables 621
10.3 The Relationship Between Arrays and Pointers 625
10.4 Pointer Arithmetic 629
10.5 Initializing Pointers 630
10.6 Comparing Pointers 632
10.7 Pointers as Function Parameters 635
10.8 Pointers to Constants and Constant Pointers 639
10.9 Focus on Software Engineering: *Dynamic Memory Allocation* 643
10.10 Focus on Software Engineering: *Returning Pointers from Functions* 648
10.11 Pointers to Class Objects and Structures 652
10.12 Focus on Software Engineering: *Selecting Members of Objects* 659
10.13 United Cause Relief Agency Case Study 661
10.14 Tying It All Together: *Pardon Me, Do You Have the Time?* 669

CHAPTER 11 More About Classes and Object-Oriented Programming 677

11.1 The `this` Pointer and Constant Member Functions 677
11.2 Static Members 681
11.3 Friends of Classes 689
11.4 Memberwise Assignment 694
11.5 Copy Constructors 695
11.6 Operator Overloading 704
11.7 Type Conversion Operators 728
11.8 Convert Constructors 731
11.9 Aggregation and Composition 734
11.10 Inheritance 740
11.11 Protected Members and Class Access 745

11.12 Constructors, Destructors, and Inheritance 750
11.13 Overriding Base Class Functions 755
11.14 Tying It All Together: *Putting Data on the World Wide Web* 757

CHAPTER 12 More About Characters, Strings, and the `string` Class 771

12.1 C-Strings 771
12.2 Library Functions for Working with C-Strings 776
12.3 Conversions Between Numbers and Strings 787
12.4 Character Testing 793
12.5 Character Case Conversion 797
12.6 Writing Your Own C-String Handling Functions 800
12.7 More About the C++ `string` Class 805
12.8 Creating Your Own `String` Class 810
12.9 Advanced Software Enterprises Case Study 817
12.10 Tying It All Together: *Program Execution Environments* 819

CHAPTER 13 Advanced File and I/O Operations 829

13.1 Files 829
13.2 Output Formatting 838
13.3 Passing File Stream Objects to Functions 845
13.4 More Detailed Error Testing 847
13.5 Member Functions for Reading and Writing Files 849
13.6 Binary Files 862
13.7 Creating Records with Structures 866
13.8 Random-Access Files 871
13.9 Opening a File for Both Input and Output 878
13.10 Online Friendship Connections Case Study 883
13.11 Tying It All Together: *File Merging and Color-Coded HTML* 888

CHAPTER 14 Recursion 901

14.1 Introduction to Recursion 901
14.2 The Recursive Factorial Function 908
14.3 The Recursive gcd Function 910
14.4 Solving Recursively Defined Problems 911
14.5 A Recursive Binary Search Function 913
14.6 Focus on Problem Solving and Program Design: *The QuickSort Algorithm* 915
14.7 The Towers of Hanoi 919
14.8 Focus on Problem Solving: *Exhaustive and Enumeration Algorithms* 922
14.9 Focus on Software Engineering: *Recursion Versus Iteration* 926
14.10 Tying It All Together: *Infix and Prefix Expressions* 927

CHAPTER 15 Polymorphism and Virtual Functions 935

15.1 Type Compatibility in Inheritance Hierarchies 935
15.2 Polymorphism and Virtual Member Functions 941
15.3 Abstract Base Classes and Pure Virtual Functions 946
15.4 Focus on Object-Oriented Programming: *Composition Versus Inheritance* 952
15.5 Secure Encryption Systems, Inc., Case Study 957
15.6 Tying It All Together: *Let's Move It* 961

CHAPTER 16 Exceptions, Templates, and the Standard Template Library (STL) 973

16.1 Exceptions 973
16.2 Function Templates 985
16.3 Class Templates 993
16.4 Class Templates and Inheritance 999
16.5 Introduction to the Standard Template Library 1002
16.6 Tying It All Together: *Word Transformers Game* 1015

CHAPTER 17 Linked Lists 1023

17.1 Introduction to the Linked List ADT 1023
17.2 Linked List Operations 1029
17.3 A Linked List Template 1041
17.4 Recursive Linked List Operations 1045
17.5 Variations of the Linked List 1054
17.6 The STL `list` Container 1054
17.7 Reliable Software Systems, Inc., Case Study 1056
17.8 Tying It All Together: *More on Graphics and Animation* 1060

CHAPTER 18 Stacks and Queues 1071

18.1 Introduction to the Stack ADT 1071
18.2 Dynamic Stacks 1079
18.3 The STL `Stack` Container 1082
18.4 Introduction to the Queue ADT 1084
18.5 Dynamic Queues 1092
18.6 The STL `deque` and `queue` Containers 1096
18.7 Focus on Problem Solving and Program Design: *Eliminating Recursion* 1098
18.8 Tying It All Together: *Converting Postfix Expressions to Infix* 1103

CHAPTER 19 Binary Trees 1111

19.1 Definition and Applications of Binary Trees 1111
19.2 Binary Search Tree Operations 1115
19.3 Template Considerations for Binary Search Trees 1131
19.4 Tying It All Together: *Genealogy Trees* 1131

Appendix A: The ASCII Character Set 1141
Appendix B: Operator Precedence and Associativity 1145
Index 1147

Student CD The following appendices are on the accompanying Student CD.

Appendix C: A Brief Introduction to Object-Oriented Programming
Appendix D: Using UML in Class Design
Appendix E: Namespaces
Appendix F: Passing Command Line Arguments
Appendix G: Header File and Library Function Reference
Appendix H: Binary Numbers and Bitwise Operations
Appendix I: C++ Casts and Run-Time Type Identification
Appendix J: Multi-Source File Programs
Appendix K: Multiple and Virtual Inheritance
Appendix L: Introduction to the MinGW C++ Compiler and the wxDev-C++ IDE
Appendix M: Introduction to Microsoft Visual C++ 2008 Express Edition
Appendix N: .NET and Managed C++
Appendix O: Introduction to Flowcharting
Appendix P: Answers to Checkpoints
Appendix Q: Answers to Odd-Numbered Review Questions

This page intentionally left blank

Preface

Welcome to *Starting Out with C++: Early Objects*, 7th Edition. This book is intended for use in a two-term or three-term C++ programming sequence, or an accelerated one-term course. Students new to programming, as well those with prior course work in other languages, will find this text beneficial. The fundamentals of programming are covered for the novice, while the details, pitfalls, and nuances of the C++ language are explored in-depth for both the beginner and more experienced student. The book is written with clear, easy-to-understand language and it covers all the necessary topics for an introductory programming course. This text is rich in example programs that are concise, practical, and real world oriented, ensuring that the student not only learns how to implement the features and constructs of C++, but why and when to use them.

What's New in the Seventh Edition

This book's pedagogy, organization, and clear writing style remain the same as in the previous edition. However, many improvements have been made to make it even more student-friendly and to keep it state of the art for introductory programming using the C++ programming language.

- **Updated Material**
  Material has been updated throughout the book to reflect changes in technology, operating systems, and software development environments, as well as to improve clarity and incorporate best practices in object-oriented programming.
- **New Material**
  New material has been added on a number of topics including embedding operating system calls in program code, using object composition and aggregation, and creating text-based graphics.
- **Completely Revised Chapter 7**
  Chapter 7, *Introduction to Classes and Objects*, has been reorganized and almost entirely rewritten to start right in with classes and objects, instead of introducing structures first.

- **Greater Focus on Object-Oriented Programming**
  Many examples throughout the text have been rewritten to incorporate appropriate use of classes and objects.
- **Reusability**
  Material has been added illustrating how to create general classes that can be appropriately reused in multiple applications.
- **Improved Diagrams**
  Many diagrams have been improved and new diagrams added to better illustrate important concepts.
- **Online VideoNotes**
  An extensive set of online videos have been developed to accompany this text. Throughout the book, *VideoNotes* icons alert the student to videos covering specific topics they are studying. Additionally, one Programming Challenge at the end of each chapter now has an accompanying video explaining how to develop the problem's solution. The videos are available at `http://www.pearsonhighered.com/gaddis/`
- **New *Tying It All Together* Sections**
  A new *Tying It All Together* section has been added at the end of every chapter that shows the student how to do something clever and fun with the material covered in that chapter.
- **New Programming Challenges**
  New Programming Challenges have been added to every chapter, including a number of Challenges that ask students to develop object-oriented solutions and to create solutions that reuse, modify, and build on previously written code.
- **New Compiler and IDE Bundled with the Book**
  The MinGW C++ Compiler and wxDev-C++ Software Development Environment now come bundled, for free, with the book.
- **New Appendices**
  An Appendix has been added on using the MinGW C++ Compiler and wxDev-C++ IDE that accompany the book. Additional new appendices cover the Microsoft Visual C++ 2008 Express Edition IDE and Multiple and Virtual Inheritance.

Organization of the Text

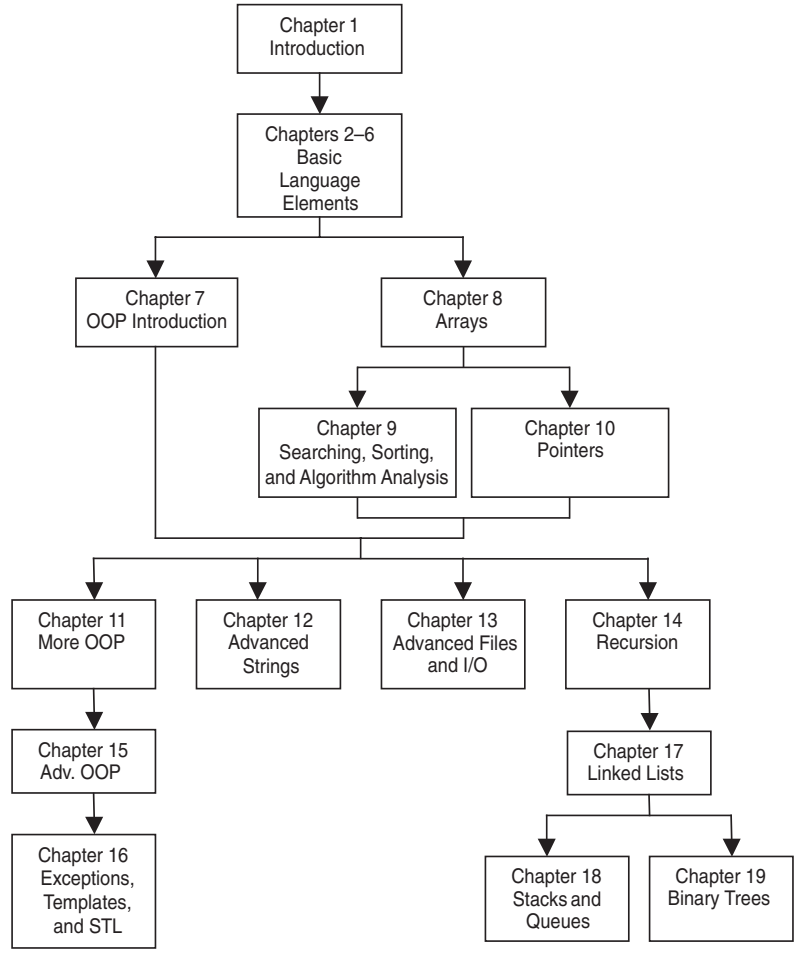
This text teaches C++ in a step-by-step fashion. Each chapter covers a major set of topics and builds knowledge as the student progresses through the book. Although the chapters can be easily taught in their existing sequence, flexibility is provided. The following dependency diagram (Figure P-1) suggests possible sequences of instruction.

Chapter 1 covers fundamental hardware, software, and programming concepts. The instructor may choose to skip this chapter if the class has already mastered those topics. Chapters 2 through 6 cover basic C++ syntax, data types, expressions, selection structures, repetition structures, and functions. Each of these chapters builds on the previous chapter and should be covered in the order presented.

Chapter 7 introduces object-oriented programming. It can be covered any time after Chapter 6, but before Chapter 11. Instructors who prefer to introduce arrays before classes can cover Chapter 8 before Chapter 7. In this case it is only necessary to postpone section 8.12 (Arrays of Class Objects) until Chapter 7 has been covered.

As Figure P-1 illustrates, in the second half of the book Chapters 11, 12, 13, and 14 can be covered in any order. Chapters 11, 15, and 16, however, should be done in sequence. Instructors who wish to introduce data structures at an earlier point in the course, without having first covered advanced C++ and OOP features, can cover Chapter 17 (Linked Lists), followed by Chapters 18 and 19 (Stacks & Queues and Binary Trees), any time after Chapter 14 (Recursion). In this case it is necessary to simply omit the sections in Chapters 17–19 that deal with templates and the Standard Template Library.

Figure P-1



Brief Overview of Each Chapter

Chapter 1: Introduction to Computers and Programming

This chapter provides an introduction to the field of computer science and covers the fundamentals of hardware, software, operating systems, programming, problem solving, and software engineering. The components of programs, such as key words, variables, operators, and punctuation are covered. The tools of the trade, such as hierarchy charts and pseudocode, are also presented. The new *Tying It All Together* section shows students how to use the `cout` statement to create a personalized output message. Two new Programming Challenges help students see how the same basic input, processing, and output structure can be used to create multiple programs.

Chapter 2: Introduction to C++

This chapter gets the student started in C++ by introducing the basic parts of a C++ program, data types, variable definitions, assignment statements, constants, comments, program output, and simple arithmetic operations. The C++ string class is presented and string objects are used from this point on in the book as the primary method of handling strings. Programming style conventions are introduced and good programming style is modeled here, as it is throughout the text. An optional section explains the difference between ANSI standard and prestandard C++ programs. The new *Tying It All Together* section lets the student play with simple text-based graphics.

Chapter 3: Expressions and Interactivity

In this chapter the student learns to write programs that input and handle numeric, character, and string data. The use of arithmetic operators and the creation of mathematical expressions are covered, with emphasis on operator precedence. Debugging is introduced, with a section on hand tracing a program. Sections are also included on using random numbers, on reading and writing sequential files, on simple output formatting, on data type conversion and type casting, and on using library functions that work with numbers. For those who wish to cover them, there is also a section on C-strings. The new *Tying It All Together* section shows students how to create a simple interactive word game.

Chapter 4: Making Decisions

Here the student learns about relational expressions and how to control the flow of a program with the `if`, `if/else`, and `if/else if` statements. Logical operators, the conditional operator, and the `switch` statement are also covered. Applications of these constructs, such as menu-driven programs, are illustrated. This chapter also continues the theme of debugging with a section on validating output results. The new *Tying It All Together* section uses random numbers and branching statements to create a fortune telling game.

Chapter 5: Looping

This chapter covers C++'s repetitive control mechanisms. The `while` loop, `do-while` loop, and `for` loop are taught, along with a variety of methods to control them. These include using counters, user input, end sentinels, and end-of-file testing. Applications utilizing loops, such as keeping a running total and performing data validation, are covered. The emphasis on testing and debugging continues, with a section on creating good test data. The new *Tying It All Together* section introduces students to Windows commands to create colorful output and uses a loop to create a multi-colored display.

Chapter 6: Functions

In this chapter the student learns how and why to modularize programs, using both `void` and value-returning functions. Parameter passing is covered, with emphasis on when arguments should be passed by value versus when they need to be passed by reference. Scope of variables is covered and sections are provided on local versus global variables and on static local variables. Overloaded functions are also introduced and demonstrated. The new *Tying It All Together* section includes a modular, menu-driven program that emphasizes the versatility of functions, illustrating how their behavior can be controlled by the arguments sent to them.

Chapter 7: Introduction to Classes and Objects

In this chapter the text begins to focus on the object-oriented paradigm. Students learn how to define classes and to create and use objects. Careful attention is paid to illustrating which functions belong in a class versus which functions belong in a client program that uses the class. Good object-oriented practices are discussed and modeled, such as protecting member data through carefully constructed accessor and mutator functions and hiding class implementation details from client programs. Once students are comfortable working with classes and objects, the chapter provides a brief introduction to the topic of object-oriented analysis and design. The chapter also introduces structures and uses them in this chapter's *Tying It All Together* section, where students learn to use screen control techniques to create an animation that simulates the motion of a yoyo.

Chapter 8: Arrays

In this chapter the student learns to create and work with single and multidimensional arrays. Many examples of array processing are provided, including functions to compute the sum, average, highest and lowest values in an array. Students also learn to create tables using two-dimensional arrays, and to analyze the array data by row or by column. Programming techniques using parallel arrays are also demonstrated, and the student is shown how to use a data file as an input source to populate an array. STL vectors are introduced and compared to arrays. Sections on arrays of objects and structures are located at the end of the chapter, so they can be covered now or saved for later if the instructor wishes to cover this chapter before Chapter 7. The new *Tying It All Together* section uses arrays to create a game of *Rock*, *Paper*, *Scissors* between a human player and the computer.

Chapter 9: Searching, Sorting, and Algorithm Analysis

Here the student learns the basics of searching for information stored in arrays and of sorting arrays, including arrays of objects. The chapter covers the Linear Search, Binary Search, Bubble Sort, and Selection Sort algorithms, and has an optional section on sorting and searching STL vectors. A brief introduction to algorithm analysis is included and students are shown how to determine which of two algorithms is more efficient. The new *Tying It All Together* section uses both a table lookup and a searching algorithm to encode and decode secret messages.

Chapter 10: Pointers

This chapter explains how to use pointers. The topics include pointer arithmetic, initialization of pointers, comparison of pointers, pointers and arrays, pointers and functions, dynamic memory allocation, and more. The new *Tying It All Together* section demonstrates the use of pointers to access library data structures and functions that return calendar and wall clock time.

Chapter 11: More about Classes and Object-Oriented Programming

This chapter continues the study of classes and object-oriented programming. It covers object aggregation and composition, as well as inheritance, and illustrates the difference between is-a and has-a relations. Constant member functions, static members, friends, memberwise assignment, copy constructors, object type conversion operators, convert constructors, and a newly rewritten section on operator overloading are also included. The new *Tying It All Together* section brings together the concepts of inheritance and convert constructors to build a program that formats the contents of an array to form an HTML table for display on a Web site.

Chapter 12: More about Characters, Strings, and the `string` Class

This chapter covers standard library functions for working with characters and C-strings, covering topics such as passing C-strings to functions and using the C++ `sstream` classes to convert between numeric and string forms of numbers. Additional material about the C++ string class and its member functions and operators is presented and a new, improved program illustrates how to write your own string class. The new *Tying It All Together* section shows students how to access string-based program environments to obtain information about the computer and the network on which the program is running.

Chapter 13: Advanced File and I/O Operations

This chapter covers sequential access, random access, text, and binary files. Various modes for opening files are discussed, as well as the many methods for reading and writing file contents. Advanced output formatting is also covered. The new *Tying It All Together* program applies many of the techniques covered in the chapter to merge two text files into an HTML document for display on the Web, with different colors used to illustrate which file each piece of data came from.

Chapter 14: Recursion

In this chapter recursion is defined and demonstrated. A visual trace of recursive calls is provided, and recursive applications are discussed. Many recursive algorithms are presented, including recursive functions for computing factorials, finding a greatest common denominator (GCD), performing a binary search, sorting QuickSort, and solving the famous Towers of Hanoi problem. For students who need more challenge, there is a section on exhaustive and enumeration algorithms. The new *Tying It All Together* section uses recursion to evaluate prefix expressions.

Chapter 15: Polymorphism and Virtual Functions

The study of classes and object-oriented programming continues in this chapter with the introduction of more advanced concepts such as polymorphism and virtual functions. Information is also presented on abstract base classes, pure virtual functions, type compatibility within an inheritance hierarchy, and virtual inheritance. The material on multiple inheritance previously in the chapter has been rewritten and moved to an appendix. The new *Tying It All Together* section illustrates the use of inheritance and polymorphism to display and animate graphical images.

Chapter 16: Exceptions, Templates, and the Standard Template Library (STL)

Here the student learns to develop enhanced error trapping techniques using exceptions. Discussion then turns to function and class templates as a method for writing generic code. Finally, the student is introduced to the containers, iterators, and algorithms offered by the Standard Template Library (STL). The new *Tying It All Together* section uses various containers in the Standard Template Library to create an educational children's game.

Chapter 17: Linked Lists

This chapter introduces concepts and techniques needed to work with lists. A linked list ADT is developed and the student is taught to code operations such as creating a linked list, appending a node, traversing the list, searching for a node, inserting a node, deleting a node, and destroying a list. A linked list class template is also demonstrated. The new *Tying It All Together* section brings together many of the most important concepts of OOP by using objects, inheritance, and polymorphism to animate a collection of images.

Chapter 18: Stacks and Queues

In this chapter the student learns to create and use static and dynamic stacks and queues. The operations of stacks and queues are defined, and templates for each ADT are demonstrated. The static array-based stack uses exception-handling to handle stack overflow and underflow, providing a realistic and natural example of defining, throwing, and catching exceptions. The new *Tying It All Together* section discusses strategies for evaluating postfix expressions and for converting them to infix.

Chapter 19: Binary Trees

This chapter covers the binary tree ADT and demonstrates many binary tree operations. The student learns to traverse a tree, insert an element, delete an element, replace an element, test for an element, and destroy a tree. The new *Tying It All Together* section introduces a tree structure versatile enough to create genealogy trees.

Appendices

Appendix A: The ASCII Character Set A list of the ASCII and extended ASCII characters and their codes.

Appendix B: Operator Precedence and Associativity A list of the C++ operators with their precedence and associativity.

The following appendices are on the accompanying student CD

Appendix C: A Brief Introduction to Object-Oriented Programming An introduction to the concepts and terminology of object-oriented programming.

Appendix D: Using UML in Class Design A brief introduction to the Unified Modeling Language (UML) class diagrams with examples of their use.

Appendix E: Namespaces An explanation of namespaces and their purpose, with examples provided on how to define a namespace and access its members.

Appendix F: Passing Command Line Arguments An introduction to writing C++ programs that accept command-line arguments. This appendix will be useful to students working in a command-line environment, such as UNIX or Linux.

Appendix G: Header File and Library Function Reference A reference for the C++ library functions and header files used in the book.

Appendix H: Binary Numbers and Bitwise Operations A guide to the binary number system and the C++ bitwise operators, as well as a tutorial on the internal storage of integers.

Appendix I: C++ Casts and Run-Time Type Identification An introduction to the different ways of doing type casting in C++ and to run-time type identification.

Appendix J: Multi-Source File Programs A tutorial on how to create, compile, and link programs with multiple source files. Includes the use of function header files, class specification files, and class implementation files.

Appendix K: Multiple and Virtual Inheritance A self-contained discussion of the C++ concepts of multiple and virtual inheritance for anyone already familiar with single inheritance.

Appendix L: Introduction to the MinGW C++ Compiler and the wxDev-C++ IDE A tutorial on how to start a wxDev-C++ project, compile and run a program, save source files, and more.

Appendix M: Introduction to Microsoft Visual C++ 2008 Express Edition A tutorial on how to start a project using Microsoft Visual C++ 2008, compile and run a program, save source files, and more.

Appendix N: .NET and Managed C++ A short introduction to Microsoft .NET and managed C++.

Appendix O: Introduction to Flowcharting A tutorial that introduces flowcharting and its symbols. Includes handling sequence, selection, case, repetition, and calls to other modules. Sample flowcharts for several of the book's example programs are presented.

Appendix P: Answers to Checkpoints A tool students can use to assess their understanding by comparing their answers to the Checkpoint exercises found throughout the book. The answers to all Checkpoint exercises are included.

Appendix Q: Answers to Odd-Numbered Review Questions Another tool students can use to gauge their understanding and progress.

Features of the Text

Concept Statements — Each major section of the text starts with a concept statement. This statement summarizes the key idea of the section.

Example Programs — The text has over 350 complete example programs, each designed to highlight the topic currently being studied. In most cases, these are practical, real-world examples. Source code for these programs is provided so that students can run the programs themselves.

Program Output — After each example program there is a sample of its screen output. This immediately shows the student how the program should function.

Tying It All Together — This special section, found at the end of every chapter, shows the student how to do something clever and fun with the material covered in that chapter.



VideoNotes — A series of online videos, developed specifically for this book, are available for viewing at `http://www.pearsonhighered.com/gaddis/`. *VideoNotes* icons appear throughout the text, alerting the student to videos about specific topics.



Checkpoints — Checkpoints are questions placed throughout each chapter as a self-test study aid. Answers for all Checkpoint questions are provided on the student CD so students can check how well they have learned a new topic.



Notes — Notes appear at appropriate places throughout the text. They are short explanations of interesting or often misunderstood points relevant to the topic at hand.



Warnings — Warnings caution the student about certain C++ features, programming techniques, or practices that can lead to malfunctioning programs or lost data.

Case Studies — Case studies that simulate real-world applications appear in many chapters throughout the text, with complete code provided for each one. Additional case studies are provided on the student CD. These case studies are designed to highlight the major topics of the chapter in which they appear.

Review Questions and Exercises — Each chapter presents a thorough and diverse set of review questions, such as fill-in-the-blank and short answer, that check the student's mastery of the basic material presented in the chapter. These are followed by exercises requiring problem solving and analysis, such as the *Algorithm Workbench*, *Predict the Output*, and *Find the Errors* sections. Each chapter ends with a *Soft Skills* exercise that focuses on communication and group process skills. Answers to the odd numbered review questions and review exercises are provided on the student CD.

Programming Challenges — Each chapter offers a pool of programming exercises designed to solidify the student's knowledge of the topics currently being studied. In most cases the assignments present real-world problems to be solved. When applicable, these exercises include input validation rules.

Group Projects	There are several group programming projects throughout the text, intended to be constructed by a team of students. One student might build the program's user interface, while another student writes the mathematical code, and another designs and implements a class the program uses. This process is similar to the way many professional programs are written and encourages team work within the classroom.
C++ Quick Reference Guide	For easy access, a quick reference guide to the C++ language is printed on the inside front and back covers.

Supplements

Student CD

This CD includes:

- MinGW C++ Compiler
- wxDev-C++ IDE
- Answers to all Checkpoint questions (Appendix P)
- Answers to all odd-numbered Review Questions and Exercises (Appendix Q)
- Complete source code for every program included in the book
- Additional case studies, complete with source code
- Serendipity Booksellers ongoing software development project
- A full set of appendices (including several tutorials) that accompany the book

If a CD did not come with your book or you can't locate your CD, you can access most of these items at `http://www.pearsonhighered.com/cssupport`

Other CDs Upon Request Professors should contact their campus Pearson Education/Addison-Wesley representative for the specific ISBN to order this book packaged with Microsoft Visual C++.

MyCodeMate—Your Own T.A. Just a Click Away

Addison-Wesley's *MyCodeMate* is a book-specific Web resource that provides tutorial help and evaluation of student programs. Example programs throughout the book and selected Programming Challenges from every chapter have been integrated into *MyCodeMate*. Using this tool, a student is able to write and compile programs from any computer with Internet access and receive guidance and feedback on how to proceed and on how to address compiler error messages. Instructors can track each student's progress on Programming Challenges from the text or can develop projects of their own. **A complimentary subscription to *MyCodeMate* is offered when the access code is ordered in a package with a new copy of this text.** Subscriptions can also be purchased online. For more information visit www.mycodemate.com, or contact your campus Pearson Education/Addison-Wesley representative.

Instructor Resources

The following supplements are available to qualified instructors only.

- Answers to all Review Questions in the text
- Solutions for all Programming Challenges in the text
- PowerPoint presentation slides for every chapter

- A computerized test bank
- A collection of lab materials
- Source code files

Visit the Pearson Education Instructor Resource Center (`http://www.pearsonhighered.com/irc`) or send an email to computing@aw.com for information on how to access them.

Textbook Web Site

A Web site for the *Starting Out with* C++ series of books is located at the following URL:

```
http://www.pearsonhighered.com/gaddis/
```

Get This Book the Way You Want It!

This book is part of the Pearson Custom Computer Science Library. Use our online `PubSelect` system to select just the chapters you need from this, and other, Pearson Education CS textbooks. You can edit the sequence to exactly match your course organization and teaching approach. Visit `www.pearsoncustom.com/cs` for details.

Which Gaddis C++ Book Is Right for You?

The *Starting Out with* C++ Series includes three books, one of which is sure to fit your course:

- *Starting Out with C++: Control Structures through Objects;*
- *Starting Out with C++: Early Objects;*
- *Starting Out with C++: Brief Version.*

The following chart will help you determine which book is right for your course.

▪ FROM CONTROL STRUCTURES THROUGH OBJECTS ▪ BRIEF VERSION	▪ EARLY OBJECTS
LATE INTRODUCTION OF OBJECTS Classes are introduced in Chapter 13 of the standard text and Chapter 11 of the brief text, after control structures, functions, arrays, and pointers. Advanced OOP topics, such as inheritance and polymorphism, are covered in the following two chapters.	**EARLIER INTRODUCTION OF OBJECTS** Classes are introduced in Chapter 7, after control structures and functions, but before arrays and pointers. Their use is then integrated into the remainder of the text. Advanced OOP topics, such as inheritance and polymorphism, are covered in Chapters 11 and 15.
USE OF C-STRINGS Null-terminated C-strings are used throughout, with the C++ `string` class covered briefly.	**USE OF `string` OBJECTS** Standard library `string` class objects are used throughout, with C-strings covered briefly.
INTRODUCTION OF DATA STRUCTURES AND RECURSION Linked lists, stacks and queues, and binary trees are introduced in the final chapters of the standard text. Recursion is covered after stacks and queues, but before binary trees. These topics are not covered in the brief text, though it does have appendices dealing with linked lists and recursion.	**INTRODUCTION OF DATA STRUCTURES AND RECURSION** Linked lists, stacks and queues, and binary trees are introduced in the final chapters of the text, after the chapter on recursion.

Acknowledgments

There have been many helping hands in the development and publication of this text. We would like to thank the following faculty reviewers for their helpful suggestions and expertise.

Reviewers of the Seventh Edition or Its Previous Versions

Ahmad Abuhejleh
University of Wisconsin, River Falls

David Akins
El Camino College

Steve Allan
Utah State University

Ijaz A. Awan
Savannah State University

John Bierbauer
North Central College

Don Biggerstaff
Fayetteville Technical Community College

Paul Bladek
Spokane Falls Community College

Chuck Boehm
Dean Foods, Inc.

Bill Brown
Pikes Peak Community College

Richard Cacace
Pensacola Junior College

Randy Campbell
Morningside College

Stephen P. Carl
Wright State University

Wayne Caruolo
Red Rocks Community College

Thomas Cheatham
Middle Tennessee State University

James Chegwidden
Tarrant County College

John Cigas
Rockhurst University

John Cross
Indiana University of Pennsylvania

Joseph DeLibero
Arizona State University

Dennis Fairclough
Utah Valley State College

Larry Farrer
Guilford Technical Community College

Richard Flint
North Central College

Sheila Foster
California State University Long Beach

David E. Fox
American River College

Cindy Fry
Baylor University

Peter Gacs
Boston University

Cristi Gale
Sterling College

James Gifford
University of Wisconsin, Stevens Point

Leon Gleiberman
Touro College

Simon Gray
Ashland University—Ohio

Margaret E. Guertin
Tufts University

Jamshid Haghighi
Guilford Technical Community College

Ranette H. Halverson
Midwestern State University, Wichita Falls, TX

Dennis Heckman
Portland Community College

Ric Heishman
Northern Virginia Community College

Patricia Hines
Brookdale Community College

Mike Holland
Northern Virginia Community College

Lister Wayne Horn
Pensacola Junior College

Richard Hull
Lenoir-Rhyne College

Norman Jacobson
University of California, Irvine

Eric Jiang
San Diego State University

David Kaeli
Northeastern University

Chris Kardaras
North Central College

Eugene Katzen
Montgomery College—Rockville

Willard Keeling
Blue Ridge Community College

A. J. Krygeris
Houston Community College

Ray Larson
Inver Hills Community College

Stephen Leach
Florida State University

Parkay Louie
Houston Community College

Zhu-qu Lu
University of Maine, Presque Isle

Tucjer Maney
George Mason University

Bill Martin
Central Piedmont Community College

Debbie Mathews
J. Sargeant Reynolds

Ron McCarty
Penn State Erie, The Behrend College

Robert McDonald
East Stroudsburg University

James McGuffee
Austin Community College

M. Dee Medley
Augusta State University

Cathi Chambley-Miller
Aiken Technical College

Sandeep Mitra
SUNY Brockport

Frank Paiano
Southwestern Community College

Theresa Park
Texas State Technical College

Mark Parker
Shoreline Community College

Robert Plantz
Sonoma State University

Tino Posillico
SUNY Farmingdale

M. Padmaja Rao
Francis Marion University

Timothy Reeves
San Juan College

Ronald Robison
Arkansas Tech University

Caroline St. Clair
North Central College

Dolly Samson
Weber State University

Kate Sanders
Rhode Island College

Lalchand Shimpi
Saint Augustine's College

Sung Shin
South Dakota State University

Garth Sorenson
Snow College

Daniel Spiegel
Kutztown University

Ray Springston
University of Texas at Arlington

Kirk Stephens
Southwestern Community College

Cherie Stevens
South Florida Community College

Hong Sung
University of Central Oklahoma

Mark Swanson
Red Wing Technical College

Martha Tillman
College of San Mateo

Delores Tull
Itawamba Community College

Rober Tureman
Paul D. Camp Community College

Jane Turk
LaSalle University

Sylvia Unwin
Bellevue Community College

Stewart Venit
California State University, Los Angeles

Doug White
University of Northern Colorado

Chris Wild
Old Dominion University

Catherine Wyman
DeVry Institute of Technology, Phoenix

Sherali Zeadally
University of the District of Columbia

The authors would like to thank their students at Haywood Community College and North Central College for inspiring them to write student-friendly books. They would also like to thank their families for their tremendous support throughout this project, as well as North Central College for providing Prof. Walters and Muganda with the sabbatical term during which they worked on this book. An especially big thanks goes to our terrific editorial, production, and marketing team at Addison-Wesley. In particular we want to thank our editor Michael Hirsch and our production project manager Heather McNally, who have been instrumental in guiding the production of this book. We also want to thank our project manager, Peggy Kellar, who helped everything run smoothly, and our meticulous and knowledgable copyeditor, Evelyn Perricone, who dedicated many hours to making this book the best book it could be. You are great people to work with!

About the Authors

Tony Gaddis is the principal author of the *Starting Out With . . .* series of textbooks. He is a highly acclaimed instructor with twenty years of experience teaching computer science courses at Haywood Community College. Tony was previously selected as the North Carolina Community College "Teacher of the Year" and has received the Teaching Excellence award from the National Institute for Staff and Organizational Development. The *Starting Out With . . .* series includes introductory books covering C++, Java™, Microsoft® Visual Basic®, Microsoft® C#, and Alice, all published by Addison-Wesley.

Judy Walters is an Associate Professor of Computer Science at North Central College in Naperville, Illinois. In addition to her many computer science courses, she also teaches two film-related courses she developed for the college's interdisciplinary freshman seminar program. She recently returned from her second semester teaching in Costa Rica, where she hopes to retire some day.

Godfrey Muganda is an Associate Professor of Computer Science at North Central College. He teaches a wide variety of courses at both the undergraduate and graduate levels, including courses in Algorithms, Computer Organization, Web Applications, and Web Services. His primary research interests are in the area of Fuzzy Sets and Systems.

CHAPTER 1

Introduction to Computers and Programming

TOPICS

1.1 Why Program?
1.2 Computer Systems: Hardware and Software
1.3 Programs and Programming Languages
1.4 What Is a Program Made of?
1.5 Input, Processing, and Output
1.6 The Programming Process
1.7 Tying It All Together: *Hi! It's Me*

1.1 Why Program?

CONCEPT: Computers can do many different jobs because they are programmable.

Every profession has tools that make its job easier to do. Carpenters use hammers, saws, and measuring tapes. Mechanics use wrenches, screwdrivers, and ratchets. Electronics technicians use probes, scopes, and meters. Some tools are unique and can be categorized as belonging to a single profession. For example, surgeons have certain tools that are designed specifically for surgical operations. Those tools probably aren't used by anyone other than surgeons. There are some tools, however, that are used in several professions. Screwdrivers, for instance, are used by mechanics, carpenters, and many others.

The computer is a tool that is used by so many professions, it cannot be easily categorized. It can perform so many different jobs that it is perhaps the most versatile tool ever made. For the accountant, computers balance books, analyze profits and losses, and prepare tax reports. For the factory worker, computers control manufacturing machines and track production. For the mechanic, computers analyze the various systems in an automobile and pinpoint hard-to-find problems.

What makes the computer so useful? Quite simply, the computer can do such a wide variety of tasks because it can be *programmed*. It is a machine specifically designed to follow instructions.

Because of the computer's programmability, it doesn't belong to any single profession. Computers are designed to do whatever job their programs, or *software*, tell them to do.

Computer programmers do a very important job. They create software that transforms computers into the specialized tools of many trades. Without programmers, the users of computers would have no software, and without software, computers would not be able to do anything.

Computer programming is both an art and a science. It is an art because every aspect of a program should be designed with care and judgment. Listed below are a few of the things that must be designed for any real-world computer program:

- The logical flow of the instructions
- The mathematical procedures
- The appearance of the screens
- The way information is presented to the user
- The program's "user-friendliness"
- Manuals and other forms of written documentation

There is also a scientific, or engineering side to programming. Because programs rarely work right the first time they are written, a lot of experimentation, correction, and redesigning is required. This demands patience and persistence of the programmer. Writing software demands discipline as well. Programmers must learn special languages like C++ because computers do not understand English or other human languages. Languages such as C++ have strict rules that must be carefully followed.

Both the artistic and scientific nature of programming makes writing computer software like designing a car. Both cars and programs should be functional, efficient, powerful, easy to use, and pleasing to look at.

1.2 Computer Systems: Hardware and Software

CONCEPT: All computer systems consist of similar hardware devices and software components. This section provides an overview of standard computer hardware and software organization.

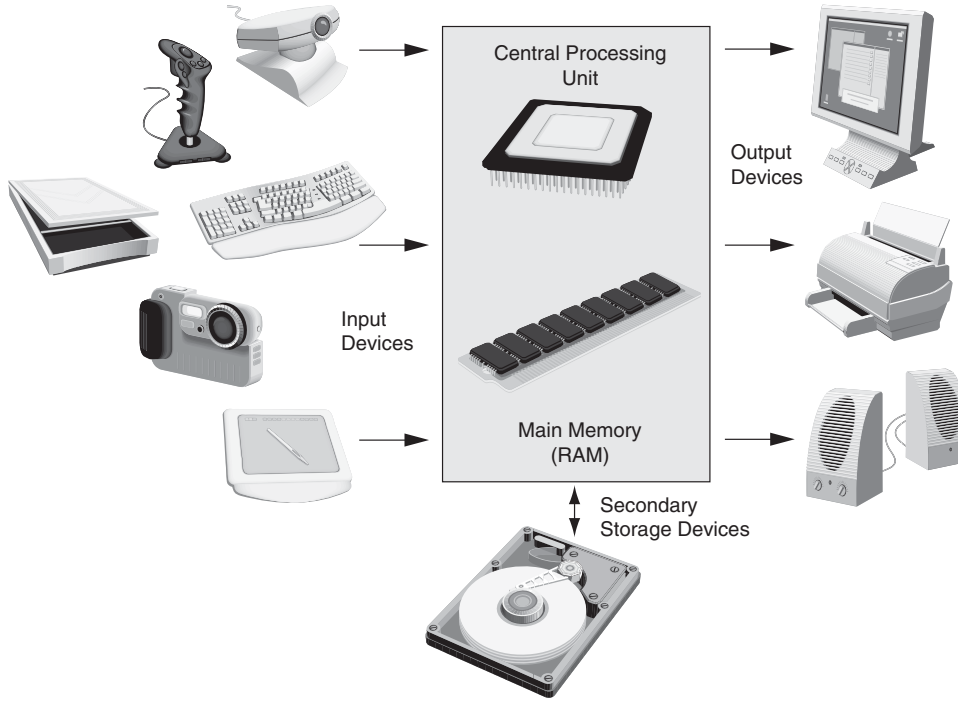
Hardware

Hardware refers to the physical components that a computer is made of. A computer, as we generally think of it, is not an individual device, but a system of devices. Like the instruments in a symphony orchestra, each device plays its own part. A typical computer system consists of the following major components:

1. The central processing unit (CPU)
2. Main memory (random-access memory, or RAM)
3. Secondary storage devices
4. Input devices
5. Output devices

The organization of a computer system is depicted in Figure 1-1.

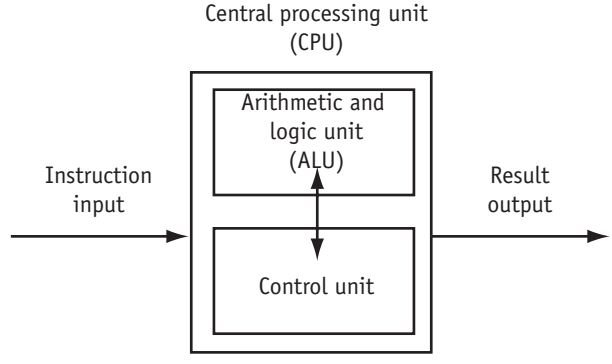
Figure 1-1



The CPU

At the heart of a computer is its *central processing unit*, or *CPU*. The CPU's job is to fetch instructions, follow the instructions, and produce some result. Internally, the central processing unit consists of two parts: the *control unit* and the *arithmetic and logic unit (ALU)*. The control unit coordinates all of the computer's operations. It is responsible for determining where to get the next instruction and regulating the other major components of the computer with control signals. The arithmetic and logic unit, as its name suggests, is designed to perform mathematical operations. The organization of the CPU is shown in Figure 1-2.

Figure 1-2



A program is a sequence of instructions stored in the computer's memory. When a computer is running a program, the CPU is engaged in a process known formally as the *fetch/decode/execute cycle*. The steps in the fetch/decode/execute cycle are as follows:

Fetch The CPU's control unit fetches, from main memory, the next instruction in the sequence of program instructions.

Decode The instruction is encoded in the form of a number. The control unit decodes the instruction and generates an electronic signal.

Execute The signal is routed to the appropriate component of the computer (such as the ALU, a disk drive, or some other device). The signal causes the component to perform an operation.

These steps are repeated as long as there are instructions to perform.

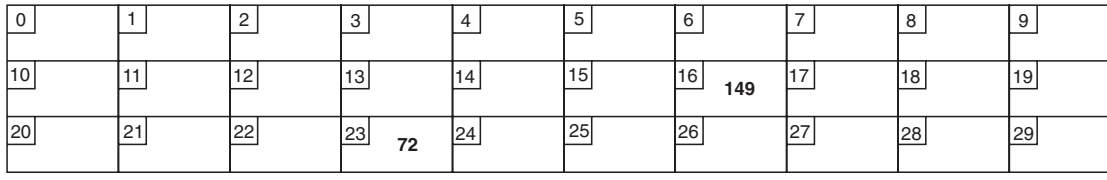
Main Memory

Commonly known as *random-access memory*, or *RAM*, the computer's main memory is a device that holds information. Specifically, RAM holds the sequences of instructions in the programs that are running and the data those programs are using.

Memory is divided into sections, or cells, that each hold an equal amount of data. Each cell typically contains eight "switches" that may be either on or off. A switch that is in the on position usually represents the number 1, while a switch in the off position usually represents the number 0. The computer stores data by setting the switches in a memory cell to a pattern that represents a piece of information. Each of these switches is known as a *bit*, which stands for *binary digit*. Each cell, which is a collection of eight bits, is known as a *byte*.

Bytes are grouped together to make *words*. On most computers a word contains four bytes. Each word is assigned a unique number known as an *address*. The addresses are ordered from lowest to highest. A word is identified by its address in much the same way a post office box is identified by an address. Figure 1-3 shows a group of memory words with their addresses. In the illustration, sample data is stored in memory. The number 149 is stored in the word with the address 16, and the number 72 is stored at address 23.

Figure 1-3



RAM is usually a volatile type of memory, used only for temporary storage. When the computer is turned off, the contents of RAM are erased.

Secondary Storage

Secondary storage is a type of memory that can hold data for long periods of time—even when there is no power to the computer. Frequently used programs are stored in secondary memory and loaded into main memory as needed. Important information, such as

word processing documents, payroll data, and inventory figures, is saved to secondary storage as well.

The most common type of secondary storage device is the disk drive. A *disk drive* stores data by magnetically encoding it onto a circular disk. Most computers have a disk drive mounted inside their case. External disk drives, which connect to one of the computer's communication ports, are also available. External disk drives can be used to create backup copies of important data or to move data to another computer.

In addition to external disk drives, many types of devices have been created for copying data and for moving it to other computers. For many years floppy disk drives were popular. A *floppy disk drive* records data onto a small, flexible ("floppy") disk, which can be removed from the drive. The use of floppy disk drives has declined dramatically in recent years, in favor of superior devices such as USB flash drives. *USB flash drives* are small devices that plug into the computer's USB (universal serial bus) port and appear to the system as a disk drive. These drives, which use *flash memory* to store data, are inexpensive, reliable, and small enough to be carried in your pocket.

Optical devices such as the *CD* (compact disc) and the *DVD* (digital versatile disc) are also popular for data storage. Data is not recorded magnetically on an optical disc, but rather is encoded as a series of pits on the disc surface. CD and DVD drives use a laser to detect the pits and thus read the encoded data. Optical discs hold large amounts of data, and because recordable CD and DVD drives are now commonplace, they are good media for creating backup copies of data.

Input Devices

Input is any information the computer collects from the outside world. The device that collects the information and sends it to the computer is called an *input device*. Common input devices are the keyboard, mouse, scanner, digital camera, and microphone. Disk drives, CD/DVD drives, and USB flash drives can also be considered input devices because programs and information are retrieved from them and loaded into the computer's memory.

Output Devices

Output is any information the computer sends to the outside world. It might be a sales report, a list of names, or a graphic image. The information is sent to an *output device,* which formats and presents it. Common output devices are computer screens, printers, and speakers. Output sent to a computer screen is sometimes called *soft copy,* while output sent to a printer is called *hard copy.* Disk drives, USB flash drives, and CD/DVD recorders can also be considered output devices because the CPU sends information to them so it can be saved.

Software

As previously mentioned, software refers to the programs that run on a computer. There are two general categories of software: *operating systems* and *application software*. An operating system is a set of programs that manages the computer's hardware devices and controls their processes. Operating systems fall into one of the following categories.

Single tasking A single tasking operating system is capable of running only one program at a time. The computer devotes all its hardware resources and CPU time to each program as it executes. MS-DOS is an example of a single tasking operating system.

Multitasking A multitasking operating system is capable of running multiple programs at once. Through a technique called *time sharing*, the system divides the allocation of hardware resources and the attention of the CPU among all the executing programs. UNIX, Windows XP, and Windows Vista are multitasking operating systems.

In addition, operating systems fall into one of the following categories, which describe the number of users they can accommodate.

Single user This type of system allows only one user to operate the computer at a time. MS-DOS and older versions of Windows are single user operating systems.

Multiuser Multiuser systems allow several users to run programs and operate the computer at once. Most variations of the UNIX operating system are multiuser systems.

Application software refers to programs that make the computer useful to the user. These programs solve specific problems or perform general operations that satisfy the needs of the user. Word processing, spreadsheet, and database programs are all examples of application software.



Checkpoint

1.1 Why is the computer used by so many different people, in so many different professions?

1.2 List the five major hardware components of a computer system.

1.3 Internally, the CPU consists of what two units?

1.4 Describe the steps in the fetch/decode/execute cycle.

1.5 What is a memory address?

1.6 Explain why computers have both main memory and secondary storage.

1.7 What are the two general categories of software?

1.8 What is the difference between a single tasking system and a multitasking system?

1.9 What is the difference between a single user system and a multiuser system?

1.3 Programs and Programming Languages

CONCEPT: A program is a set of instructions a computer follows in order to perform a task. A programming language is a special language used to write computer programs.

What Is a Program?

Computers are designed to follow instructions. A computer program is a set of instructions that tells the computer how to solve a problem or perform a task. For example, suppose we want the computer to calculate someone's gross pay. Here is a list of things the computer might do:

1. Display a message on the screen asking "How many hours did you work?"

2. Wait for the user to enter the number of hours worked. Once the user enters a number, store it in memory.
3. Display a message on the screen asking "How much do you get paid per hour?"
4. Wait for the user to enter an hourly pay rate. Once the user enters a number, store it in memory.
5. Multiply the number of hours by the amount paid per hour, and store the result in memory.
6. Display a message on the screen that tells the amount of money earned. The message must include the result of the calculation performed in step 5.

Collectively, these instructions are called an *algorithm*. An algorithm is a set of well-defined steps for performing a task or solving a problem. Notice these steps are sequentially ordered. Step 1 should be performed before step 2, and so forth. It is important that these instructions be performed in their proper sequence.

Although a person might easily understand the instructions in the pay-calculating algorithm, it is not ready to be executed on a computer. A computer's CPU can only process instructions that are written in *machine language*. A machine language program consists of a sequence of *binary numbers* (numbers consisting of only 1s and 0s) which the CPU interprets as commands. Here is an example of what a machine language instruction might look like:

```
1011010000000101
```

As you can imagine, the process of encoding an algorithm in machine language is very tedious and difficult. In addition, each different type of CPU has its own machine language. If you wrote a machine language program for computer A and then wanted to run it on a computer B that has a different type of CPU, you would have to rewrite the program in computer B's machine language.

Programming languages, which use words instead of numbers, were invented to ease the task of programming. A program can be written in a programming language such as C++, which is much easier to understand than machine language. Programmers save their programs in text files, and then use special software to convert their programs to machine language.

Program 1-1 shows how the pay-calculating algorithm might be written in C++.



NOTE: The line numbers shown in Program 1-1 are *not* part of the program. This book shows line numbers in all program listings to help point out specific parts of the program.

Program 1-1

```
1 // This program calculates the user's pay.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7    double hours, rate, pay;
8
```

(program continues)

Program 1-1 *(continued)*

```
 9    // Get the number of hours worked.
10    cout << "How many hours did you work? ";
11    cin  >> hours;
12
13    // Get the hourly pay rate.
14    cout << "How much do you get paid per hour? ";
15    cin  >> rate;
16
17    // Calculate the pay.
18    pay = hours * rate;
19
20    // Display the pay.
21    cout << "You have earned $" << pay << endl;
22    return 0;
23 }
```

Program Output with Example Input Shown in Bold

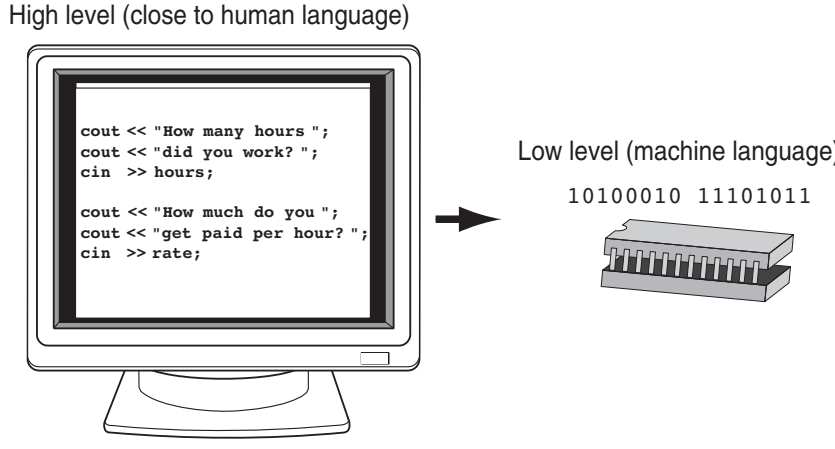
```
How many hours did you work? 10 [Enter]
How much do you get paid per hour? 15 [Enter]
You have earned $150
```

The "Program Output with Example Input Shown in Bold" shows what the program will display on the screen when it is running. In the example, the user enters 10 for the number of hours worked and 15 for the hourly pay. The program displays the earnings, which are $150.

Programming Languages

In a broad sense, there are two categories of programming languages: low-level and high-level. A *low-level language* is close to the level of the computer, which means it resembles the numeric machine language of the computer more than the natural language of humans. The easiest languages for people to learn are *high-level languages*. They are called "high-level" because they are closer to the level of human-readability than computer-readability. Figure 1-4 illustrates the concept of language levels.

Figure 1-4



Many high-level languages have been created. Table 1-1 lists a few of the well-known ones.

Table 1-1 Well-Known High-Level Programming Languages

Language	Description
BASIC	Beginners All-purpose Symbolic Instruction Code. A general programming language originally designed to be simple enough for beginners to learn.
C	A structured, general-purpose language developed at Bell Laboratories. C offers both high-level and low-level features.
C++	Based on the C language, C++ offers object-oriented features not found in C. Also invented at Bell Laboratories.
C#	Pronounced "C sharp." A language invented by Microsoft for developing applications based on the Microsoft .NET platform.
COBOL	Common Business-Oriented Language. A language designed for business applications.
FORTRAN	Formula Translator. A language designed for programming complex mathematical algorithms.
Java	An object-oriented language invented at Sun Microsystems. Java may be used to develop programs that run over the Internet in a Web browser.
JavaScript	A language used to write small programs that run in Web pages. Despite its name, JavaScript is not related to Java.
Pascal	A structured, general-purpose language designed primarily for teaching programming.
Python	A general purpose language created in the early 1990s. It has become popular for both business and academic applications.
Ruby	A general purpose language created in the 1990s. It is becoming increasingly popular for programs that run on Web servers.
Visual Basic	A Microsoft programming language and software development environment that allows programmers to quickly create Windows-based applications.

C++ is a widely used language because, in addition to the high-level features necessary for writing applications such as payroll systems and inventory programs, it also has many low-level features. C++ is based on the C language, which was invented for purposes such as writing operating systems and compilers. Because C++ evolved from C, it carries all of C's low-level capabilities with it.

C++ is also popular because of its *portability*. This means that a C++ program can be written on one type of computer and then run on many other types of systems. This usually requires

that the program is recompiled on each type of system, but the program itself may need little or no change.



NOTE: Programs written for specific graphical environments often require significant changes when moved to a different type of system. Examples of such graphical environments are Windows, the X-Window System, and the Mac OS X operating system.

Source Code, Object Code, and Executable Code

When a C++ program is written, it must be typed into the computer and saved to a file. A *text editor*, which is similar to a word processing program, is used for this task. The statements written by the programmer are called *source code*, and the file they are saved in is called the *source file*.

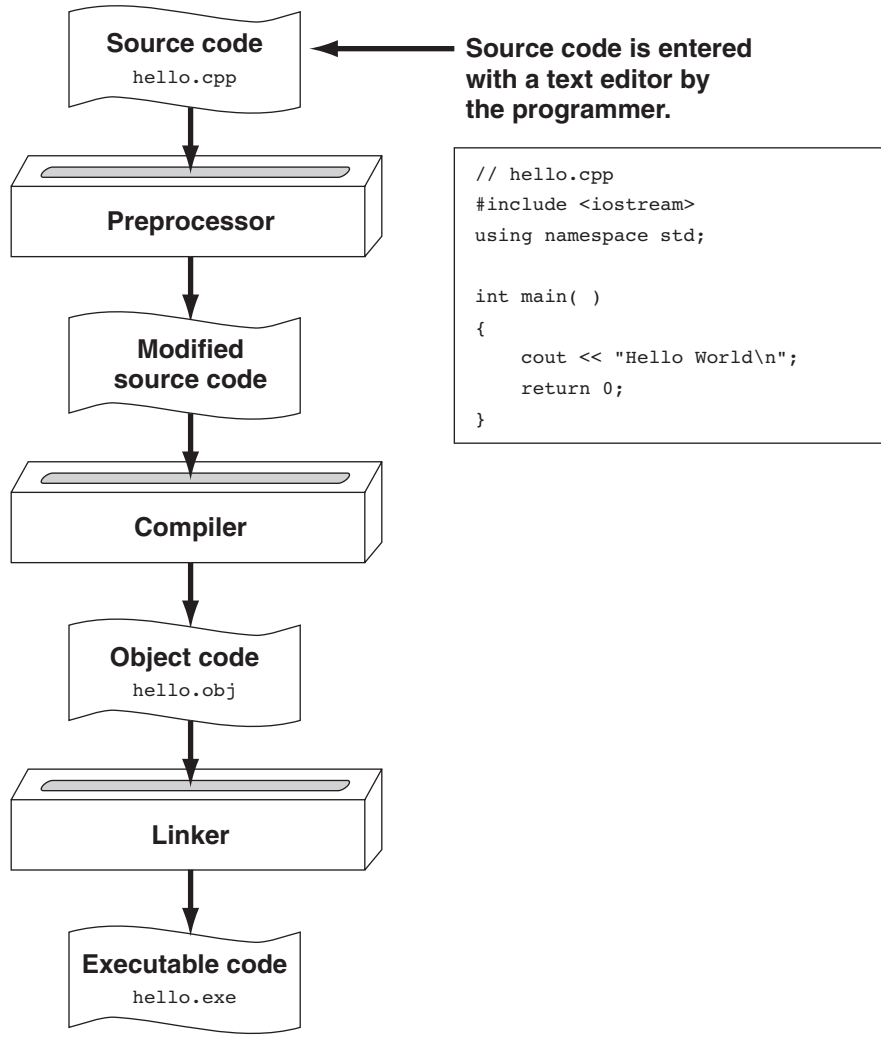
After the source code is saved to a file, the process of translating it to machine language can begin. During the first phase of this process, a program called the *preprocessor* reads the source code. The preprocessor searches for special lines that begin with the # symbol. These lines contain commands, or *directives*, that cause the preprocessor to amend or process the source code in some way. During the next phase the *compiler* steps through the preprocessed source code, translating each source code instruction into the appropriate machine language instruction. This process will uncover any *syntax errors* that may be in the program. Syntax errors are illegal uses of key words, operators, punctuation, and other language elements. If the program is free of syntax errors, the compiler stores the translated machine language instructions, which are called *object code*, in an *object file*.

Although an object file contains machine language instructions, it is not a complete program. Here is why. C++ is conveniently equipped with a library of prewritten code for performing common operations or sometimes-difficult tasks. For example, the library contains hardware-specific code for displaying messages on the screen and reading input from the keyboard. It also provides routines for mathematical functions, such as calculating the square root of a number. This collection of code, called the *run-time library*, is extensive. Programs almost always use some part of it. When the compiler generates an object file, however, it does not include machine code for any run-time library routines the programmer might have used. During the last phase of the translation process, another program called the *linker* combines the object file with the necessary library routines. Once the linker has finished with this step, an *executable file* is created. The executable file contains machine language instructions, or *executable code*, and is ready to run on the computer.

Figure 1-5 illustrates the process of translating a C++ source file into an executable file. The entire process of invoking the preprocessor, compiler, and linker can be initiated with a single action. For example, on a Linux system, the following command causes the C++ program named `hello.cpp` to be preprocessed, compiled, and linked. The executable code is stored in a file named `hello`.

```
g++ -o hello hello.cpp
```

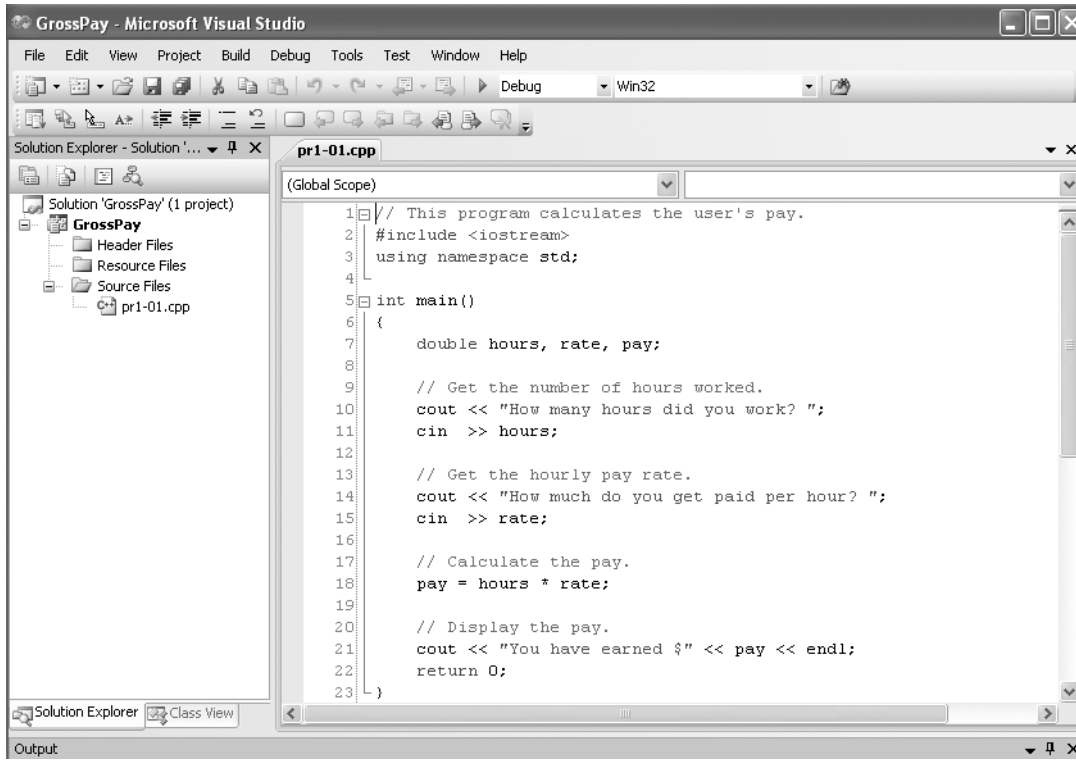
Figure 1-5



Appendix M on the student CD explains how compiling works in Microsoft Visual C++.

Many development systems, particularly those on personal computers, have *integrated development environments (IDEs)*. These environments consist of a text editor, compiler, debugger, and other utilities integrated into a package with a single set of menus. Preprocessing, compiling, linking, and even executing a program is done with a single click of a button, or by selecting a single item from a menu. Figure 1-6 shows a screen from the Microsoft Visual C++ IDE.

Figure 1-6



Checkpoint

1.10 What is an algorithm?

1.11 Why were computer programming languages invented?

1.12 What is the difference between a high-level language and a low-level language?

1.13 What does *portability* mean?

1.14 Explain the operations carried out by the preprocessor, compiler, and linker.

1.15 Explain what is stored in a source file, an object file, and an executable file.

1.16 What is an integrated development environment?

1.4 What Is a Program Made of?

CONCEPT: **There are certain elements that are common to all programming languages.**

Language Elements

All programming languages have a few things in common. Table 1-2 lists the common elements found in almost every language.

Table 1-2 Programming Language Elements

Language Element	Description
Key Words	Words that have a special meaning. Key words may only be used for their intended purpose. Key words are also known as reserved words.
Programmer-Defined Identifiers	Words or names defined by the programmer. They are symbolic names that refer to variables or programming routines.
Operators	Operators perform operations on one or more operands. An operand is usually a piece of data, like a number.
Punctuation	Punctuation characters that mark the beginning or ending of a statement, or separate items in a list.
Syntax	Rules that must be followed when constructing a program. Syntax dictates how key words and operators may be used, and where punctuation symbols must appear.

Let's look at some specific parts of Program 1-1 (the pay-calculating program) to see examples of each element listed in the table above. For convenience, Program 1-1 is listed again.

Program 1-1

```
1  // This program calculates the user's pay.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     double hours, rate, pay;
8
9     // Get the number of hours worked.
10    cout << "How many hours did you work? ";
11    cin  >> hours;
12
13    // Get the hourly pay rate.
14    cout << "How much do you get paid per hour? ";
15    cin  >> rate;
16
17    // Calculate the pay.
18    pay = hours * rate;
19
20    // Display the pay.
21    cout << "You have earned $" << pay << endl;
22    return 0;
23 }
```

Key Words (reserved words)

Three of C++'s key words appear on lines 3 and 5: `using`, `namespace`, and `int`. The word `double`, which appears on line 7, is also a C++ key word. These words, which are always written in lowercase, each have a special meaning in C++ and can only be used for their intended purposes. As you will see, the programmer is allowed to make up his or her own

names for certain things in a program. Key words, however, are reserved and cannot be used for anything other than their designated purposes. Part of learning a programming language is learning what the key words are, what they mean, and how to use them.



NOTE: The `#include <iostream>` statement in line 2 is a preprocessor directive.



NOTE: In C++, key words are written in all lowercase.

Programmer-Defined Identifiers

The words `hours`, `rate`, and `pay` that appear in the program on lines 7, 11, 15, 18, and 21 are programmer-defined identifiers. They are not part of the C++ language but rather are names made up by the programmer. In this particular program, these are the names of variables. As you will learn later in this chapter, variables are the names of memory locations that may hold data.

Operators

On line 18 the following statement appears:

```
pay = hours * rate;
```

The `=` and `*` symbols are both operators. They perform operations on pieces of data, known as operands. The `*` operator multiplies its two operands, which in this example are the variables `hours` and `rate`. The `=` symbol is called the *assignment operator*. It takes the value of the expression on the right and stores it in the variable whose name appears on the left. In this example, the `=` operator stores in the `pay` variable the result of the `hours` variable multiplied by the `rate` variable. In other words, the statement says, "Make the `pay` variable equal to `hours` times `rate`" or "`pay` is assigned the value of `hours` times `rate`."

Punctuation

Notice that many lines end with a semicolon. A semicolon in C++ is similar to a period in English. It marks the end of a complete sentence (or statement, as it is called in programming). Semicolons do not appear at the end of every line in a C++ program, however. There are rules that govern where semicolons are required and where they are not. Part of learning C++ is learning where to place semicolons and other punctuation symbols.

Lines and Statements

Often, the contents of a program are thought of in terms of lines and statements. A *line* is just that—a single line as it appears in the body of a program. Program 1-1 is shown with each of its lines numbered. Most of the lines contain something meaningful; however some of the lines are empty. The blank lines are only there to make the program more readable.

A *statement* is a complete instruction that causes the computer to perform some action. Here is the statement that appears in line 10 of Program 1-1:

```
cout << "How many hours did you work? ";
```

It causes the computer to display the message "How many hours did you work?" on the screen. Statements can be a combination of key words, operators, and programmer-defined symbols. Statements usually occupy only one line in a program, but sometimes they are spread out over more than one line.

Variables

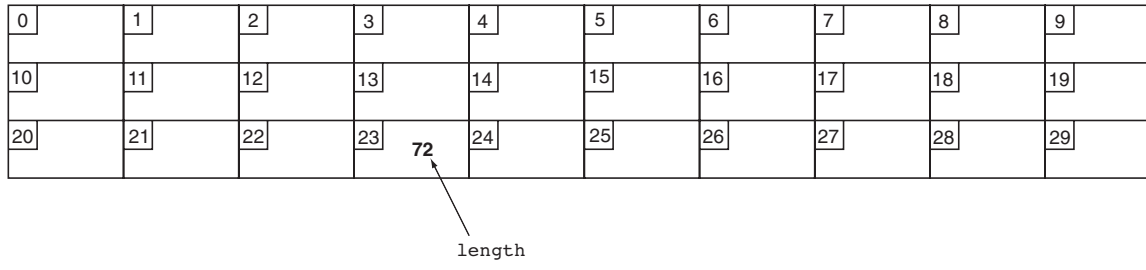
A *variable* is a named storage location in the computer's memory for holding a piece of data. The data stored in variables may change while the program is running (hence the name "variable"). Notice that in Program 1-1 the words `hours`, `rate`, and `pay` appear in several places. All three of these are the names of variables. The `hours` variable is used to store the number of hours the user worked. The `rate` variable stores the user's hourly pay rate. The `pay` variable holds the result of `hours` multiplied by `rate`, which is the user's gross pay.



NOTE: Notice the variables in Program 1-1 have names that reflect their purpose. In fact, it would be easy to guess what the variables were used for just by reading their names. This is discussed further in Chapter 2.

Variables are symbolic names that represent locations in the computer's random-access memory (RAM). When information is stored in a variable, it is actually stored in RAM. Assume a program has a variable named `length`. Figure 1-7 illustrates the way the variable name represents a memory location.

Figure 1-7



In Figure 1-7 the variable `length` is holding the value 72. The number 72 is actually stored in RAM at address 23, but the name `length` symbolically represents this storage location. You can think of a variable as a box that holds information. In Figure 1-7, the number 72 is stored in the box named `length`. Only one item may be stored in the box at any given time. If the program stores another value in the box, it will take the place of the number 72.

Variable Definitions

In programming, there are two general types of data: numbers, such as 3, and characters, such as the letter 'A'. Numbers are used to perform mathematical operations and characters are used to print information on the screen or on paper.

Numeric data can be categorized even further. For instance, the following are all whole numbers, or integers:

```
5
7
-129
32154
```

The following are real, or floating-point, numbers:

```
3.14159
6.7
1.0002
```

When creating a variable in a C++ program, you must know what type of data the program will be storing in it. Look at line 7 of Program 1-1:

```
double hours, rate, pay;
```

The word `double` in the statement indicates that the variables `hours`, `rate`, and `pay` will be used to hold double precision floating-point numbers. This statement is called a *variable definition*. In C++, all variables must be defined before they can be used because the variable definition is what causes the variables to be created in memory. If you review the listing of Program 1-1, you will see that the variable definitions come before any other statements using those variables.

1.5 Input, Processing, and Output

CONCEPT: The three primary activities of a program are input, processing, and output.

Computer programs typically perform a three-step process of gathering input, performing some process on the information gathered, and then producing output. Input is information a program collects from the outside world. It can be sent to the program by the user, who is entering data at the keyboard or using the mouse. It can also be read from disk files or hardware devices connected to the computer. Program 1-1 allows the user to enter two items of information: the number of hours worked and the hourly pay rate. Lines 11 and 15 use the `cin` (pronounced "see in") object to perform these input operations:

```
cin >> hours;
cin >> rate;
```

Once information is gathered from the outside world, a program usually processes it in some manner. In Program 1-1, the hours worked and hourly pay rate are multiplied in line 18 to produce the value assigned to the variable `pay`:

```
pay = hours * rate;
```

Output is information that a program sends to the outside world. It can be words or graphics displayed on a screen, a report sent to the printer, data stored in a file, or information sent to any device connected to the computer.

Lines 10, 14, and 21 in Program 1-1 all use the `cout` (pronounced "see out") object to display messages on the computer's screen.

```
cout << "How many hours did you work? ";
cout << "How much do you get paid per hour? ";
cout << "You have earned $" << pay << endl;
```

You will learn more about objects later in the book and about the `cin` and `cout` objects in Chapters 2 and 3.



Checkpoint

1.17 Describe the difference between a key word and a programmer-defined symbol.

1.18 Describe the difference between operators and punctuation symbols.

1.19 Describe the difference between a program line and a statement.

1.20 Why are variables called "variable"?

1.21 What happens to a variable's current contents when a new value is stored there?

1.22 What must take place in a program before a variable is used?

1.23 What are the three primary activities of a program?

1.6 The Programming Process

CONCEPT: The programming process consists of several steps, which include design, creation, testing, and debugging activities.

Designing and Creating a Program

Now that you have been introduced to what a program is, it's time to consider the process of creating a program. Quite often, when inexperienced students are given programming assignments, they have trouble getting started because they don't know what to do first. If you find yourself in this dilemma, the steps listed in Figure 1-8 may help. These are the steps recommended for the process of writing a program.

Figure 1-8

1. Define what the program is to do.
2. Visualize the program running on the computer.
3. Use design tools to create a model of the program.
4. Check the model for logical errors.
5. Write the program source code.
6. Compile the source code.
7. Correct any errors found during compilation.
8. Link the program to create an executable file.
9. Run the program using test data for input.
10. Correct any errors found while running the program. Repeat steps 4 through 10 as many times as necessary.
11. Validate the results of the program.

The steps listed in Figure 1-8 emphasize the importance of planning. Just as there are good ways and bad ways to paint a house, there are good ways and bad ways to create a program. A good program always begins with planning.

With the pay-calculating program as our example, let's look at each of the steps in more detail.

1. Define what the program is to do.

This step requires that you clearly identify the purpose of the program, the information that is to be input, the processing that is to take place, and the desired output. Here are the requirements for the example program:

Purpose	To calculate the user's gross pay.
Input	Number of hours worked, hourly pay rate.
Processing	Multiply number of hours worked by hourly pay rate. The result is the user's gross pay.
Output	Display a message indicating the user's gross pay.

2. Visualize the program running on the computer.

Before you create a program on the computer, you should first create it in your mind. Step 2 is the visualization of the program. Try to imagine what the computer screen looks like while the program is running. If it helps, draw pictures of the screen, with sample input and output, at various points in the program. For instance, here is the screen produced by the pay-calculating program:

```
How many hours did you work? 10
How much do you get paid per hour? 15
You earned $ 150
```

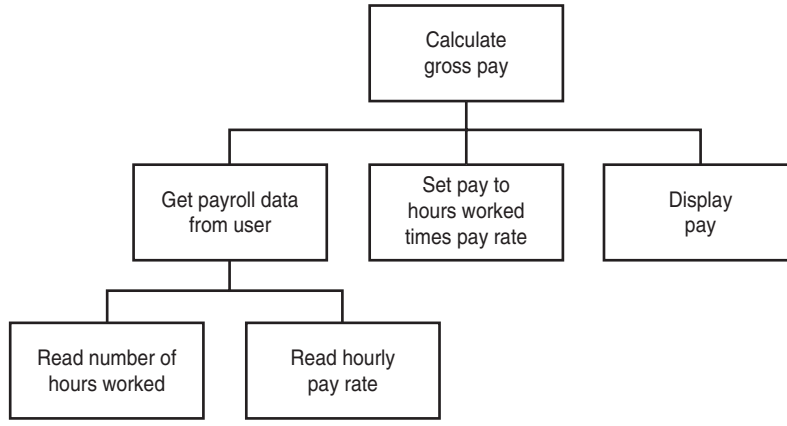
In this step, you must put yourself in the shoes of the user. What messages should the program display? What questions should it ask? By addressing these issues, you will have already determined most of the program's output.

3. Use design tools to create a model of the program.

While planning a program, the programmer uses one or more design tools to create a model of the program. Three common design tools are hierarchy charts, flowcharts, and pseudocode. A *hierarchy chart* is a diagram that graphically depicts the structure of a program. It has boxes that represent each step in the program. The boxes are connected in a way that illustrates their relationship to one another. Figure 1-9 shows a hierarchy chart for the pay-calculating program.

A hierarchy chart begins with the overall task, and then refines it into smaller subtasks. Each of the subtasks is then refined into even smaller sets of subtasks, until each is small enough to be easily performed. For instance, in Figure 1-9, the overall task "Calculate Gross Pay" is listed in the top-level box. That task is broken into three subtasks. The first subtask, "Get Payroll Data from User," is broken further into two subtasks. This process of "divide and conquer" is known as *top-down design*.

Figure 1-9



A *flowchart* is a diagram that shows the logical flow of a program. It is a useful tool for planning each operation a program must perform, and the order in which the operations are to occur. For more information see Appendix O, Introduction to Flowcharting.

Pseudocode is a cross between human language and a programming language. Although the computer can't understand pseudocode, programmers often find it helpful to write an algorithm using it. This is because pseudocode is similar to natural language, yet close enough to programming language that it can be easily converted later into program source code. By writing the algorithm in pseudocode first, the programmer can focus on just the logical steps the program must perform, without having to worry yet about syntax or about details such as how output will be displayed.

Pseudocode can be written at a high level or at a detailed level. Many programmers use both forms. High level pseudocode simply lists the steps a program must perform. Here is high level pseudocode for the pay-calculating program.

```
Get payroll data
Calculate gross pay
Display gross pay
```

High level pseudocode can be expanded to produce detailed pseudocode. Here is the detailed pseudocode for the same program. Notice that it even names variables and tells what mathematical operations to perform.



VideoNote
Designing a Program with Pseudocode

```
Ask the user to input the number of hours worked
Input hours
Ask the user to input the hourly pay rate
Input rate
Set pay equal to hours times rate
Display pay
```

4. Check the model for logical errors.

Logical errors, also called *logic errors*, are mistakes that cause a program to produce erroneous results. Examples of logical errors would be using the wrong variable's value in a computation or performing order-dependent actions in the wrong order. Once a model of the program has been created, it should be checked for logical errors. The programmer should trace through the charts or pseudocode, checking the logic of each step. If an error is found, the model can be corrected before the actual program source code is written. In general, the earlier an error is detected in the programming process, the easier it is to correct.

5. Write the program source code.

Once a model of the program (hierarchy chart, flowchart, or pseudocode) has been created, checked, and corrected, the programmer is ready to write the source code, using an actual computer programming language, such as C++. Many programmers write the code directly on the computer, typing it into a text editor. Some programmers, however, prefer to write the program on paper first, then enter it into the computer. Once the program has been entered, the source code is saved to a file.

6. Compile the source code.

Next the saved source code is ready to be compiled. The compiler will translate the source code to machine language.

7. Correct any errors found during compilation.

If the compiler reports any errors, they must be corrected and the code recompiled. This step is repeated until the program is free of compile-time errors.

8. Link the program to create an executable file.

Once the source code compiles with no errors, it can be linked with the libraries specified by the program `#include` statements to create an executable file. If an error occurs during the linking process, it is likely that the program has failed to include a needed library file. The needed file must be included and the program relinked.

9. Run the program using test data for input.

Once an executable file is generated, the program is ready to be tested for run-time and logic errors. A run-time error occurs when the running program asks the computer to do something that is impossible, such as divide by zero. Normally a run-time error causes the program to abort. If the program runs, but fails to produce correct results, it likely contains one or more logic errors. To help identify such errors, it is important that the program be executed with carefully selected sample data that allows the correct output to be predicted.

10. Correct any errors found while running the program.

When run-time or logic errors occur in a program, they must be corrected. You must identify the step where the error occurred and determine the cause.

Desk-checking is a process that can help locate these types of errors. The term desk-checking means the programmer starts reading the program, or a portion of the program, and steps through each statement. A sheet of paper is often used in this process to jot down the current contents of all variables and sketch what the screen looks like after each output operation.

When a variable's contents change, or information is displayed on the screen, this is noted. By stepping through each statement in this manner, many errors can be located and corrected.

If the error is a result of incorrect logic (such as an improperly stated math formula), you must correct the statement or statements involved in the logic. If the error is due to an incomplete understanding of the program requirements, then you must restate the program's purpose and modify all affected charts, pseudocode, and source code. The program must then be saved, recompiled, relinked, and retested. This means steps 4 though 10 must be repeated until the program reliably produces satisfactory results.

11. Validate the results of the program.
When you believe you have corrected all errors, enter test data to verify that the program solves the original problem.

What Is Software Engineering?

The field of software engineering encompasses the complete process of crafting computer software. It includes designing, writing, testing, debugging, documenting, modifying, and maintaining complex software development projects. Like traditional engineers, software engineers use a number of tools in their craft. Here are a few examples:

- Program specifications
- Charts and diagrams of screen output
- Hierarchy charts
- Pseudocode
- Examples of expected input and desired output
- Special software designed for testing programs

Most commercial software applications are very large. In many instances one or more teams of programmers, not a single individual, develop them. It is important that the program requirements be thoroughly analyzed and divided into subtasks that are handled by individual teams or individuals within a team.

In step 3 of the programming process, you were introduced to the hierarchy chart as a tool for top-down design. When the subtasks identified in a top-down design are long or complex, they can be developed as modules, or separate components, of a program. If the program is very large or complex, a team of software engineers can be assigned to work on the individual modules. As the project develops, the modules are coordinated to become a single software application.



Checkpoint

1.24 What four items should you identify when defining what a program is to do?

1.25 What does it mean to "visualize a program running"? What is the value of doing this?

1.26 What is a hierarchy chart?

1.27 What is pseudocode?

1.28 What is the difference between high level pseudocode and detailed pseudocode?

1.29 Describe what a compiler does with a program's source code.

1.30 What is a logic error?

1.31 What is a run-time error?

1.32 Describe the process of desk-checking.

1.7 Tying It All Together: *Hi! It's Me*

Most programs, as you have learned, have three primary activities: input, processing, and output. But it is possible to write a program that has only output. Program 1-2, shown below, displays the message:

```
Hi! It's me.
I'm learning to program!
```

Program 1-2 can be found in the Chapter 1 folder on the student CD that came with your textbook. Open the program in whatever C++ development environment your class is using. Then compile it and run it. Your instructor will show you how to do this.

Program 1-2

```
1  //This program prints a message with your name in it.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      cout << "Hi! It\'s me.\n";
8      cout << "I\'m learning to program!\n";
9      return 0;
10 }
```

Once you have run it, change the word `me` on line 7 to *your* name to personalize the message. Then recompile and rerun the program.

In the next chapter you will learn what the `\'` and `\n` do.

Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. Computers can do many different jobs because they can be _____.
2. The job of the _____ is to fetch instructions, carry out the operations commanded by the instructions, and produce some outcome or resultant information.
3. Internally, the CPU consists of the _____ and the _____.
4. A(n) _____ is an example of a secondary storage device.

5. The two general categories of software are _____ and _____.
6. A program is a set of _____.
7. Since computers can't be programmed in natural human language, algorithms must be written in a(n) _____ language.
8. _____ is the only language computers really process.
9. _____ languages are close to the level of humans in terms of readability.
10. _____ languages are close to the level of the computer.
11. A program's ability to run on several different types of computer systems is called _____.
12. Words that have special meaning in a programming language are called _____ words.
13. Words or names defined by the programmer are called _____.
14. _____ are characters or symbols that perform operations on one or more operands.
15. _____ characters or symbols mark the beginning or ending of programming statements, or separate items in a list.
16. The rules that must be followed when constructing a program are called _____.
17. A(n) _____ is a named storage location.
18. A variable must be _____ before it can be used in a program.
19. The three primary activities of a program are _____, _____, and _____.
20. _____ is information a program gathers from the outside world.
21. _____ is information a program sends to the outside world.
22. A(n) _____ is a diagram that graphically illustrates the structure of a program.
23. Both main memory and secondary storage are types of memory. Describe the difference between the two.
24. What is the difference between operating system software and application software?
25. What is the difference between a syntax error and a logical error?

Algorithm Workbench

26. **Available Credit**

    Design a hierarchy chart for a program that calculates a customer's available credit. The program should carry out the following steps:

    1. Display the message "Enter the customer's maximum credit."
    2. Wait for the user to enter the customer's maximum credit.
    3. Display the message "Enter the amount of credit used by the customer."
    4. Wait for the user to enter the customer's credit used.
    5. Subtract the used credit from the maximum credit to get the customer's available credit.
    6. Display a message that shows the customer's available credit.

27. **Account Balance**



VideoNote
Designing the Account Balance Program

Write high-level and detailed psuedocode for a program that calculates the current balance in a bank account. The program must ask the user for

- The starting balance
- The total dollar amount of deposits made
- The total dollar amount of withdrawals made

Once the program calculates the current balance, it should be displayed on the screen.

28. **Sales Tax**

Write high-level and detailed psuedocode for a program that calculates the total of a retail sale. The program should ask the user for

- The retail price of the item being purchased
- The sales tax rate

Once these items have been entered, the program should calculate and display

- The sales tax for the purchase
- The total of the sale

Predict the Output

Questions 29–32 are programs expressed as English statements. What would each display on the screen if they were actual programs?

29. The variable `sum` starts with the value 0.
Add 10 to `sum`.
Add 15 to `sum`.
Add 20 to `sum`.
Display the value of `sum` on the screen.



VideoNote
Predicting the Output of Problem 30

30. The variable `x` starts with the value 0.
The variable `y` starts with the value 5.
Add 1 to `x`.
Add 1 to `y`.
Add `x` and `y`, and store the result in `y`.
Display the value in `y` on the screen.

31. The variable `j` starts with the value 10.
The variable `k` starts with the value 2.
The variable `l` starts with the value 4.
Store the value of `j` times `k` in `j`.
Store the value of `k` times `l` in `l`.
Add `j` and `l`, and store the result in `k`.
Display the value in `k` on the screen.

32. The variable `a` starts with the value 1.
    The variable `b` starts with the value 10.
    The variable `c` starts with the value 100.
    The variable `x` starts with the value 0.
    Store the value of `c` times 3 in `x`.
    Add the value of `b` times 6 to the value already in `x`.
    Add the value of `a` times 5 to the value already in `x`.
    Display the value in `x` on the screen.

Find the Error

33. The following pseudocode algorithm has an error. It is supposed to use values input for a rectangular room's length and width to calculate and display its area. Find the error.

```
area = width × length.
Display "What is the room's width?".
Input width.
Display "What is the room's length?".
Input length.
Display area.
```

Soft Skills

Before a programmer can design a program he or she must have some basic knowledge about the domain, or area, the program will deal with and must understand exactly what it is that the client wants the program to do. Otherwise the final program may not work correctly or may not meet the client's needs.

34. Suppose one of your friends, who paints the insides of houses, has asked you to develop a program that determines and displays how much paint is needed to paint a room if the length and width of the room is input. What information are you lacking that you need to write this program? Write at least three questions that you would need to ask your friend before starting the project.

Programming Challenges



VideoNote
Solving the Candy Bar Sales Problem

1. Candy Bar Sales

Using Program 1-1 as an example, write a program that calculates how much a student organization earns during its fund raising candy sale. The program should prompt the user to enter the number of candy bars sold and the amount the organization earns for each bar sold. It should then calculate and display the total amount earned.

2. Baseball Costs

Using Program 1-1 as an example, write a program that calculates how much a little league baseball team spent last year to purchase new baseballs. The program should prompt the user to enter the number of baseballs purchased and the cost of each baseball. It should then calculate and display the total amount spent to purchase the baseballs.

This page intentionally left blank

CHAPTER 2

Introduction to C++

TOPICS

2.1 The Parts of a C++ Program
2.2 The `cout` Object
2.3 The `#include` Directive
2.4 Standard and Prestandard C++
2.5 Variables, Constants, and the Assignment Statement
2.6 Identifiers
2.7 Integer Data Types
2.8 The `char` Data Type
2.9 The C++ `string` Class
2.10 Floating-Point Data Types
2.11 The `bool` Data Type
2.12 Determining the Size of a Data Type
2.13 More on Variable Assignments and Initialization
2.14 Scope
2.15 Arithmetic Operators
2.16 Comments
2.17 Focus on Software Engineering: *Programming Style*
2.18 Tying It All Together: *Smile!*

2.1 The Parts of a C++ Program

CONCEPT: C++ programs have parts and components that serve specific purposes.

Every C++ program has an anatomy. Unlike human anatomy, the parts of C++ programs are not always in the same place. Nevertheless, the parts are there and your first step in learning C++ is to learn what they are. We will begin by looking at Program 2-1.

Program 2-1

```
1 // A simple C++ program
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7    cout << "Programming is great fun!";
8    return 0;
9 }
```

The output of the program is shown below. This is what appears on the screen when the program runs.

Program Output

```
Programming is great fun!
```

Let's examine the program line by line. Here's the first line:

```
// A simple C++ program
```

The `//` marks the beginning of a *comment*. The compiler ignores everything from the double-slash to the end of the line. That means you can type anything you want on that line and the compiler will never complain! Although comments are not required, they are very important to programmers. Most programs are much more complicated than the example in Program 2-1, and comments help explain what's going on.

Line 2 looks like this:

```
#include <iostream>
```

This line must be included in a C++ program in order to get input from the keyboard or print output to the screen. Since the `cout` statement (on line 7) will print output to the computer screen, we need to include this line. When a line begins with a `#` it indicates it is a *preprocessor directive*. The preprocessor reads your program before it is compiled and only executes those lines beginning with a `#` symbol. Think of the preprocessor as a program that "sets up" your source code for the compiler.

The `#include` directive causes the preprocessor to include the contents of another file in the program. The word inside the brackets, `iostream`, is the name of the file that is to be included. The `iostream` file contains code that allows a C++ program to display output on the screen and read input from the keyboard. Because this program uses `cout` to display screen output, the `iostream` file must be included. Its contents are included in the program at the point the `#include` statement appears. The `iostream` file is called a *header file*, so it should be included at the head, or top, of the program.

Line 3 reads

```
using namespace std;
```

Programs usually contain various types of items with unique names. In this chapter you will learn to create variables. In Chapter 6 you will learn to create functions. In Chapter 7 you will learn to create objects. Variables, functions, and objects are examples of

program entities that must have names. C++ uses *namespaces* to organize the names of program entities. The statement `using namespace std;` declares that the program will be accessing entities whose names are part of the namespace called `std`. (Yes, even namespaces have names.) The program needs access to the `std` namespace because every name created by the `iostream` file is part of that namespace. In order for a program to use the entities in `iostream`, it must have access to the `std` namespace. More information on namespaces can be found in Appendix E.

Line 5 reads

```
int main()
```

This marks the beginning of a function. A *function* can be thought of as a group of one or more programming statements that has a name. The name of this function is `main`, and the set of parentheses that follows the name indicates that it is a function. The word `int` stands for "integer." It indicates that the function sends an integer value back to the operating system when it is finished executing.

Although most C++ programs have more than one function, every C++ program must have a function called `main`. It is the starting point of the program. If you're ever reading someone else's program and want to find where it starts, just look for the function called `main`.



NOTE: C++ is a case-sensitive language. That means it regards uppercase letters as being entirely different characters than their lowercase counterparts. In C++, the name of the function `main` must be written in all lowercase letters. C++ doesn't see "main" the same as "Main" or "MAIN."

Line 6 contains a single, solitary character:

```
{
```

This is called a left-brace, or an opening brace, and it is associated with the beginning of the function `main`. All the statements that make up a function are enclosed in a set of braces. If you look at the third line down from the opening brace you'll see the closing brace. Everything between the two braces is the contents of the function `main`.



WARNING! Make sure you have a closing brace for every opening brace in your program.

After the opening brace you see the following statement in line 7:

```
cout << "Programming is great fun!";
```

To put it simply, this line displays a message on the screen. You will read more about `cout` and the `<<` operator later in this chapter. The message "Programming is great fun!" is printed without the quotation marks. In programming terms, the group of characters inside the quotation marks is called a *string literal,* a *string constant,* or simply a *string*.



NOTE: This is the only line in the program that causes anything to be printed on the screen. The other lines, like `#include <iostream>` and `int main()`, are necessary for the framework of your program, but they do not cause any screen output. Remember, a program is a set of instructions for the computer. If something is to be displayed on the screen, you must use a programming statement for that purpose.

Notice that line 7 ends with a semicolon. Just as a period marks the end of a sentence, a semicolon is required to mark the end of a complete statement in C++. But many C++ lines do not end with semicolons. Some of these include comments, preprocessor directives, and the beginning of functions. Here are some examples of when to use, and not use, semicolons.

```
// Semicolon examples        // This is a comment
# include <iostream>        // This is a preprocessor directive
int main()                  // This begins a function
cout << "Hello";            // This is a complete statement
```

As you spend more time working with C++ you will get a feel for where you should and should not use semicolons. For now don't worry about it. Just concentrate on learning the parts of a program.

Line 8 reads

```
return 0;
```

This sends the integer value 0 back to the operating system upon the program's completion. The value 0 usually indicates that a program executed successfully.

The last line of the program, line 9, contains the closing brace:

```
}
```

This brace marks the end of the `main` function. Because `main` is the only function in this program, it also marks the end of the program.

In the sample program you encountered several sets of special characters. Table 2-1 provides a short summary of how they were used.

Table 2-1 Special Characters

Character	Name	Description
`//`	Double slash	Marks the beginning of a comment.
`#`	Pound sign	Marks the beginning of a preprocessor directive.
`< >`	Opening and closing brackets	Encloses a filename when used with the `#include` directive.
`( )`	Opening and closing parentheses	Used in naming a function, as in `int main()`.
`{ }`	Opening and closing braces	Encloses a group of statements, such as the contents of a function.
`" "`	Opening and closing quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen.
`;`	Semicolon	Marks the end of a complete programming statement.

Checkpoint

2.1 The following C++ program will not compile because the lines have been mixed up.

```
int main()
}
// A crazy mixed up program
#include <iostream>
return 0;
cout << "In 1492 Columbus sailed the ocean blue.";
{
using namespace std;
```

When the lines are properly arranged the program should display the following on the screen:

```
In 1492 Columbus sailed the ocean blue.
```

Rearrange the lines in the correct order. Test the program by entering it on the computer, compiling it, and running it.

2.2 On paper, write a program that will display your name on the screen. Use Program 2-1 as your guide. Place a comment with today's date at the top of the program. Test your program by entering, compiling, and running it.

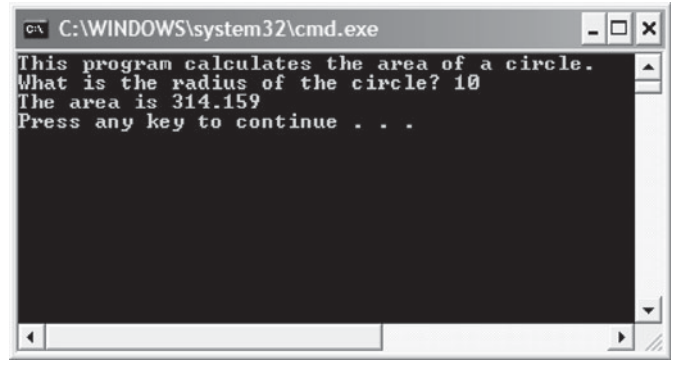
2.2 The cout Object

CONCEPT: cout is used to display information on the computer's screen.

In this section you will learn to write programs that produce output on the screen. The simplest type of screen output that a program can display is *console output*, which is merely plain text. The word *console* is an old computer term. It comes from the days when a computer operator interacted with the system by typing on a terminal. The terminal, which consisted of a simple screen and keyboard, was known as the console.

On modern computers, running graphical operating systems such as Windows or Mac OS X, console output is usually displayed in a window such as the one shown in Figure 2-1. C++ provides an object named `cout` that is used to produce console output. (You can think of the word `cout` as meaning **c**onsole **out**put.)

Figure 2-1 A Console Window





VideoNote
Using cout to Display Output

cout is classified as a *stream object*, which means it works with streams of data. To print a message on the screen, you send a stream of characters to cout. Let's look at line 7 from Progam 2-1:

```
cout << "Programming is great fun!";
```

The << operator is used to send the string "Programming is great fun!" to cout. When the << symbol is used this way, it is called the *stream-insertion operator.* The item immediately to the right of the operator is sent to cout and then displayed on the screen.



NOTE: The stream insertion operator is always written as two less-than signs with no space between them. Because you are using it to send a stream of data to the cout object, you can think of the stream insertion operator as an arrow that must point toward cout, as shown here.

```
cout << "Hello";
cout ← "Hello";
```

Program 2-2 shows another way to write the same program.

Program 2-2

```
1 // A simple C++ program
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7    cout << "Programming is " << "great fun!";
8    return 0;
9 }
```

Program Output

```
Programming is great fun!
```

As you can see, the stream-insertion operator can be used to send more than one item to cout. The output of this program is identical to Program 2-1. Program 2-3 shows yet another way to accomplish the same thing.

Program 2-3

```
1 // A simple C++ program
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7    cout << "Programming is ";
8    cout << "great fun!";
9    return 0;
10 }
```

Program Output

```
Programming is great fun!
```

An important concept to understand about Program 2-3 is that although the output is broken into two programming statements, this program will still display the message on a single line. Unless you specify otherwise, the information you send to `cout` is displayed in a continuous stream. Sometimes this can produce less-than-desirable results. Program 2-4 illustrates this.

Program 2-4

```
1  // An unruly printing program
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     cout << "The following items were top sellers";
8     cout << "during the month of June:";
9     cout << "Computer games";
10    cout << "Coffee";
11    cout << "Aspirin";
12    return 0;
13 }
```

Program Output

```
The following items were top sellersduring the month of June:Computer
gamesCoffeeAspirin
```

The layout of the actual output looks nothing like the arrangement of the strings in the source code. First, notice there is no space displayed between the words "sellers" and "during," or between "June:" and "Computer." `cout` displays messages exactly as they are sent. If spaces are to be displayed, they must appear in the strings.

Second, even though the output is broken into five lines in the source code, it comes out as one long line of output. Because the output is too long to fit on one line of the screen, it wraps around to a second line when displayed. The reason the output comes out as one long line is that `cout` does not start a new line unless told to do so. There are two ways to instruct `cout` to start a new line. The first is to send `cout` a *stream manipulator* called `endl` (pronounced "end-line" or "end-L"). Program 2-5 does this.

Program 2-5

```
1  // A well-adjusted printing program
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     cout << "The following items were top sellers" << endl;
8     cout << "during the month of June:" << endl;
9     cout << "Computer games" << endl;
10    cout << "Coffee" << endl;
11    cout << "Aspirin" << endl;
12    return 0;
13 }
```

(program continues)

Program 2-5 *(continued)*

Program Output

```
The following items were top sellers
during the month of June:
Computer games
Coffee
Aspirin
```



NOTE: The last character in `endl` is the lowercase letter L, *not* the number one.

Every time `cout` encounters an `endl` stream manipulator it advances the output to the beginning of the next line for subsequent printing. The manipulator can be inserted anywhere in the stream of characters sent to `cout`, outside the double quotes. Notice that an `endl` is also used at the end of the last line of output.

Another way to cause subsequent output to begin on a new line is to insert a `\n` in the string that is being output. Program 2-6 does this.

Program 2-6

```
1  // Another well-adjusted printing program
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     cout << "The following items were top sellers\n";
8     cout << "during the month of June:\n";
9     cout << "Computer games\nCoffee";
10    cout << "\nAspirin\n";
11    return 0;
12 }
```

Program Output

```
The following items were top sellers
during the month of June:
Computer games
Coffee
Aspirin
```

`\n` is an example of an *escape sequence*. Escape sequences are written as a backslash character (`\`) followed by one or more control characters and are used to control the way output is displayed. There are many escape sequences in C++. The newline escape sequence (`\n`) is just one of them.

When `cout` encounters `\n` in a string, it doesn't print it on the screen but interprets it as a special command to advance the output cursor to the next line. You have probably noticed inserting the escape sequence requires less typing than inserting `endl`. That's why some programmers prefer it.

Escape sequences give you the ability to exercise greater control over the way information is output by your program. Table 2-2 lists a few of them.

Table 2-2 Common Escape Sequences

Escape Sequence	Name	Description
\n	Newline	Causes the cursor to go to the next line for subsequent printing.
\t	Horizontal tab	Causes the cursor to skip over to the next tab stop.
\a	Alarm	Causes the computer to beep.
\b	Backspace	Causes the cursor to back up, or move left one position.
\r	Return	Causes the cursor to go to the beginning of the current line, not the next line.
\\	Backslash	Causes a backslash to be printed.
\'	Single quote	Causes a single quotation mark to be printed.
\"	Double quote	Causes a double quotation mark to be printed.

A common mistake made by beginning C++ students is to use a forward slash (/) instead of a back slash (\) when trying to write an escape sequence. This will not work. For example, look at the following line of code.

```
cout << "Four score/nAnd seven/nYears ago./n";  // Error!
```

Because the programmer accidentally wrote /n instead of \n, cout will simply display the /n characters on the screen, rather than starting a new line of output. This code will create the following output:

```
Four score/nAnd seven/nYears ago./n
```

Another common mistake is to forget to put the \n inside quotation marks. For example, the following code will not compile.

```
cout << "Good" << \n;          // Error!
cout << "Morning" << \n;       // This code will not compile.
```

We can correct the code by placing the \n sequences inside the string literals, as shown here:

```
cout << "Good\n";              // This will work.
cout << "Morning\n";
```

To summarize, do not confuse the backslash (\) with the forward slash (/). An escape sequence must start with a backslash, be placed inside quotation marks, and have no spaces between the backslash and the control character.

2.3 The #include Directive

CONCEPT: **The #include directive causes the contents of another file to be inserted into the program.**

Now is a good time to expand our discussion of the #include directive. The following line has appeared near the top of every example program.

```
#include <iostream>
```

The header file `iostream` must be included in any program that uses the `cout` object. This is because `cout` is not part of the "core" of the C++ language. Specifically, it is part of the *input-output stream library*. The header file, `iostream`, contains information describing `iostream` objects. Without it, the compiler will not know how to properly compile a program that uses `cout`.

Preprocessor directives are not C++ statements. They are commands to the preprocessor, which runs prior to the compiler (hence the name "preprocessor"). The preprocessor's job is to set programs up in a way that makes life easier for the programmer.

For example, any program that uses the `cout` object must contain the extensive setup information found in the `iostream` file. The programmer could type all this information into the program, but it would be too time consuming. An alternative would be to use an editor to "cut and paste" the information into the program, but that would still be inefficient. The solution is to let the preprocessor insert the contents of `iostream` automatically.



WARNING! Do not use semicolons at the end of preprocessor directives. Because preprocessor directives are not C++ statements, they do not require them. In fact, in many cases an error will result if a preprocessor directive is terminated with a semicolon.

An `#include` directive must always contain the name of a file. The preprocessor inserts the entire contents of the file into the program at the point it encounters the `#include` directive. The compiler doesn't actually see the `#include` directive. Instead it sees the code that was inserted by the preprocessor, just as if the programmer had typed it there.

The code contained in header files is C++ code. Typically it describes complex objects like `cout`. Later you will learn to create your own header files.



Checkpoint

2.3 The following `cout` statement contains errors.

```
cout << "red /n" << "blue \ n" << "yellow" \n << "green";
```

Correct it so that it will display a list of colors, with one item per line.

2.4 What output will the following lines of code display on the screen?

```
cout << "The works of Wolfgang\ninclude the following";
cout << "\nThe Turkish March" << endl;
cout << "and Symphony No. 40 ";
cout << "in G minor." << endl;
```

2.5 On paper, write a program that will display your name on the first line, your street address on the second line, your city, state, and ZIP code on the third line, and your telephone number on the fourth line. Test your program by entering, compiling, and running it.

2.4 Standard and Prestandard C++

CONCEPT: **C++ programs written before the language became standardized may appear slightly different from programs written today.**

C++ is now a standardized programming language, but it hasn't always been. The language has evolved over the years and, as a result, there is a "newer style" and an "older style" of writing C++ code. The newer style is the way programs are written with standard C++, while the older style is the way programs were typically written using prestandard C++. Although the differences between the older and newer styles are subtle, it is important that you recognize them. When you go to work as a computer science professional, it is likely that you will see programs written in the older style.

Older Style Header Files

In older style C++, all header files end with the ".h" extension. For example, in a prestandard C++ program the statement that includes the `iostream` header file is written as

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

Absence of `using namespace std;`

Another difference between the newer and older styles is that older style programs typically do not use the `using namespace std;` statement. In fact, some older compilers do not support namespaces at all and will produce an error message if a program has that statement.

An Older Style Program

To illustrate these differences, look at the following program. It is a modification of Program 2-1, written in the older style.

```
// A simple C++ program
#include <iostream.h>

void main(void)
{
   cout << "Programming is great fun!";
}
```

Some standard C++ compilers do not support programs written in the older style, and prestandard compilers normally do not support programs written in the newer style.

2.5 Variables, Constants, and the Assignment Statement

CONCEPT: **Variables represent storage locations in the computer's memory. Constants are data items whose values cannot change while the program is running.**

The concept of a variable in computer programming is somewhat different from the concept of a variable in mathematics. In programming, as you learned in Chapter 1, a variable is a

named storage location for holding data. Variables allow you to store and work with data in the computer's memory. They provide an "interface" to RAM. Part of the job of programming is to determine how many variables a program will need and what type of information each will hold. Program 2-7 is an example of a C++ program with a variable.

Program 2-7

```
1  // This program has a variable.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int number;
8
9     number = 5;
10    cout << "The value of number is " << "number" << endl;
11    cout << "The value of number is " <<  number  << endl;
12
13    number = 7;
14    cout << "Now the value of number is " << number << endl;
15
16    return 0;
17 }
```

Program Output

```
The value of number is number
The value of number is 5
Now the value of number is 7
```

Let's look more closely at this program. Start by looking at line 7.

```
int number;
```

This is called a *variable definition*. It tells the compiler the variable's name and the type of data it will hold. Notice that the data type is written first, followed by the name of the variable. This variable's name is `number`. The word `int` stands for integer, so `number` may only be used to hold integer numbers. Later in this chapter you will learn all the types of data that C++ allows.



NOTE: You must have a definition for every variable you use in a program. In C++, a variable definition can appear at any point in the program as long as it occurs before the variable is ever used. Later in this chapter, and throughout the book, you will learn the best places to define variables. Notice that variable definitions end with a semicolon.

Now look at line 9.

```
number = 5;
```

This is called an *assignment statement* and the = sign is called the *assignment operator*. This operator copies the value on its right (5) into the variable named on its left (`number`). This line does not print anything on the computer's screen. It runs silently behind the scenes, storing a value in RAM. After this line executes, `number` will be set to 5.



NOTE: The item on the left hand side of an assignment statement *must* be a variable. It would be incorrect to say `5 = number;`

Now look at lines 10 and 11. Notice that in line 10 the word `number` has double quotation marks around it and in line 11, it does not.

```
cout << "The value of number is " << "number" << endl;
cout << "The value of number is " <<  number  << endl;
```

Now compare these two lines with the output they produce. In the first `cout` statement, the string constant `"number"` is inserted into the output stream, so the output produced is

```
The value of number is number
```

In the second `cout` statement, because there are no quotation marks around it, it is the variable name `number` that is inserted into the output stream, causing its value to print.

```
The value of number is 5
```

Recall from Chapter 1 that variables are called variables because their values can change. The assignment statement on line 13 replaces the previous value stored in `number` with a 7.

```
number = 7;
```

Therefore the final `cout` statement on line 14

```
cout << "Now the value of number is " << number  << endl;
```

causes the following output to print.

```
Now the value of number is 7
```

Sometimes a Number Isn't a Number

As shown in Program 2-7, placing quotation marks around a variable name made it a string constant, or string literal. When string literals are sent to `cout`, they are printed exactly as they appear inside the quotation marks. You've probably noticed by now that the `endl` stream manipulator is written with no quotation marks around it. If we were to put the following line in a program, it would print out the word `endl`, rather than cause subsequent output to begin on a new line.

```
cout << "endl";     // Wrong!
```

In fact, placing double quotation marks around anything that is not intended to be a string literal will create an error of some type. For example, in Program 2-7 the number 5 was assigned to the variable `number`. It would have been incorrect to write the assignment this way:

```
number = "5";       // Wrong!
```

In this line, 5 is no longer an integer, but a string literal. Because `number` was defined to be an integer variable, you can only store integers in it. The integer 5 and the string literal "5" are not the same thing.

The fact that numbers can be represented as strings frequently confuses people who are new to programming. Just remember that strings are intended for humans to read. They are to be printed on computer screens or paper. Numbers, however, are intended primarily for mathematical operations. You cannot perform math on strings, and you cannot display numbers on the screen without first converting them to strings. (Fortunately, `cout` handles the conversion automatically when you send a number to it.)

Constants

Unlike a variable, a constant is a data item whose value cannot change during the program's execution. Program 2-8 contains integer constants, string constants (i.e., string literals), and a variable.

Program 2-8

```
1  // This program has constants and a variable.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int apples;
8
9     apples = 20;
10    cout << "On Sunday we sold " << apples << " bushels of apples. \n";
11
12    apples = 15;
13    cout << "On Monday we sold " << apples << " bushels of apples. \n";
14    return 0;
15 }
```

Program Output

```
On Sunday we sold 20 bushels of apples.
On Monday we sold 15 bushels of apples.
```

Of course, the variable is `apples`. Table 2-3 lists the constants found in the program.

Table 2-3 Program 2-8 Constants

Integer Constants	String Literals
20	"On Sunday we sold"
15	"On Monday we sold"
0	"bushels of apples. \n"

What are constants used for? As you can see from Program 2-8, they are commonly used to store known values in variables and to display messages on the screen.



Checkpoint

2.6 Which of the following are legal C++ assignment statements?

```
a.  a = 7;
b.  7 = a;
c.  7 = 7;
```

2.7 List all the variables and constants that appear below.

```
int main()
{
   int little;
   int big;
   little = 2;
   big = 2000;
   cout << "The little number is " << little << endl;
   cout << "The big number is " << big << endl;
   return 0;
}
```

2.8 When the above program is run, what will display on the screen?

2.9 What will the following program display on the screen?

```
#include <iostream>
using namespace std;

int main()
{
   int number;

   number = 712;
   cout << "The value is " << "number" << endl;
   return 0;
}
```

2.6 Identifiers

CONCEPT: A variable name should indicate what the variable is used for.

An *identifier* is a programmer-defined name that represents some element of a program. Variable names are examples of identifiers. You may choose your own variable names in C++, as long as you do not use any of the C++ *key words*. The key words make up the "core" of the language and have specific purposes. Table 2-4 shows a complete list of the C++ key words. Note that they are all lowercase.

Table 2-4 C++ Key Words

and	continue	goto	public	try
and_eq	default	if	register	typedef
asm	delete	inline	reinterpret_cast	typeid
auto	do	int	return	typename
bitand	double	long	short	union
bitor	dynamic_cast	mutable	signed	unsigned
bool	else	namespace	sizeof	using
break	enum	new	static	virtual
case	explicit	not	static_cast	void
catch	export	not_eq	struct	volatile
char	extern	operator	switch	wchar_t
class	false	or	template	while
compl	float	or_eq	this	xor
const	for	private	throw	xor_eq
const_cast	friend	protected	true	

You should always choose names for your variables that give an indication of what the variables are used for. You may be tempted to give variables names like this:

```
int x;
```

However, the rather nondescript name, `x`, gives no clue as to the variable's purpose. Here is a better example.

```
int itemsOrdered;
```

The name `itemsOrdered` gives anyone reading the program an idea of the variable's use. This way of coding helps produce self-documenting programs, which means you can get an understanding of what the program is doing just by reading its code. Because real-world programs usually have thousands of lines, it is important that they be as self-documenting as possible.

You probably have noticed the mixture of uppercase and lowercase letters in the variable name `itemsOrdered`. Although all of C++'s key words must be written in lowercase, you may use uppercase letters in variable names.

The reason the `O` in `itemsOrdered` is capitalized is to improve readability. Normally "items ordered" is two words. However, you cannot have spaces in a variable name, so the two words must be combined into one. When "items" and "ordered" are stuck together you get a variable definition like this:

```
int itemsordered;
```

Capitalization of the first letter of the second word and any succeeding words makes variable names like `itemsOrdered` easier to read and is the convention we use for naming variables in this book. However, this style of coding is not required. You are free to use all lowercase letters, all uppercase letters, or any combination of both. In fact, some programmers use the underscore character to separate words in a variable name, as in the following.

```
int items_ordered;
```

Legal Identifiers

Regardless of which style you adopt, be consistent and make your variable names as sensible as possible. Here are some specific rules that must be followed with all C++ identifiers.

- The first character must be one of the letters a through z, A through Z, or an underscore character (_).

- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct. This means `ItemsOrdered` is not the same as `itemsordered`.

Table 2-5 lists variable names and indicates whether each is legal or illegal in C++.

Table 2-5 Some C++ Variable Names

Variable Name	Legal or Illegal
`dayOfWeek`	Legal.
`3dGraph`	Illegal. Variable names cannot begin with a digit.
`_employee_num`	Legal.
`June1997`	Legal.
`Mixture#3`	Illegal. Variable names may only use letters, digits, and underscores.

2.7 Integer Data Types

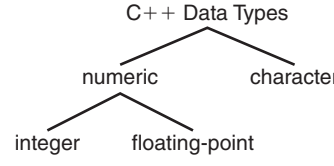
CONCEPT: **There are many different types of data. Variables are classified according to their data type, which determines the kind of information that may be stored in them. Integer variables can only hold whole numbers.**

Computer programs collect pieces of data from the real world and manipulate them in various ways. There are many different types of data. In the realm of numeric information, for example, there are whole numbers and fractional numbers. There are negative numbers and positive numbers. And there are numbers so large, and others so small, that they don't even have a name. Then there is textual information. Names and addresses, for instance, are stored as groups of characters. When you write a program you must determine what types of information it will be likely to encounter.

If you are writing a program to calculate the number of miles to a distant star, you'll need variables that can hold very large numbers. If you are designing software to record microscopic dimensions, you'll need to store very small and precise numbers. Additionally, if you are writing a program that must perform thousands of intensive calculations, you'll want data stored in variables that can be processed quickly. The data type of a variable determines all of these factors.

Although C++ offers many data types, in the very broadest sense there are only two: numeric and character. Numeric data types are broken into two additional categories: integer and floating-point, as shown in Figure 2-2.

Figure 2-2 Basic C++ Data Types



Integers are whole numbers like –2, 19, and 24. Floating-point numbers have a decimal point like –2.35, 19.0, and 0.024. Additionally, the integer and floating-point data types are broken into even more classifications.

Your primary considerations for selecting the best data type for a numeric variable are the following:

- whether the variable needs to hold integers or floating-point values,
- the largest and smallest numbers that the variable needs to be able to store,
- whether the variable needs to hold signed (both positive and negative) or only unsigned (just zero and positive) numbers, and
- the number of decimal places of precision needed for values stored in the variable.

Let's begin by looking at integer data types. C++ has six different data types for storing integers. On most computers each of these has either two or four bytes of memory. The number of bytes a data type can hold is called its *size*. Typically, the larger the size a data type is, the greater the range of values it can hold.

Recall from Chapter 1 that a byte is made up of 8 bits. So a data type that stores data in two bytes of memory can hold 16 bits of information. This means it can store 2^{16} *bit patterns*, which is 65,536 different combinations of zeros and ones. A data type that uses 4 bytes of memory has 32 bits, so it can hold 2^{32} different bit patterns, which is 4,294,967,296 different combinations. What these different combinations are used for depends on the data type. For example, the `unsigned short` data type, which is for storing non-negative integers such as ages or weights, uses its 16 bits to represent the values 0 through +65,535. The `short` data type, on the other hand, stores both positive and negative numbers, so it uses its 16 bits to represent the values from –32,768 to +32,767. Figure 2-3 shows how numbers are stored in an `unsigned short` variable.

Figure 2-3 Unsigned Short Data Type Storage

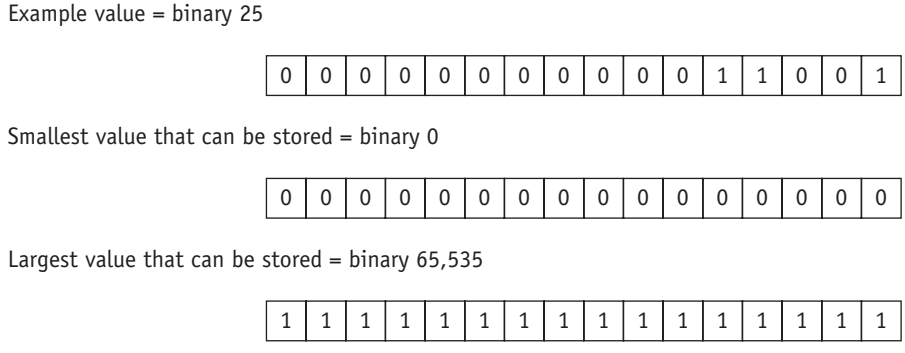


Table 2-6 shows all six C++ integer data types with their typical sizes and ranges. Depending on your operating system, the sizes and ranges may be different.

Table 2-6 Integer Data Types, Sizes, and Ranges

Data Type	Size	Range
`short`	2 bytes	–32,768 to +32,767
`unsigned short`	2 bytes	0 to +65,535
`int`	4 bytes	–2,147,483,648 to +2,147,483,647
`unsigned int`	4 bytes	0 to 4,294,967,295
`long`	4 bytes	–2,147,483,648 to +2,147,483,647
`unsigned long`	4 bytes	0 to 4,294,967,295

Here are some examples of integer variable definitions. Notice that an `unsigned int` variable can also be defined using only the word `unsigned`, as shown below.

```
short  count;
unsigned short age;
int speed;
unsigned int days;     // These two definitions
unsigned days;         // are equivalent.
long deficit;
unsigned long insects;
```

Notice also that in Table 2-6 the `int` and `long` data types have the same sizes and ranges, and the `unsigned int` data type has the same size and range as the `unsigned long` data type. This is not always true because the size of integers is dependent on the type of system you are using. Here are the only guarantees:

- Integers are at least as big as short integers.
- Long integers are at least as big as integers.
- Unsigned short integers are the same size as short integers.
- Unsigned integers are the same size as integers.
- Unsigned long integers are the same size as long integers.

Later in this chapter you will learn to use the `sizeof` operator to determine how large all the data types are on your computer.

Program 2-9 uses integer, unsigned integer, and long integer variables.

Program 2-9

```
1  // This program has variables of several of the integer types.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int checking;
8     unsigned int miles;
9     long days;
10
```

(program continues)

Program 2-9 *(continued)*

```
11    checking = -20;
12    miles = 4276;
13    days = 190000;
14
15    cout << "We have made a long journey of " << miles << " miles.";
16    cout << "\nOur checking account balance is " << checking;
17    cout << "\nAbout " << days << " days ago Columbus ";
18    cout << "stood on this spot.\n";
19    return 0;
20 }
```

Program Output

```
We have made a long journey of 4276 miles.
Our checking account balance is -20
About 190000 days ago Columbus stood on this spot.
```

In most programs you will need more than one variable of any given data type. If a program uses two integers, `length` and `width`, they can be defined separately, like this:

```
int length;
int width;
```

It is also possible to combine both variable definitions in a single statement:

```
int length, width;
```

Many instructors, however, prefer that each variable be placed on its own line:

```
int length,
    width;
```

Whether you place multiple variables on the same line or each variable on its own line, when you define several variables of the same type in a single statement, simply separate their names with commas. A semicolon is used at the end of the entire definition. Program 2-10 illustrates this. This program also shows how it is possible to give an initial value to a variable at the time it is defined.

Program 2-10

```
1  // This program defines three variables in the same statement.
2  // They are given initial values at the time they are defined.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int floors =  15,
9         rooms  = 300,
10        suites =  30;
11
12    cout << "The Grande Hotel has " << floors << " floors\n";
13    cout << "with " << rooms << " rooms and " << suites;
14    cout << " suites.\n";
15    return 0;
16 }
```

(program continues)

Program 2-10 *(continued)*

Program Output

```
The Grande Hotel has 15 floors
with 300 rooms and 30 suites.
```

Integer and Long Integer Constants

Look at the following statement from Program 2-10:

```
int floors = 15,
    rooms = 300,
    suites = 30;
```

This statement contains three integer constants. In C++, integer constants are normally stored in memory just as an `int`.

One of the pleasing characteristics of the C++ language is that it allows you to control almost every aspect of your program. If you need to change the way something is stored in memory, the tools are provided to do that. For example, what if you are in a situation where you have an integer constant, but you need it to be stored in memory as a long integer? (Rest assured, this is a situation that does arise.) C++ allows you to force an integer constant to be stored as a long integer by placing the letter L at the end of the number. Here is an example:

```
32L
```

On a computer that uses 2-byte integers and 4-byte long integers, this constant will use 4 bytes. This is called a long integer constant.



NOTE: Although C++ allows you to use either an uppercase or lowercase L, the lowercase l looks too much like the number 1, so you should always use the uppercase L.

Hexadecimal and Octal Constants (enrichment)

Programmers commonly express values in numbering systems other than decimal (or base 10). Hexadecimal (base 16) and octal (base 8) are popular because they make certain programming tasks more convenient than decimal numbers do.

By default, C++ assumes that all integer constants are expressed in decimal. You express hexadecimal numbers by placing `0x` in front of them. (This is zero-x, not oh-x.) Here is how the hexadecimal number F4 would be expressed in C++:

```
0xF4
```

Octal numbers must be preceded by a 0 (zero, not oh). For example, the octal 31 would be written

```
031
```



NOTE: You will not be writing programs for some time that require this type of manipulation. However, good programmers develop the skills for reading other people's source code. You may find yourself reading programs that use items like long integer, hexadecimal, or octal constants.



Checkpoint

2.10 Which of the following are illegal C++ variable names, and why?

```
x
99bottles
july97
theSalesFigureForFiscalYear98
r&d
grade_report
```

2.11 Is the variable name `Sales` the same as `sales`? Why or why not?

2.12 Refer to the data types listed in Table 2-6 for these questions.

A) If a variable needs to hold numbers in the range 32 to 6,000, what data type would be best?

B) If a variable needs to hold numbers in the range –40,000 to +40,000, what data type would be best?

C) Which of the following constants use more memory, `20` or `20L`?

2.13 Which integer data types can only hold non-negative values?

2.14 How would you consolidate the following variable definition and assignment statement into a single statement?

```
int apples;
apples = 20;
```

2.8 The `char` Data Type

CONCEPT: A variable of the `char` data type holds only a single character.

You might be wondering why there isn't a 1-byte integer data type. Actually there is. It is called the `char` data type, which gets its name from the word "character." A variable defined as a `char` can hold a single character, but strictly speaking, it is an integer data type.



NOTE: On some systems the `char` data type is larger than 1 byte.

The reason an integer data type is used to store characters is because characters are internally represented by numbers. Each printable character, as well as many nonprintable characters, are assigned a unique number. The most commonly used method for encoding characters is ASCII, which stands for the American Standard Code for Information Interchange. (There are other codes, such as EBCDIC, which is used by many IBM mainframes.)

When a character is stored in memory, it is actually the numeric code that is stored. When the computer is instructed to print the value on the screen, it displays the character that corresponds with the numeric code.

You may want to refer to Appendix A, which shows the ASCII character set. Notice that the number 65 is the code for A, 66 is the code for B, and so on. Program 2-11 demonstrates that when you work with characters, you are actually working with numbers.

Program 2-11

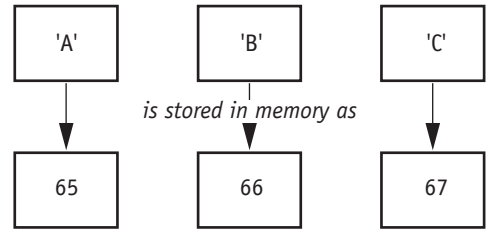
```
1  // This program demonstrates the close relationship between
2  // characters and integers.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     char letter;
9
10    letter = 65;
11    cout << letter << endl;
12
13    letter = 66;
14    cout << letter << endl;
15    return 0;
16 }
```

Program Output

```
A
B
```

Figure 2-4 illustrates that when you think of characters, such as A, B, and C, being stored in memory, it is really the numbers 65, 66, and 67 that are stored.

Figure 2-4



Character and String Constants

Although Program 2-11 nicely illustrates the way characters are represented by numbers, it isn't necessary to work with the ASCII codes themselves. Program 2-12 is another version that works the same way.

Program 2-12

```
1  // This program uses character constants.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     char letter;
```

(program continues)

Program 2-12 *(continued)*

```
 8
 9     letter = 'A';
10     cout << letter << endl;
11
12     letter = 'B';
13     cout << letter << endl;
14     return 0;
15 }
```

Program Output

```
A
B
```

Program 2-12 assigns character constants to the variable `letter`. Anytime a program works with a character, it internally works with the code used to represent that character, so this program is still assigning the values 65 and 66 to `letter`.

Character constants can only hold a single character. To store a series of characters in a constant we need a string constant. In the following example, `'H'` is a character constant and `"Hello"` is a string constant. Notice that a character constant is enclosed in single quotation marks whereas a string constant is enclosed in double quotation marks.

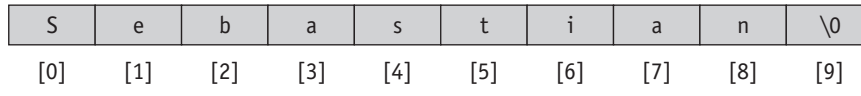
```
cout << 'H' << endl;
cout << "Hello" << endl;
```

Strings, which allow a series of characters to be stored in consecutive memory locations, can be virtually any length. This means that there must be some way for the program to know how long the string is. In C++ this is done by appending an extra byte to the end of string constants. In this last byte, the number 0 is stored. It is called the *null terminator* or *null character* and marks the end of the string.

Don't confuse the null terminator with the character `'0'`. If you look at Appendix A you will see that the character `'0'` has ASCII code 48, whereas the null terminator has ASCII code 0. If you want to print the character `0` on the screen, you use ASCII code 48. If you want to mark the end of a string, you use ASCII code 0.

Let's look at an example of how a string constant is stored in memory. Figure 2-5 depicts the way the string `"Sebastian"` would be stored.

Figure 2-5



First, notice the quotation marks are not stored with the string. They are simply a way of marking the beginning and end of the string in your source code. Second, notice the very last byte of the string. It contains the null terminator, which is represented by the \0 character. The addition of this last byte means that although the string `"Sebastian"` is 9 characters long, it occupies 10 bytes of memory.

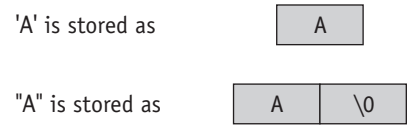
The null terminator is another example of something that sits quietly in the background. It doesn't print on the screen when you display a string, but nevertheless, it is there silently doing its job.



NOTE: C++ automatically places the null terminator at the end of string constants.

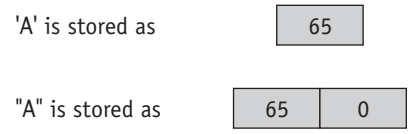
Now let's compare the way a string and a `char` are stored. Suppose you have the constants `'A'` and `"A"` in a program. Figure 2-6 depicts their internal storage.

Figure 2-6



As you can see, `'A'` is a 1-byte element and `"A"` is a 2-byte element. Since characters are really stored as ASCII codes, Figure 2-7 shows what is actually being stored in memory.

Figure 2-7



Because a `char` variable can only hold a single character, it can be assigned the character `'A'`, but not the string `"A"`.

```
char letterOne = 'A';         // This will work.
char letterTwo = "A";         // This will NOT work!
```

You have learned that some strings look like a single character but really aren't. It is also possible to have a character that looks like a string. A good example is the newline character, `\n`. Although it is represented by two characters, a slash and an `n`, it is internally represented as one character. In fact, all escape sequences, internally, are just 1 byte.

Program 2-13 shows the use of `\n` as a character constant, enclosed in single quotation marks. If you refer to the ASCII chart in Appendix A, you will see that ASCII code 10 is the linefeed character. This is the code C++ uses for the newline character.

Program 2-13

```
1 // This program uses character constants.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7    char letter;
```

(program continues)

Program 2-13 *(continued)*

```
 8
 9     letter = 'A';
10     cout << letter << '\n';
11
12     letter = 'B';
13     cout << letter << '\n';
14     return 0;
15 }
```

Program Output

```
A
B
```

Let's review some important points regarding characters and strings:

- Printable characters are internally represented by numeric codes. Most computers use ASCII codes for this purpose.
- Characters normally occupy a single byte of memory.
- Strings are consecutive sequences of characters that occupy consecutive bytes of memory.
- String constants have a null terminator at the end. This marks the end of the string.
- Character constants are enclosed in single quotation marks.
- String constants are enclosed in double quotation marks.
- Escape sequences such as `'\n'` are stored internally as a single character.

2.9 The C++ `string` Class

CONCEPT: Standard C++ provides a special data type for storing and working with strings.

Because a `char` variable can store only one character in its memory location, another data type is needed for a variable able to hold an entire string. While C++ does not have a built-in data type able to do this, Standard C++ provides something called the `string` class that allows the programmer to create a string type variable.

Using the `string` Class

The first step in using the `string` class is to `#include` the `string` header file. This is accomplished with the following preprocessor directive:

```
#include <string>
```

The next step is to define a string type variable, called a string object. Defining a string object is similar to defining a variable of a primitive type. For example, the following statement defines a string object named `movieTitle`.

```
string movieTitle;
```

You can assign a string literal to movieTitle with the assignment operator:

```
movieTitle = "Wheels of Fury";
```

The contents of movieTitle can be displayed on the screen with cout, as shown in the next statement:

```
cout << "My favorite movie is " << movieTitle << endl;
```

Program 2-14 is a complete program that demonstrates the preceding statements.

Program 2-14

```
1  // This program demonstrates the string class.
2  #include <iostream>
3  #include <string>            // Required for the string class.
4  using namespace std;
5
6  int main()
7  {
8     string movieTitle;
9
10    movieTitle = "Wheels of Fury";
11    cout << "My favorite movie is " << movieTitle << endl;
12    return 0;
13 }
```

Program Output

```
My favorite movie is Wheels of Fury
```

As you can see, working with string objects is similar to working with variables of other types. Throughout this text we will continue to discuss string class features and capabilities.



Checkpoint

2.15 What are the ASCII codes for the following characters? (Refer to Appendix A)

```
C
F
W
```

2.16 Which of the following is a character constant?

```
'B'
"B"
```

2.17 Assuming the char data type uses 1 byte of memory, how many bytes do each of the following constants use?

```
'Q'
"Q"
"Sales"
'\n'
```

2.18 What is wrong with the following program statement?

```
char letter = "Z";
```

2.19 What header file must you include in order to use string objects?

2.20 Write a program that stores your name, address, and phone number in three separate string objects. Display the contents of the string objects on the screen.

2.10 Floating-Point Data Types

CONCEPT: Floating-point data types are used to define variables that can hold real numbers.

Whole numbers are not adequate for many jobs. If you are writing a program that works with dollar amounts or precise measurements, you need a data type that allows fractional values. In programming terms, these are called *floating-point* numbers.

Internally, floating-point numbers are stored in a manner similar to *scientific notation*. Take the number 47,281.97. In scientific notation this number is 4.728197×10^4. (10^4 is equal to 10,000, and $4.728197 \times 10{,}000$ is 47,281.97.) The first part of the number, 4.728197, is called the *mantissa*. The mantissa is multiplied by a power of 10.

Computers typically use *E notation* to represent floating-point values. In E notation, the number 47,281.97 would be 4.728197E4. The part of the number before the E is the mantissa, and the part after the E is the power of 10. When a floating-point number is stored in memory, it is stored as the mantissa and the power of 10.

Table 2-7 shows other numbers represented in scientific and E notation.

Table 2-7 Floating-Point Representations

Decimal Notation	Scientific Notation	E Notation
247.91	2.4791×10^2	2.4791E2
0.00072	7.2×10^{-4}	7.2E–4
2,900,000	2.9×10^6	2.9E6

In C++ there are three data types that can represent floating-point numbers. They are

```
float
double
long double
```

The `float` data type is considered *single precision*. The `double` data type is usually twice as big as `float`, so it is considered *double precision*. As you've probably guessed, the `long double` is intended to be larger than the `double`. The exact sizes of these data types is dependent on the computer you are using. The only guarantees are

- A `double` is at least as big as a `float`.
- A `long double` is at least as big as a `double`.

Table 2-8 shows the sizes and ranges of floating-point data types usually found on PCs.

Table 2-8 Floating-Point Data Types on PCs

Data Type	Key Word	Size	Range	Significant Digits
Single precision	`float`	4 bytes	Numbers between ±3.4E-38 and ±3.4E38	7
Double precision	`double`	8 bytes	Numbers between ±1.7E-308 and ±1.7E308	16
Long double precision	`long double`	8 bytes*	Numbers between ±1.7E-308 and ±1.7E308	16

*Some compilers use more than 8 bytes for `long doubles`. These allow greater ranges.

You will notice there are no unsigned floating-point data types. On all machines, variables of the `float`, `double`, and `long double` data type can store both positive and negative numbers. Program 2-15 uses floating-point data types.

Program 2-15

```
1  // This program uses two floating-point data types, float and double.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     float distance = 1.496E8;          // in kilometers
8     double mass = 1.989E30;            // in kilograms
9
10    cout << "The Sun is " << distance << " kilometers away.\n";
11    cout << "The Sun\'s mass is " << mass << " kilograms.\n";
12    return 0;
13 }
```

Program Output

```
The Sun is 1.496e+008 kilometers away.
The Sun's mass is 1.989e+030 kilograms.
```

Floating-Point Constants

Floating-point constants may be expressed in a variety of ways. As shown in Program 2-15, E notation is one method. When you are writing numbers that are extremely large or extremely small, this will probably be the easiest way. E notation numbers may be expressed with an uppercase E or a lowercase e. Notice in the source code the constants were written as 1.496E8 and 1.989E30, but the program printed them as 1.496e+008 and 1.989e+030. The uppercase E and lowercase e are equivalent. The plus sign in front of the exponent is also optional.

You can also express floating-point constants in decimal notation. The constant 1.496E8 could have been written as

```
149600000.0
```

Obviously the E notation is more convenient for lengthy numbers; but for numbers like 47.39, decimal notation is preferable to 4.739E1.

All of the following floating-point constants are equivalent:

```
1.496E8
1.496e8
1.496E+8
1.496e+8
149600000.0
```

Floating-point constants are normally stored in memory as `doubles`. Just in case you need to force a constant to be stored as a `float`, you can append the letter `F` or `f` to the end of it. For example, the following constants would be stored as `float` numbers:

```
1.2F
45.907f
```



NOTE: Because floating-point constants are normally stored in memory as `doubles`, some compilers issue a warning message when you assign a floating-point constant to a `float` variable. For example, assuming `num` is a `float`, the following statement might cause the compiler to generate a warning message:

```
num = 14.725;
```

You can suppress the error message by appending the `f` suffix to the floating-point constant, as shown here:

```
num = 14.725f;
```

If you want to force a value to be stored as a `long double`, append an `L` to it, as shown here:

```
1034.56L
```

The compiler won't confuse this with a long integer because of the decimal point. A lowercase letter `l` can also be used to define a floating-point constant to be a `long double`, but an uppercase `L` is preferable, as the lowercase letter `l` is easily confused with the digit `1`.

Assigning Floating-Point Values to Integer Variables

When a floating-point value is assigned to an integer variable, the fractional part of the value (the part after the decimal point) is discarded. This occurs because an integer variable cannot hold any value containing decimals. For example, look at the following code.

```
int number;
number = 7.8;              // Assigns 7 to number
```

This code attempts to assign the floating-point value 7.8 to the integer variable `number`. Because this is not possible, the value 7 is assigned to `number`, and the fractional part is discarded. When part of a value is discarded in this manner, the value is said to be *truncated*.

Assigning a floating-point variable to an integer variable has the same effect. For example, look at the following code.

```
int intVar;
double doubleVar = 7.8;
intVar = doubleVar;         // Assigns 7 to intVar
                            // doubleVar remains 7.8
```



WARNING! Floating-point variables can hold a much larger range of values than integer variables can. If a floating-point value is stored in an integer variable, and the whole part of the value (the part before the decimal point) is too large for the integer variable, an invalid value will be stored in the integer variable.

2.11 The bool Data Type

CONCEPT: Boolean variables are set to either `true` or `false`.

Expressions that have a `true` or `false` value are called *Boolean* expressions, named in honor of English mathematician George Boole (1815–1864).

The `bool` data type allows you to create variables that hold `true` or `false` values. Program 2-16 demonstrates the definition and use of a `bool` variable. Although it appears that it is storing the *words* true and false, it is actually an integer variable that stores `0` for false and `1` for true, as you can see from the program output.

Program 2-16

```
1  // This program uses Boolean variables.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     bool boolValue;
8
9     boolValue = true;
10    cout << boolValue << endl;
11
12    boolValue = false;
13    cout << boolValue << endl;
14    return 0;
15 }
```

Program Output

```
1
0
```

2.12 Determining the Size of a Data Type

CONCEPT: The `sizeof` operator may be used to determine the size of a data type on any system.

Chapter 1 discussed the portability of the C++ language. As you have seen in this chapter, one of the problems of portability is the lack of common sizes of data types on all machines. If you are not sure what the sizes of data types are on your computer, C++ provides a way to find out.

A special operator called `sizeof` will report the number of bytes of memory used by any data type or variable. Program 2-17 illustrates its use. The first line that uses the operator is line 9.

```
cout << "The size of an integer is " << sizeof(int);
```

The name of the data type or variable is placed inside the parentheses that follow the operator. The operator "returns" the number of bytes used by that item. This operator can be used anywhere you can use an unsigned integer, including in mathematical operations.

Program 2-17

```
1  // This program displays the size of various data types.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     long double apple;
8
9     cout << "The size of an integer is " << sizeof(int);
10    cout << " bytes.\n";
11    cout << "The size of a long integer is " << sizeof(long);
12    cout << " bytes.\n";
13    cout << "An apple can be eaten in " << sizeof(apple);
14    cout << " bytes!\n";
15    return 0;
16 }
```

Program Output

```
The size of an integer is 4 bytes.
The size of a long integer is 4 bytes.
An apple can be eaten in 8 bytes!
```

Checkpoint

2.21 How would the following number in scientific notation be represented in E notation?

6.31×10^{17}

2.22 What will the following code display?

```
int number;
number = 3.625:
cout << number;
```

2.23 Write a program that defines an integer variable named `age` and a `double` variable named `weight`. Store your age and weight, as constants, in the variables. The program should display these values on the screen in a manner similar to the following:

Program Output
My age is 26 and my weight is 168.5 pounds.

(Feel free to lie to the computer about your age and weight. It will never know!)

2.13 More on Variable Assignments and Initialization

CONCEPT: An assignment operation assigns, or copies, a value into a variable. When a value is assigned to a variable as part of the variable's definition, it is called an initialization.



VideoNote
Assignment Statements

As you have already seen in several examples, a value is stored in a variable with an *assignment statement*. For example, the following statement copies the value 12 into the variable `unitsSold`.

```
unitsSold = 12;
```

The = symbol, as you recall, is called the *assignment operator*. Operators perform operations on data. The data that operators work with are called *operands*. The assignment operator has two operands. In the previous statement, the operands are `unitsSold` and `12`.

It is important to remember that in an assignment statement, C++ requires the name of the variable receiving the assignment to appear on the left side of the operator. The following statement is incorrect.

```
12 = unitsSold;  // Incorrect!
```

In C++ terminology, the operand on the left side of the = symbol must be an lvalue. An *lvalue* is something that identifies a place in memory whose contents may be changed, so a new value can be stored there. Most of the time the lvalue will be a variable name. It is called an lvalue because it is a value that may appear on the left-hand side of an assignment operator.

The operand on the right side of the = symbol must be an rvalue. An *rvalue* is any expression that has a value. This could be a single number, like 12, or the result of a calculation, such as 4 + 8. The assignment statement evaluates the expression on the right-hand side to get the value of the rvalue and then puts it in the memory location identified by the lvalue. Both of the following statements assign the value 12 to the `unitsSold` variable.

```
unitsSold = 12;
unitsSold = 4 + 8;
```

You have also seen that it is possible to assign values to variables when they are defined. This is called *initialization*. When multiple variables are defined in the same statement, it is

possible to initialize some of them without having to initialize all of them. Program 2-18 illustrates this.

Program 2-18

```
1  // This program shows variable initialization.
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  int main()
7  {
8     string month = "February";     // month is initialized to "February"
9     int year,                      // year is not initialized
10        days = 28;                  // days is initialized to 28
11
12    year = 2007;                   // Now year is assigned a value
13
14    cout << "In "   << year << " " << month
15         << " had " << days << " days.\n";
16
17    return 0;
18 }
```

Program Output

```
In 2007 February had 28 days.
```

2.14 Scope

CONCEPT: A variable's scope is the part of the program that has access to the variable.

Every variable has a *scope*. The scope of a variable is the part of the program where it may be used. The rules that define a variable's scope are complex, and we will just introduce the concept here. Later in the book we will cover this topic in more depth.

The first rule of scope is that a variable cannot be used in any part of the program before it is defined. Program 2-19 illustrates this.

Program 2-19

```
1  // This program can't find its variable.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     cout << value;    // ERROR! value has not been defined yet!
8
9     int value = 100;
10    return 0;
11 }
```

The program will not work because line 7 attempts to send the contents of the variable `value` to `cout` before the variable is defined. The compiler reads a program from top to bottom. If it encounters a statement that uses a variable before the variable is defined, an error will result. To correct the program, the variable definition must be put before any statement that uses it.

2.15 Arithmetic Operators

CONCEPT: There are many operators for manipulating numeric values and performing arithmetic operations.



VideoNote
Arithmetic Operators

C++ provides many operators for manipulating data. Generally, there are three types of operators: unary, binary, and ternary. These terms reflect the number of operands an operator requires.

Unary operators only require a single operand. For example, consider the following expression: –5

Of course, we understand this represents the value negative five. The constant 5 is preceded by the minus sign. The minus sign, when used this way, is called the *negation operator*. Since it only requires one operand, it is a unary operator.

Binary operators work with two operands. *Ternary operators*, as you may have guessed, require three operands. C++ only has one ternary operator, which will be discussed in Chapter 4.

Arithmetic operations occur frequently in programming. Table 2-9 shows the common arithmetic operators in C++. All are binary operators.

Table 2-9 Fundamental Arithmetic Operators

Operator	Meaning	Example
+	Addition	`total = cost + tax;`
–	Subtraction	`cost = total - tax;`
*	Multiplication	`tax = cost * rate;`
/	Division	`salePrice = original / 2;`
%	Modulus	`remainder = value % 3;`

Here is an example of how each of these operators works.

The *addition operator* returns the sum of its two operands.

```
total = 4 + 8;          // total is assigned the value 12
```

The *subtraction operator* returns the value of its right operand subtracted from its left operand.

```
candyBars = 8 - 3;      // candyBars is assigned the value 5
```

The *multiplication operator* returns the product of its two operands.

```
points = 3 * 7          // points is assigned the value 21
```

The *division operator* works differently depending on whether its operands are integer or floating-point numbers. When both numbers are integers, the division operator performs *integer division*. This means that the result is always an integer. If there is any remainder, it is discarded.

```
fullBoxes = 26 / 8;      // fullBoxes is assigned the value 3
```

The variable `fullBoxes` is assigned the value 3 because 8 goes into 26 three whole times with a remainder of 2. The remainder is discarded.

If you want the division operator to perform regular division, you must make sure at least one of the operands is a floating-point number.

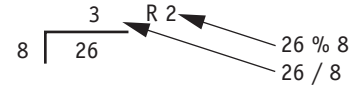
```
boxes = 26.0 / 8;        // boxes is assigned the value 3.25
```

The *modulus operator* computes the *remainder* of doing an integer divide.

```
leftOver = 26 % 8;       // leftOver is assigned the value 2
```

Figure 2-8 illustrates the use of the integer divide and modulus operations.

Figure 2-8 Integer Divide and Modulus Operations



In Chapter 3 you will learn how to use these operators in more complex mathematical formulas. For now we will concentrate on their basic usage. For example, suppose we need to write a program that calculates and displays an employee's total wages for the week. The regular hours for the work week are 40, and any hours worked over 40 are considered overtime. The employee earns $18.25 per hour for regular hours and $27.38 per hour for overtime hours. The employee has worked 50 hours this week. The following pseudocode algorithm shows the program's logic.

```
Regular wages = base pay rate × regular hours
Overtime wages = overtime pay rate × overtime hours
Total wages = regular wages + overtime wages
Display the total wages
```

Program 2-20 shows the C++ code for the program.

Program 2-20

```
1 // This program calculates hourly wages, including overtime.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7    double basePayRate     = 18.25,   // Base pay rate
8           overtimePayRate = 27.38,   // Overtime pay rate
9           regularHours    = 40.0,    // Regular hours worked}
```

(program continues)

Program 2-20 *(continued)*

```
10           overtimeHours   = 10,      // Overtime hours worked
11           regularWages,              // Computed regular wages
12           overtimeWages,             // Computed overtime wages
13           totalWages;                // Computed total wages
14
15    // Calculate regular wages
16    regularWages = basePayRate * regularHours;
17
18    // Calculate overtime wages
19    overtimeWages = overtimePayRate * overtimeHours;
20
21    // Calculate total wages
22    totalWages = regularWages + overtimeWages;
23
24    // Display total wages
25    cout << "Wages for this week are $" << totalWages << endl;
26    return 0;
27 }
```

Program Output

```
Wages for this week are $1003.8
```

Notice that the output displays the wages as $1003.8, with just one digit after the decimal point. In Chapter 3 you will learn to format output so you can control how it displays.



Checkpoint

2.24 Is the following assignment statement valid or invalid? If it is invalid, why?

```
72 = amount;
```

2.25 What is wrong with the following program? How would you correct it?

```
#include <iostream>
using namespace std;
int main()
{
   critter = 62.7;
   double critter;
   cout << critter << endl;
   return 0;
}
```

2.26 What will be assigned to `x` in each of the following statements?

```
x = 8 + 3;
x = 8 - 3;
x = 8 * 3;
x = 8 % 3;
```

2.27 Is the following an example of integer division or floating-point division? What value will be stored in `portion`?

```
portion = 16 / 3;
```

2.16 Comments

CONCEPT: **Comments are notes of explanation that document lines or sections of a program.**

It may surprise you that one of the most important parts of a program has absolutely no impact on the way it runs. We are speaking, of course, of the comments. Comments are part of the program, but the compiler ignores them. They are intended for people who may be reading the source code.

Some programmers resist putting more than just a few comments in their source code. After all, it may seem like enough work to type the parts of the program that actually do something. It is crucial, however, that you develop the habit of thoroughly annotating your code with descriptive comments. It might take extra time now, but it will almost certainly save time in the future.

Imagine writing a program of medium complexity with about 8,000 to 10,000 lines of C++ code. Once you have written the code and satisfactorily debugged it, you happily put it away and move on to the next project. Ten months later you are asked to make a modification to the program (or worse, track down and fix an elusive bug). You pull out the massive pile of paper that contains your source code and stare at thousands of statements only to discover they now make no sense at all. You find variables with names like `z2`, and you can't remember what they are for. If only you had left some notes to yourself explaining all the program's nuances and oddities. But it's too late now. All that's left to do is decide what will take less time: figuring out the old program or completely rewriting it!

This scenario might sound extreme, but it's one you don't want to happen to you. Real-world programs are big and complex. Thoroughly documented programs will make your life easier, not to mention the work of other programmers who may have to read your code in the future. In addition to telling what the program does and describing the purpose of variables, comments can also be used to explain complex procedures in your code and to provide information such as who wrote the program and when it was written or last modified.

Single Line Comments

You have already seen one way to place comments in a C++ program. As was illustrated in Program 2-20, you simply place two forward slashes (`//`) where you want the comment to begin. The compiler ignores everything from that point to the end of the line.

Multi-Line Comments

The second type of comment in C++ is the *multi-line comment*. Multi-line comments start with /* (a forward slash followed by an asterisk) and end with */ (an asterisk followed by a forward slash). Everything between these markers is ignored. Program 2-21 illustrates the use of both a multi-line comment and single line comments. The multi-line comment starts on line 1 with the /* symbol, and ends on line 6 with the */ symbol.

Program 2-21

```
1  /*
2     PROGRAM: PAYROLL.CPP
3     Written by Herbert Dorfmann
4     This program calculates company payroll
5     Last modified: 8/20/2006
6  */
7  #include <iostream>
8  using namespace std;
9
10 int main()
11 {
12    int employeeID;    // Employee ID number
13    double payRate;    // Employees hourly pay rate
14    double hours;      // Hours employee worked this week

   (The remainder of this program is left out.)
```

Notice that unlike a comment started with //, a multi-line comment can span several lines. This makes it more convenient to write large blocks of comments because you do not have to mark every line. On the other hand, the multi-line comment is inconvenient for writing single line comments because you must type both a beginning and ending comment symbol.



NOTE: Many programmers use a combination of single line comments and multi-line comments, as illustrated in the previous sample program. Convenience usually dictates which style to use.

When using multi-line comments:

- Be careful not to reverse the beginning symbol with the ending symbol.
- Be sure not to forget the ending symbol.

Both of these mistakes can be difficult to track down, and will prevent the program from compiling correctly.

2.17 Focus on Software Engineering: *Programming Style*

CONCEPT: **Programming style refers to the way a programmer uses identifiers, spaces, tabs, blank lines, and punctuation characters to visually arrange a program's source code. These are some, but not all, of the elements of programming style.**

In Chapter 1 you learned that syntax rules govern the way a language may be used. The syntax rules of C++ dictate how and where to place key words, semicolons, commas, braces, and other components of the language. The compiler's job is to check for syntax errors and, if there are none, to generate object code.

When the compiler reads a program it processes it as one long stream of characters. The compiler is not influenced by whether each statement is on a separate line, or whether spaces

separate operators from operands. Humans, on the other hand, find it difficult to read programs that aren't written in a visually pleasing manner. Consider Program 2-22 for example.

Program 2-22

```
1 #include <iostream>
2 using namespace std;int main(){double shares=220.0;double
3 avgPrice=14.67;cout
4 <<"There were "<<shares<<" shares sold at $"<<avgPrice<<
5 " per share.\n";return 0;}
```

Program Output

```
There were 220 shares sold at $14.67 per share.
```

Although the program is syntactically correct (it doesn't violate any rules of C++), it is difficult to read. The same program is shown in Program 2-23, written in a more reasonable style.

Program 2-23

```
1  // This program is visually arranged to make it readable.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     double shares = 220.0;
8     double avgPrice = 14.67;
9
10    cout << "There were " << shares << " shares sold at $";
11    cout << avgPrice << " per share.\n";
12    return 0;
13 }
```

Program Output

```
There were 220 shares sold at $14.67 per share.
```

Programming style refers to the way source code is visually arranged. Ideally, it is a consistent method of putting spaces and indentions in a program so visual cues are created. These cues quickly tell a programmer important information about a program.

For example, notice in Program 2-23 that the opening and closing braces of the `main` function align and inside the braces each line is indented. It is a common C++ style to indent all the lines inside a set of braces. You will also notice the blank line between the variable definitions and the `cout` statements. This is intended to visually separate the definitions from the executable statements.



NOTE: Although you are free to develop your own style, you should adhere to common programming practices. By doing so, you will write programs that visually make sense to other programmers and that minimize the likelihood of errors.

Another aspect of programming style is how to handle statements that are too long to fit on one line. Because C++ is a free-flowing language, it is usually possible to spread a statement over several lines. For example, here is a `cout` statement that uses four lines:

```
cout << "The fahrenheit temperature is "
     << fahrenheit
     << " and the celsius temperature is "
     << celsius << endl;
```

This statement works just as if it were typed on one line. You have already seen variable definitions treated similarly:

```
int fahrenheit,
    celsius,
    kelvin;
```

Other issues related to programming style will be presented throughout the book.

2.18 Tying It All Together: *Smile!*

With just the little bit of C++ covered so far, you can print pictures using `cout` statements. Here is the code to make a simple smiley face. Try it!

```
 ^   ^
   *
 \___/
```

Program 2-24

```
1  // This  program prints  a  simple  smiley  face.
2  #include  <iostream>
3  using namespace  std;
4
5  int main()
6  {
7     cout << "\n\n";
8     cout << "     ^   ^  \n";
9     cout << "       *    \n";
10    cout << "     \\___/  \n";
11    return 0;
12 }
```

Now try revising Program 2-24 to make faces like these.

```
 o   o      ^   ^
   o        v   v
 (___)        :
             \UUU/
```

Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. Every complete statement ends with a _____.
2. To use `cout` statements you must include the _____ file in your program.
3. Every C++ program must have a function named _____.
4. Preprocessor directives begin with a _____.
5. A group of statements, such as the body of a function, must be enclosed in _____.
6. `72`, `'A'`, and `"Hello World"` are all examples of _____.
7. 978.65×10^{12} would be written in E notation as _____.
8. The character constant `'A'` requires _____ byte(s) of memory, whereas the string constant `"A"` requires _____ byte(s).
9. Which of the following are *not* valid assignment statements?

   A) `total = 9;`
   B) `72 = amount;`
   C) `yourAge = myAge;`
10. If the variable `letter` has been defined as a `char` variable, which of the following are *not* valid assignment statements?

    A) `letter = w;`
    B) `letter = 'w';`
    C) `letter = "w";`
11. Which of the following are *not* valid `cout` statements?

    A) `cout << "Hello" << endl;`
    B) `cout << "Hello" << \n;`
    C) `cout << Hello;`
12. Which of the following are *not* valid `cout` statements?

    A) `cout << "Hello world";`
    B) `cout << Hello world;`
    C) `cout << "Hello" << " world";`
13. Assume `x = 4`, `y = 7`, and `z = 2`. What value will be stored in integer variable `result` by each of the following statements?

    A) `result = x + y;`
    B) `result = y * 2;`
    C) `result = y / z;`
14. Assume `x = 2.5`, `y = 7.0`, and `z = 3`. What value will be stored in integer variable `result` by each of the following statements?

    A) `result = x + y;`
    B) `result = y * 2;`
    C) `result = y / z;`

15. Write a C++ statement that defines the `double` variables `temp`, `weight`, and `height` all in the same statement.
16. Write a C++ statement that defines the `int` variables `months`, `days`, and `years` all in the same statement, with `months` initialized to 2 and `years` initialized to 3.
17. Write assignment statements that perform the following operations with `int` variable `i`, `double` variables `d1` and `d2`, and `char` variable `c`.
    A) Add 2 to `d1` and store the result in `d2`.
    B) Multiply `d2` time 4 and store the result in `d1`.
    C) Store the character `'K'` in `c`.
    D) Store the ASCII code for the character `'K'` in `i`.
    E) Subtract 1 from `i` and store the result back in `i`.
18. Write assignment statements that perform the following operations with `int` variable `i`, `double` variables `d1` and `d2`, and `char` variable `c`.
    A) Subtract 8.5 from `d2` and store the result in `d1`.
    B) Divide `d1` by 3.14 and store the result in `d2`.
    C) Store the ASCII code for the character `'F'` in `c`.
    D) Add 1 to `i` and store the new value back in `i`.
    E) Add `d1` to the current value of `d2` and store the result back in `d2` as its new value.
19. Modify the following program segment so it prints two blank lines between each line of text.

```
cout << "Two mandolins like creatures in the";
cout << "dark";
cout << "Creating the agony of ecstasy.";
cout << "                    - George Barker";
```

20. Rewrite the follow statement to use the newline escape character, instead of an `endl`, each time subsequent output is to be displayed on a new line.

```
cout << "L" << endl
     << "E" << endl
     << "A" << endl
     << "F" << endl;
```

Algorithm Workbench

21. Create detailed pseudocode for a program that calculates how many days are left until Christmas, when given as an input how many weeks are left until Christmas. Use variables named `weeks` and `days`.
22. Create detailed pseudocode for a program that determines how many full 12-egg cartons of eggs a farmer can pack when given as an input the number of eggs he has collected on a given day. Use variables named `eggs` and `cartons`.
23. Create detailed pseudocode for a program that determines distance traveled when given inputs of speed and time. Use variables named `speed`, `time`, and `distance`.
24. Create detailed pseudocode for a program that determines miles per gallon a vehicle gets when given inputs of miles traveled and gallons of gas used. Use variables named `miles`, `gallons`, and `milesPerGallon`.

Predict the Output

25. What will the following programs print on the screen?

A)
```
#include <iostream>
using namespace std;
int main()
{
   int freeze = 32, boil = 212;
   freeze = 0;
   boil = 100;
   cout << freeze << endl << boil << endl;
   return 0;
}
```

B)
```
#include <iostream>
using namespace std;
int main()
{
   int x = 0, y = 2;
   x = y * 4;
   cout << x << endl << y << endl;
   return 0;
}
```

C)
```
#include <iostream>
using namespace std;
int main()
{
   cout << "I am the incredible";
   cout << "computing\nmachine";
   cout << "\nand I will\namaze\n";
   cout << "you.\n";
   return 0;
}
```

26. A)
```
#include <iostream>
using namespace std;

int main()
{
   cout << "Be careful!\n";
   cout << "This might/n be a trick ";
   cout << "question.\n";
   return 0;
}
```

B)
```
#include <iostream>
using namespace std;

int main()
{
   int a, x = 23;

   a = x % 2;
   cout << x << endl << a << endl;
   return 0;
}
```

Find the Error

27. There are a number of syntax errors in the following program. Locate as many as you can.

```
*/ What's wrong with this program? /*
#include iostream
using namespace std;

int main();
}
   int a, b, c\\ Three integers
   a = 3
   b = 4
   c = a + b
   Cout < "The value of c is %d" < C;
   return 0;
{
```

Soft Skills

Programmers need good communication skills as well as good analytical and problem-solving skills. Good communication can minimize misunderstandings that easily arise when expectations of different individuals involved in a project are not clearly enough articulated before the project begins. A detailed set of project specifications can clarify the scope of a project, what interaction will occur between the user and the program, and exactly what the program will and will not do.

28. Pair up with another student in the class. One of you is the client and the other is the software developer. Briefly discuss a simple program the client wants that the programmer will create. Here are some possible ideas.
    - the paint problem described in the Chapter 1 Soft Skills exercise
    - a program that can halve the quantities of ingredients for a recipe
    - a program that determines how long it will take to drive from point A to point B

Once you have decided on a program, you should independently, with no further communication, each write down detailed specifications. The client writes down exactly what he wants the program to do and the developer writes down her understanding of exactly what the program will do. When you are done, compare what you have written. Rarely will the two agree.

Now discuss the discrepancies and see if you can come to a clear understanding of exactly what the program must do. Together create a program specification sufficiently detailed that both of you believe it leaves no room for misunderstanding.

Programming Challenges

1. Sum of Two Numbers

Write a program that stores the integers 62 and 99 in variables, and stores the sum of these two in a variable named `total`. Display the total on the screen.

2. Sales Prediction

The East Coast sales division of a company generates 62 percent of total sales. Based on that percentage, write a program that will predict how much the East Coast division will generate if the company has $4.6 million in sales this year. Display the result on the screen.



3. Sales Tax

Write a program that computes the total sales tax on a $52 purchase. Assume the state sales tax is 4 percent and the county sales tax is 2 percent. Display the purchase price, state tax, county tax, and total tax amounts on the screen.



VideoNote
Solving the Restaurant Bill Problem

4. Restaurant Bill

Write a program that computes the tax and tip on a restaurant bill for a patron with a $44.50 meal charge. The tax should be 6.75 percent of the meal cost. The tip should be 15 percent of the total after adding the tax. Display the meal cost, tax amount, tip amount, and total bill on the screen.

5. Cyborg Data Type Sizes

You have been given a job as a programmer on a Cyborg supercomputer. In order to accomplish some calculations, you need to know how many bytes the following data types use: `char`, `int`, `float`, and `double`. You do not have any manuals, so you can't look up this information. Write a C++ program that will determine the amount of memory used by these types and display the information on the screen.



6. Miles per Gallon

A car holds 16 gallons of gasoline and can travel 350 miles before refueling. Write a program that calculates the number of miles per gallon the car gets. Display the result on the screen.

7. Distance per Tank of Gas

A car with a 20 gallon gas tank averages 21.5 miles per gallon when driven in town and 26.8 miles per gallon when driven on the highway. Write a program that calculates and displays the distance the car can travel on one tank of gas when driven in town and when driven on the highway.

8. Land Calculation

In the United States, land is often measured in square feet. In many other countries it is measured in square meters. One acre of land is equivalent to 43,560 square feet. A square meter is equivalent to 10.7639 square feet. Write a program that computes and displays the number of square feet and the number of square meters in $\frac{1}{4}$ acre of land.

Hint: Because a square meter is larger than a square foot, there will be fewer square meters in $\frac{1}{4}$ acre than there are square feet.

9. Circuit Board Price

An electronics company sells circuit boards at a 40 percent profit. Write a program that calculates the selling price of a circuit board that costs them $12.67 to produce. Display the result on the screen.

10. Personal Information

Write a program that displays the following information, each on a separate line:

Your name
Your address, with city, state, and zip code
Your telephone number
Your college major

Use only a single `cout` statement to display all of this information.



11. Triangle Pattern

Write a program that displays the following pattern on the screen:

```
   *
  ***
 *****
*******
```

12. Diamond Pattern

Write a program that displays the following pattern on the screen:

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

13. Pay Period Gross Pay

A particular employee earns $32,500 annually. Write a program that determines and displays what the amount of his gross pay will be for each pay period if he is paid twice a month (24 pay checks per year) and if he is paid bi-weekly (26 checks per year).

14. Basketball Player Height

The star player of a high school basketball team is 73 inches tall. Write a program to compute and display the height in feet / inches form.

Hint: Try using the modulus and integer divide operations.

15. Stock Loss

Kathryn bought 600 shares of stock at a price of $21.77 per share. A year later she sold them for just $16.44 per share. Write a program that calculates and displays the following:

- The total amount paid for the stock.
- The total amount received from selling the stock.
- The total amount of money she lost.

16. Energy Drink Consumption

A soft drink company recently surveyed 12,467 of its customers and found that approximately 14 percent of those surveyed purchase one or more energy drinks per week. Of those customers who purchase energy drinks, approximately 64 percent of them prefer citrus flavored energy drinks. Write a program that displays the following:

- The approximate number of customers in the survey who purchase one or more energy drinks per week.
- The approximate number of customers in the survey who prefer citrus flavored energy drinks.

17. Past Ocean Levels

The Earth's ocean levels have risen an average of 1.8 millimeters per year over the past century. Write a program that computes and displays the number of centimeters and number of inches the oceans rose during this time. One millimeter is equivalent to 0.1 centimeters. One centimeter is equivalent to 0.3937 inches.

18. Future Ocean Levels

During the past decade ocean levels have been rising faster than in the past, an average of approximately 3.1 millimeters per year. Write a program that computes how much ocean levels are expected to rise during the next 20 years if they continue rising at this rate. Display the answer in both centimeters and inches.

CHAPTER 3

Expressions and Interactivity

TOPICS

3.1 The `cin` Object
3.2 Mathematical Expressions
3.3 Implicit Type Conversion
3.4 Explicit Type Conversion
3.5 Overflow and Underflow
3.6 Named Constants
3.7 Multiple and Combined Assignment
3.8 Formatting Output
3.9 Working with Characters and String Objects
3.10 Using C-Strings
3.11 More Mathematical Library Functions
3.12 Introduction to Files
3.13 Focus on Debugging: *Hand Tracing a Program*
3.14 Green Fields Landscaping Case Study—Part 1
3.15 Tying It All Together: *Word Game*

3.1 The `cin` Object

CONCEPT: `cin` can be used to read data typed at the keyboard.

So far you have written programs with built-in information. You have initialized the variables with the necessary starting values without letting the user enter his or her own data. These types of programs are limited to performing their task with only a single set of starting information. If you decide to change the initial value of any variable, the program must be modified and recompiled.



VideoNote
Using `cin` to Read Input

In reality, most programs ask for values that will be assigned to variables. This means the program does not have to be modified if the user wants to run it several times with different sets of information. For example, a program that calculates the area of a circle might ask the user to enter the circle's radius. When the circle area has been computed and printed, the program could be run again and a different radius could be entered.

Just as C++ provides the `cout` object to produce console output, it provides an object named `cin` that is used to read console input. (You can think of the word `cin` as meaning console **in**put.) Program 3-1 shows `cin` being used to read values input by the user.

Program 3-1

```
1  // This program calculates and displays the area of a rectangle.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int length, width, area;
8
9     cout << "This program calculates the area of a rectangle.\n";
10
11    // Have the user input the rectangle's length and width
12    cout << "What is the length of the rectangle? ";
13    cin  >> length;
14    cout << "What is the width of the rectangle? ";
15    cin  >> width;
16
17    // Compute and display the area
18    area = length * width;
19    cout << "The area of the rectangle is " << area << endl;
20    return 0;
21 }
```

Program Output with Example Input Shown in Bold

```
This program calculates the area of a rectangle.
What is the length of the rectangle? 10[Enter]
What is the width of the rectangle? 20[Enter]
The area of the rectangle is 200.
```

Instead of calculating the area of one rectangle, this program can be used to compute the area of any rectangle. The values that are stored in the `length` and `width` variables are entered by the user when the program is running. Look at lines 12 and 13.

```
cout << "What is the length of the rectangle? ";
cin  >> length;
```

In line 12 `cout` is used to display the question "What is the length of the rectangle?" This is called a *prompt*. It lets the user know that an input is expected and prompts them as to what must be entered. When `cin` will be used to get input from the user, it should always be preceded by a prompt.

Line 13 uses `cin` to read a value from the keyboard. The >> symbol is the *stream extraction operator*. It gets characters from the stream object on its left and stores them in the variable whose name appears on its right. In this example line, the characters read in by `cin` are taken from the `cin` object and stored in the `length` variable.

Gathering input from the user is normally a two-step process:

1. Use `cout` to display a prompt on the screen.
2. Use `cin` to read a value from the keyboard.

The prompt should ask the user a question, or tell the user to enter a specific value. For example, the code we just examined from Program 3-1 displays the following prompt:

```
What is the length of the rectangle?
```

This tells the user to enter the rectangle's length. After the prompt displays, the program uses `cin` to read a value from the keyboard and store it in the `length` variable.

Notice that the << and >> operators appear to point in the direction that data is flowing. It may help to think of them as arrows. In a statement that uses `cout`, the << operator always points toward `cout`, as shown here. This indicates that data is flowing from a variable or a literal to the `cout` object.

```
cout << "What is the length of the rectangle? ";
cout ← "What is the length of the rectangle? ";
```

In a statement that uses `cin`, the >> operator always points toward the variable receiving the value. This indicates that data is flowing from the `cin` object to a variable.

```
cin >> length;
cin → length;
```

The `cin` object causes a program to wait until data is typed at the keyboard and the [Enter] key is pressed. No other lines will be executed until `cin` gets its input.

When the user enters characters from the keyboard, they are temporarily placed in an area of memory called the *input buffer*, or *keyboard buffer*. `cin` automatically converts this data to the data type of the variable it is to be stored in. If the user types 10, it is read as the characters '1' and '0', but `cin` is smart enough to know this will have to be converted to the `int` value 10 before it is stored in `length`. If the user enters a floating-point number like 10.7, however, there is a problem. `cin` knows such a value cannot be stored in an integer variable, so it stops reading when it gets to the decimal point, leaving the decimal point and the rest of the digits in the input buffer. This can cause a problem when the next value is read in. Program 3-2 illustrates this problem.

Program 3-2

```
1  // This program illustrates what can happen when a
2  // floating-point number is entered for an integer variable.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int intNumber;
9     double floatNumber;
10
11    cout << "Input a number. ";
12    cin  >> intNumber;
13    cout << "Input a second number.\n";
14    cin  >> floatNumber;
15    cout << "You entered: " << intNumber
16         << " and " << floatNumber << endl;
17
18    return 0;
19 }
```

(program continues)

Program 3-2 *(continued)*

Program Output with Example Input Shown in Bold

```
Input a number. 12.3[Enter]
Input a second number.
You entered: 12 and 0.3
```

Let's look more closely at what occurred in Program 3-2. When prompted for the first number, the user entered 12.3 from the keyboard. However, because `cin` was reading a value into `intNumber`, an integer variable, it stopped reading when it got to the decimal point and a 12 was stored in `intNumber`. When the second `cin` statement needed a value to read into `floatNumber`, it found that it already had a value in the input buffer, the .3 left over from the user's first input. Instead of waiting for the user to enter a second number, the .3 was read in and stored in `floatNumber`.

Later you will learn how to prevent something like this from happening, but for now this illustrates the need to provide the user with clear prompts. If the user had been specifically prompted to enter an integer for the first number, there would have been less chance of a problem occurring.



NOTE: You must include the `iostream` file in any program that uses `cin`.

Entering Multiple Values

You can use `cin` to input multiple values at once. Look at Program 3-3, which is a modified version of Program 3-1.

Program 3-3

```
1  // This program calculates and displays the area of a rectangle.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int length, width, area;
8
9     cout << "This program calculates the area of a rectangle.\n";
10
11    // Have the user input the rectangle's length and width
12    cout << "Enter the length and width of the rectangle ";
13    cout << "separated by a space.\n";
14    cin  >> length >> width;
15
16    // Compute and display the area
17    area = length * width;
18    cout << "The area of the rectangle is " << area << endl;
19    return 0;
20 }
```

(program continues)

Program 3-3 *(continued)*

Program Output with Example Input Shown in Bold

```
This program calculates the area of a rectangle.
Enter the length and width of the rectangle separated by a space.
10 20[Enter]
The area of the rectangle is 200
```

Line 14 waits for the user to enter two values. The first is assigned to `length` and the second to `width`.

```
cin >> length >> width;
```

In the example output, the user entered 10 and 20, so 10 is stored in `length` and 20 is stored in `width`.

Notice the user separates the numbers by spaces as they are entered. This is how `cin` knows where each number begins and ends. It doesn't matter how many spaces are entered between the individual numbers. For example, the user could have entered

```
10          20
```



NOTE: The [Enter] key is pressed after the last number is entered.

`cin` will also read multiple values of different data types. This is shown in Program 3-4.

Program 3-4

```
1  // This program demonstrates how cin can read multiple values
2  // of different data types.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int whole;
9     double fractional;
10    char letter;
11
12    cout << "Enter an integer, a double, and a character: ";
13    cin  >> whole >> fractional >> letter;
14
15    cout << "whole: " << whole << endl;
16    cout << "fractional: " << fractional << endl;
17    cout << "letter: " << letter << endl;
18    return 0;
19 }
```

Program Output with Example Input Shown in Bold

```
Enter an integer, a double, and a character: 4 5.7 b[Enter]
whole: 4
fractional: 5.7
letter: b
```

As you can see in the example output, the values are stored in the order entered in their respective variables.

But what if the user had entered the values in the wrong order, as shown in the following sample run?

Program 3-4 Output with Different Example Input Shown in Bold

```
Enter an integer, a double, and a character: 5.7 4 b[Enter]
whole: 5
fractional: 0.7
letter: 4
```

Because the data was not entered in the specified order, there is a complete mix-up of what value is stored for each variable. The cin statement on line 13 reads `5` for `int` variable `whole`, `.7` for `double` variable `fractional`, and `4` for `char` variable `letter`. The character `b` is left in the input buffer. For a program to function correctly it is important that the user enter data values in the order the program expects to receive them and that a floating-point number not be entered when an integer is expected.



Checkpoint

3.1 What header file must be included in programs using `cin`?

3.2 What is the >> symbol called?

3.3 Where does `cin` read its input from?

3.4 True or False: `cin` requires the user to press the [Enter] key after entering data.

3.5 Assume `value` is an integer variable. If the user enters 3.14 in response to the following programming statement, what will be stored in `value`?

```
cin >> value;
```

3.6 A program has the following variable definitions.

```
long miles;
int feet;
double inches;
```

Write a single `cin` statement that reads a value into each of these variables.

3.7 The following program will run, but the user will have difficulty understanding what to do. How would you improve the program?

```
// This program multiplies two numbers and displays the result.
#include <iostream>
using namespace std;

int main()
{
   double first, second, product;
   cin >> first >> second;
   product = first * second;
   cout << product;
   return 0;
}
```

3.8 Complete the following program skeleton so it asks for the user's weight (in pounds) and displays the equivalent weight in kilograms.

```
#include <iostream>
using namespace std;

int main()
{
   double pounds, kilograms;

   // Write a prompt to tell the user to enter his or her weight
   // in pounds.
   // Write code here that reads in the user's weight in pounds.
   // The following line does the conversion.

   kilograms = pounds / 2.2;

   // Write code here that displays the user's weight in kilograms.

   return 0;
}
```

3.2 Mathematical Expressions

CONCEPT: **C++ allows you to construct complex mathematical expressions using multiple operators and grouping symbols.**



VideoNote
Evaluating Mathematical Expressions

In Chapter 2 you were introduced to the basic mathematical operators, which are used to build mathematical *expressions*. An *expression* is a programming statement that has a value. Usually, an expression consists of an operator and its operands. Look at the following statement:

```
sum = 21 + 3;
```

Since 21 + 3 has a value, it is an expression. Its value, 24, is stored in the variable `sum`. Expressions do not have to be in the form of mathematical operations. In the following statement, 3 is an expression.

```
number = 3;
```

Here are some programming statements where the variable `result` is being assigned the value of an expression. They are called assignment statements.

```
result = x;
result = 4;
result = 15 / 3;
result = 22 * number;
result = sizeof(int);
result = a + b + c;
```

In each of these statements, a number, variable name, or mathematical expression appears on the right side of the = symbol. A value is obtained from each of these and stored in the

variable `result`. These are all examples of a variable being assigned the value of an expression.

Although some instructors prefer that you not perform mathematical operations within a `cout` statement, it is possible to do so. Program 3-5 illustrates how to do this.

Program 3-5

```
1  // This program displays the decimal value of a fraction.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     double numerator, denominator;
8
9     cout << "This program shows the decimal value of a fraction.\n";
10
11    // Have the user enter the numerator and denominator
12    cout << "Enter the numerator: ";
13    cin  >> numerator;
14    cout << "Enter the denominator: ";
15    cin  >> denominator;
16
17    // Compute and display the decimal value
18    cout << "The decimal value is "<< (numerator / denominator) << endl;
19    return 0;
20 }
```

Program Output with Example Input Shown in Bold

```
This program shows the decimal value of a fraction.
Enter the numerator: 3[Enter]
Enter the denominator: 16[Enter]
The decimal value is 0.1875
```

The `cout` object can display the value of any legal expression in C++. In Program 3-5 the value of the expression `numerator / denominator` is displayed.



NOTE: The Program 3-5 example input shows the user entering 3 and 16. Because these values are assigned to `double` variables, they are stored as 3.0 and 16.0.



NOTE: When sending an expression that includes an operator to `cout`, it is always a good idea to put parentheses around the expression. Some operators will yield unexpected results otherwise.

Operator Precedence

It is possible to build mathematical expressions with several operators. The following statement assigns the sum of 17, *x*, 21, and *y* to the variable `answer`.

```
answer = 17 + x + 21 + y;
```

Some expressions are not that straightforward, however. Consider the following statement:

```
outcome = 12 + 6 / 3;
```

What value will be stored in `outcome`? It could be assigned either 6 or 14, depending on whether the addition operation or the division operation takes place first. The answer is 14 because the division operator has higher *precedence* than the addition operator. This is exactly the same as the operator precedence found in algebra.

Mathematical expressions are evaluated from left to right. However, when there are two operators and one has higher precedence than the other, it is done first. Multiplication and division have higher precedence than addition and subtraction, so the example statement works like this:

A) 6 is divided by 3, yielding a result of 2

B) 12 is added to 2, yielding a result of 14

It could be diagrammed in the following way:

```
12 + 6 / 3
      ↓
12 +  2
  ↓
  14
```

Table 3-1 shows the precedence of the arithmetic operators. The operators at the top of the table have higher precedence than the ones below it.

Table 3-1 Precedence of Arithmetic Operators (Highest to Lowest)

( )		Expressions within parentheses are evaluated first
–	unary	Negation of a value, e.g., -6
* / %	binary	Multiplication, division, and modulus
+ –	binary	Addition and subtraction

The multiplication, division, and modulus operators have the same precedence. This is also true of the addition and subtraction operators. Table 3-2 shows some expressions with their values.

Table 3-2 Some Expressions

Expression	Value
`5 + 2 * 4`	13
`10 / 2 - 3`	2
`8 + 12 * 2 - 4`	28
`4 + 17 % 2 - 1`	4
`6 - 3 * 2 + 7 - 1`	6

Associativity

Associativity is the order in which an operator works with its operands. Associativity is either *left to right* or *right to left*. The associativity of the division operator is left to right, so it divides the operand on its left by the operand on its right. Table 3-3 shows the arithmetic operators and their associativity.

Table 3-3 Associativity of Arithmetic Operators

Operator	Associativity
`(unary negation) -`	Right to left
`* / %`	Left to right
`+ -`	Left to right

Grouping with Parentheses

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the following statement, the sum of `a` plus `b` is divided by 4.

```
average = (a + b) / 4;
```

Without the parentheses `b` would be divided by 4 **before** adding `a` to the result. Table 3-4 shows more expressions and their values.

Table 3-4 More Arithmetic Expressions

Expression	Value
`(5 + 2) * 4`	28
`10 / (5 - 3)`	5
`8 + 12 * (6 - 2)`	56
`(4 + 17) % 2 - 1`	0
`(6 - 3) * (2 + 7) / 3`	9

Converting Algebraic Expressions to Programming Statements

In algebra it is not always necessary to use an operator for multiplication. C++, however, requires an operator for any mathematical operation. Table 3-5 shows some algebraic expressions that perform multiplication and the equivalent C++ expressions.

Table 3-5 Algebraic and C++ Multiplication Expressions

Algebraic Expression	Operation	C++ Equivalent
6B	6 times B	`6 * B`
(3)(12)	3 times 12	`3 * 12`
4xy	4 times x times y	`4 * x * y`

When converting some algebraic expressions to C++, you may have to insert parentheses that do not appear in the algebraic expression. For example, look at the following expression:

$$x = \frac{a + b}{c}$$

To convert this to a C++ statement, $a + b$ will have to be enclosed in parentheses:

```
x = (a + b) / c;
```

Table 3-6 shows more algebraic expressions and their C++ equivalents.

Table 3-6 Algebraic and C++ Expressions

Algebraic Expression	C++ Expression
$y = 3\frac{x}{2}$	`y = x / 2 * 3;`
$z = 3bc + 4$	`z = 3 * b * c + 4;`
$a = \frac{3x + 2}{4a - 1}$	`a = (3 * x + 2) / (4 * a - 1)`

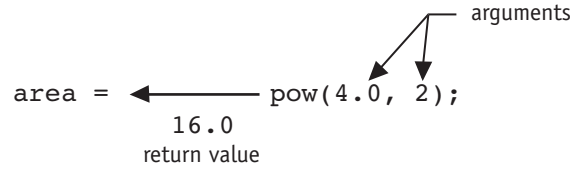
No Exponents Please!

Unlike many programming languages, C++ does not have an exponent operator. Raising a number to a power requires the use of a *library function*. The C++ library isn't a place where you check out books, but a collection of specialized functions. Think of a library function as a "routine" that performs a specific operation. One of the library functions is called `pow`, and its purpose is to raise a number to a power. Here is an example of how it's used:

```
area = pow(4.0, 2);
```

This statement contains a *call* to the `pow` function. The numbers inside the parentheses are *arguments*. Arguments are information being sent to the function. The `pow` function always raises the first argument to the power of the second argument. In this example, 4.0 is raised to the power of 2. The result is *returned* from the function and used in the statement where the function call appears. The `pow` function expects floating-point arguments. On some C++ compilers integer arguments will also work, but since many compilers require that at least the first argument be a `double`, that is the convention we use in this book. The value returned from the function is always a `double` number. In this case, 16.0 is returned from `pow` and assigned to the variable `area`. This is illustrated in Figure 3-1.

Figure 3-1



The statement `area = pow(4.0, 2)` is equivalent to the following algebraic statement:

$$area = 4^2$$

Here is another example of a statement using the `pow` function. It assigns 3 times 6^3 to x:

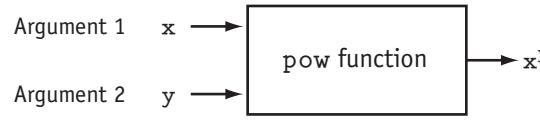
```
x = 3 * pow(6.0, 3);
```

And the following statement displays the value of 5 raised to the power of 4:

```
cout << pow(5.0, 4);
```

It might be helpful to think of `pow` as a "black box" that accepts two numbers and then sends a third number out. The number that comes out has the value of the first number raised to the power of the second number, as illustrated in Figure 3-2.

Figure 3-2



There are some guidelines that should be followed when the `pow` function is used. First, the program must include the `cmath` header file. Second, at least the first of the two arguments you pass to the function, if not both, should be a `double`. Third, because the `pow` function returns a `double` value, any variable that value is assigned to should also be a `double`. For example, in the following statement the variable `area` should be defined as a `double`:

```
area = pow(4.0, 2);
```

Program 3-6 solves a simple algebraic problem. It asks the user to enter the radius of a circle and then calculates the area of the circle. The formula is

$$\text{Area} = \pi r^2$$

which is expressed in the program as

```
area = 3.14159 * pow(radius, 2);
```

Program 3-6

```
1  // This program calculates the area of a circle. The formula for the
2  // area of a circle is PI times the radius squared. PI is 3.14159.
3  #include <iostream>
4  #include <cmath>     // Needed for the pow function
5  using namespace std;
6
7  int main()
8  {
9     double area, radius;
10
11    cout << "This program calculates the area of a circle.\n";
12
```

(program continues)

Program 3-6 *(continued)*

```
13    // Get the radius
14    cout << "What is the radius of the circle? ";
15    cin  >> radius;
16
17    // Compute and display the area
18    area = 3.14159 * pow(radius, 2);
19    cout << "The area is " << area << endl;
20    return 0;
21 }
```

Program Output with Example Input Shown in Bold

```
This program calculates the area of a circle.
What is the radius of the circle? 10[Enter]
The area is 314.159
```



NOTE: Program 3-6 is presented as a demonstration of the pow function. In reality, there is no reason to use this function in such a simple operation. Line 18 could just as easily be written

```
area = 3.14159 * radius * radius;
```

The pow function is useful, however, in operations that involve larger exponents.



Checkpoint

3.9 In each of the following cases, tell which operator has higher precedence or whether they have the same precedence.

A) + and *
B) * and /
C) / and %

3.10 Complete the following table by writing the value of each expression in the Value column.

Expression	Value
6 + 3 * 5	
12 / 2 - 4	
9 + 14 * 2 - 6	
5 + 19 % 3 - 1	
(6 + 2) * 3	
14 / (11 - 4)	
9 + 12 * (8 - 3)	
(6 + 17) % 2 - 1	
(9 - 3) * (6 + 9) / 3	

3.11 Write C++ expressions for the following algebraic expressions:

$y = 6x$ $\quad a = 2b + 4c$ $\quad y = x^3$

$g = \dfrac{x + 2}{z^2}$ $\quad y = \dfrac{x^2}{z^2}$

3.12 Study the following program *and* complete the table.

```
#include <iostream>
#include <cmath>

int main()
{
   double value1, value2, value3;
   cout << "Enter a number: ";
   cin  >> value1;
   value2 = 2 * pow(value1, 2);
   value3 = 3 + value2 / 2 - 1;
   cout << value3;
   return 0;
}
```

If the user enters…	The program will display what number (stored in `value3`)?
2	
5	
4.3	
6	

3.13 Complete the following program skeleton so it displays the volume of a cylindrical fuel tank. The formula for the volume of a cylinder is

Volume = $\pi r^2 h$

where
π is 3.14159
r is the radius of the tank
h is the height of the tank

```
#include <iostream>
#include <cmath>

int main()
{
   double volume, radius, height;
   cout << "This program will tell you the volume of\n";
   cout << "a cylinder-shaped fuel tank.\n";
   cout << "How tall is the tank? ";
   cin  >> height;
   cout << "What is the radius of the tank? ";
   cin  >> radius;

   // You must complete the program.
   return 0;
}
```

3.3 Implicit Type Conversion

CONCEPT: **When an operator's operands are of different data types, C++ will automatically convert them to the same data type. This can affect the results of mathematical expressions.**

If a floating-point value is assigned to an `int` variable, what value will the variable receive? If an `int` is multiplied by a `float`, what data type will the result be? What if a `double` is divided by an `unsigned int`? Is there any way of predicting what will happen in these instances? The answer is yes. C++ follows a set of rules when performing mathematical operations on variables of different data types. It's helpful to understand these rules to prevent subtle errors from creeping into your programs.

Just like officers in the military, data types are ranked. One data type outranks another if it can hold a larger number. For example, a `float` outranks an `int` and a `double` outranks a `float`. Table 3-7 lists the data types in order of their rank, from highest to lowest.

Table 3-7 Data Type Ranking

`long double`
`double`
`float`
`unsigned long`
`long`
`unsigned int`
`int`
`unsigned short`
`short`
`char`

One exception to the ranking in Table 3-7 is when an `int` and a `long` are the same size. In that case, an `unsigned int` outranks `long` because it can hold a higher value.

When C++ is working with an operator, it strives to convert the operands to the same type. This implicit, or automatic, conversion is known as *type coercion*. When a value is converted to a higher data type, it is said to be *promoted*. To *demote* a value means to convert it to a lower data type. Let's look at the specific rules that govern the evaluation of mathematical expressions.

Rule 1: `char`, `short`, and `unsigned short` are automatically promoted to `int`.

Anytime these data types are used in a mathematical expression, they are automatically promoted to an `int`.*

* The only exception to this rule is when an `unsigned short` holds a value larger than can be held by an `int`. This can happen on systems where a `short` is the same size as an `int`. In this case, the `unsigned short` is promoted to `unsigned int`.

Rule 2: When an operator works with two values of different data types, the lower-ranking value is promoted to the type of the higher-ranking value.

In the following expression, assume that `years` is an `int` and `interestRate` is a `double`:

```
years * interestRate
```

Before the multiplication takes place, the value in `years` will be promoted to a `double`.

Rule 3: When the final value of an expression is assigned to a variable, it will be converted to the data type of that variable.

This means that if the variable receiving the value is of a lower data type than the value it is receiving, the value will be demoted to the type of the variable. If the variable's data type does not have enough storage space to hold the value, part of the value will be lost, and the variable could receive an inaccurate result. As mentioned in Chapter 2, if the variable receiving the value is an integer and the value being assigned to it is a floating-point number, the value will be *truncated* before being stored in the variable. This means everything after the decimal point will be discarded:

```
int x;
double y = 3.75;
x = y;                  // x is assigned 3
                        // y remains 3.75
```

If the variable receiving the value has a higher data type than the value being assigned to it, there is no problem.

In the following statement, assume that `area` is a `long int`, while `length` and `width` are both `int`:

```
area = length * width;
```

Because `length` and `width` are both an `int`, they will not be converted to any other data type. The result of the multiplication, however, will be converted to `long` so it can be stored in `area`.

3.4 Explicit Type Conversion

CONCEPT: **Type casting allows you to explicitly perform data type conversion.**

A *type cast expression* lets you manually promote or demote a value. The general format of a type cast expression is

```
static_cast<DataType>(Value)
```

where *`Value`* is a variable or literal value that you wish to convert and *`DataType`* is the data type you wish to convert it to. Here is an example of code that uses a type cast expression:

```
double number = 3.7;
int val;
val = static_cast<int>(number);
```

This code defines two variables: `number`, a `double`, and `val`, an `int`. The type cast expression in the third statement returns a copy of the value in `number`, converted to an `int`. When a `double` or `float` is converted to an `int` the fractional part is truncated, so this statement stores 3 in `val`. The value of `number`, `3.7`, is not changed.

Type cast expressions are useful in situations where C++ will not perform the desired conversion automatically. Program 3-7 shows an example where a type cast expression is used to prevent integer division from taking place. The statement that uses the type cast expression is

```
booksPerMonth = static_cast<double>(books) / months;
```

Program 3-7

```
1  // This program uses a type cast to avoid an integer division.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int    books,
8            months;
9     double booksPerMonth;
10
11    // Get user inputs
12    cout << "How many books do you plan to read? ";
13    cin  >> books;
14    cout << "How many months will it take you to read them? ";
15    cin  >> months;
16
17    // Compute and display books read per month
18    booksPerMonth = static_cast<double>(books) / months;
19    cout << "That is " << booksPerMonth << " books per month.\n";
20    return 0;
21 }
```

Program Output with Example Input Shown in Bold

```
How many books do you plan to read? 30[Enter]
How many months will it take you to read them? 7[Enter]
That is 4.28571 books per month.
```

The variable `books` is an integer, but a copy of its value is converted to a `double` before it is used in the division operation. Without the type cast expression in line 18, integer division would have been performed, resulting in an incorrect answer.



WARNING! In Program 3-7, the following statement would still have resulted in integer division:

```
booksPerMonth = static_cast<double>(books / months);
```

The result of the expression `books / months` is 4. When 4 is converted to a `double`, it is 4.0. To prevent the integer division from taking place, one of the operands should be converted to a `double` prior to the division operation. This forces C++ to automatically convert the value of the other operand to a `double`.

Program 3-8 shows another use of a type cast.

Program 3-8

```
1  // This program prints a character from its ASCII code.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int number = 65;
8
9     // Display the value of the number variable
10    cout << number << endl;
11
12    // Use a type cast to display the value of number
13    // converted to the char data type
14    cout << static_cast<char>(number) << endl;
15    return 0;
16 }
```

Program Output

```
65
A
```

Let's take a closer look at this program. In line 7 the `int` variable `number` is initialized with the value 65. In line 10, `number` is sent to `cout`, causing 65 to be displayed. In line 14, a type cast expression is used to convert the value in `number` to the `char` data type before sending it to `cout`. Recall from Chapter 2 that characters are stored in memory as integer ASCII codes. Because the number 65 is the ASCII code for the letter 'A', the statement on line 14 causes the letter 'A' to be displayed.



NOTE: C++ provides several different type cast expressions. A `static_cast` is the most commonly used type cast expression, so it is the one we will primarily use in this book. Additional information on type casts is contained in Appendix I, which can be found on the CD accompanying your textbook.

C-style and Prestandard C++ Type Cast Expressions

C++ also supports two older methods of creating type cast expressions: the C-style form and the prestandard C++ form. The C-style cast places the data type to be converted to, enclosed in parentheses, in front of the operand whose value is to be converted. Here are three examples.

```
cout << (int) 2.6;          // Displays integer 2

intVal = (int)number;       // Assigns intVal the value of
                            // number, converted to an int

booksPerMonth =             // Converts a copy of the value
  (double)books / months;   // stored in books to a double
                            // before performing the division
                            // operation
```

Because the typecast operator appears in parentheses preceding the operand, this form of type cast notation is called *prefix notation*.

The prestandard C++ form of the type cast expression also places the data type to be converted to before the operand whose value is to be converted, but it places the parentheses around the operand, rather than around the data type. Here are the same three examples as they would be written using the prestandard C++ form of type casting.

```
cout << int(2.6);
intVal = int(number);
booksPerMonth = double(books) / months;
```

This type cast notation is called *functional notation*.

The `static_cast` expression is recommended by the ANSI standard for this type of data type conversion and is now considered preferable to either the C-style or the prestandard C++ form of type casting. However, you will probably see code in the workplace that uses these older styles. Program 3-9 illustrates how Program 3-7 would be written using a prestandard C++ type cast.

Program 3-9

```
1  // This program illustrates the prestandard C++ form of type casting.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int    books,
8            months;
9     double booksPerMonth;
10
11    // Get user inputs
12    cout << "How many books do you plan to read? ";
13    cin  >> books;
14    cout << "How many months will it take you to read them? ";
15    cin  >> months;
16
```

(program continues)

Program 3-9 *(continued)*

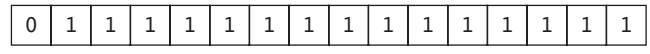
```
17    // Compute and display books read per month
18    booksPerMonth = double(books) / months;
19    cout << "That is " << booksPerMonth << " books per month.\n";
20    return 0;
21 }
```

The output is identical to that produced by Program 3-7.

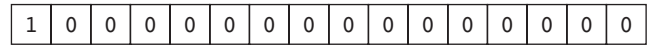
3.5 Overflow and Underflow

CONCEPT: When a value cannot fit in the number of bits provided by a variable's data type, overflow or underflow occurs.

Just as a bucket will overflow if you try to put more water in it than it can hold, a variable will experience a similar problem if you try to store a value in it that requires more bits than it has available. Let's look at an example. Suppose a `short int` that uses 2 bytes of memory has the following value stored in it.



This is the binary representation of 32,767, the largest value that will fit in this data type. Without going into the details of how negative numbers are stored, it is helpful to understand that for integer data types that store both positive and negative numbers, a number with a 0 in the *high order* (i.e., leftmost) *bit* is interpreted as a positive number and a number with a 1 in the high order bit is interpreted as a negative number. If 1 is added to the value stored above, the variable will now be holding the following bit pattern.



But this is not 32,768. It is interpreted as a negative number instead, which was not what was intended. A binary 1 has "flowed" into the high bit position. This is called *overflow*.

Likewise, when an integer variable is holding the value at the far end of its data type's negative range and 1 is subtracted from it, the 1 in its high order bit will become a 0 and the resulting number will be interpreted as a positive number. This is another example of overflow.

In addition to overflow, floating-point values can also experience *underflow*. This occurs when a value is too close to zero, so small that more digits of precision are needed to express it than can be stored in the variable holding it. Program 3-10 illustrates both overflow and underflow.

Program 3-10

```
1  // This program demonstrates overflow and underflow.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      // Set intVar to the maximum value a short int can hold
8      short intVar = 32767;
9
10     // Set floatVar to a number too small to fit in a float
11     float floatVar = 3.0E-47;
12
13     // Display intVar
14     cout << "Original value of intVar     " << intVar << endl;
15
16     // Add 1 to intVar to make it overflow
17     intVar = intVar + 1;
18     cout << "intVar after overflow        " << intVar << endl;
19
20     // Subtract 1 from intVar to make it overflow again
21     intVar = intVar - 1;
22     cout << "intVar after 2nd overflow    " << intVar << endl;
23
24     // Display floatVar
25     cout << "Value of very tiny floatVar  " << floatVar;
26     return 0}
27 }
```

Program Output

```
Original value of intVar    32767
intVar after overflow      -32768
intVar after 2nd overflow   32767
Value of very tiny floatVar 0
```

Although some systems display an error message when an overflow or underflow occurs, most do not. The variable simply holds an incorrect value now and the program keeps running. Therefore, it is important to select a data type for each variable that has enough bits to hold the values you will store in it.



Checkpoint

3.14 Assume the following variable definitions:

```
int a = 5, b = 12;
double x = 3.4, z = 9.1;
```

What are the values of the following expressions?

A) `b / a`

B) `x * a`

C) `static_cast<double>(b / a)`

D) `static_cast<double>(b) / a`

E) `b / static_cast<double>(a)`

F) `static_cast<double>(b) / static_cast<double>(a)`

G) `b / static_cast<int>(x)`

H) `static_cast<int>(x) * static_cast<int>(z)`

I) `static_cast<int>(x * z)`

J) `static_cast<double>(static_cast<int>(x) * static_cast<int>(z))`

3.15 What will the following program display if a capital B is entered when the `cin` statement asks the user to input a letter?

```
#include <iostream>
using namespace std;

int main()
{
   char letter;

   cout << "The ASCII values of uppercase letters are "
        << static_cast<int>('A') << " - "
        << static_cast<int>('Z') << endl;

   cout << "The ASCII values of lowercase letters are "
        << static_cast<int>('a') << " - "
        << static_cast<int>('z') << endl << endl;

   cout << "Enter a letter and I will tell you its ASCII code: ";
   cin  >> letter;
   cout << "The ASCII code for " << letter << " is "
        << static_cast<int>(letter) << endl;

   return 0;
}
```

3.16 What will the following program display?

```
#include <iostream>
using namespace std;

int main()
{
   int   integer1 = 19,
         integer2 = 2;
   double doubleVal;

   doubleVal = integer1 / integer2;
   cout << doubleVal << endl;
   doubleVal = static_cast<double>(integer1) / integer2;
   cout << doubleVal << endl;
   doubleVal = static_cast<double>(integer1 / integer2);
   cout << doubleVal << endl;
   return 0;
}
```

3.6 Named Constants

CONCEPT: Constants may be given names that symbolically represent them in a program.

In Chapter 2 you learned about numbers and strings being expressed as constants. For example, the following statement contains the numeric constant 0.129:

```
newAmount = balance * 0.129;
```

Let's assume this statement appears in a banking program that calculates data pertaining to loans. In such a program, two potential problems arise. First, it is not clear to anyone other than the original programmer what 0.129 is. It appears to be an interest rate, but in some situations there are fees associated with loan payments. How can the purpose of this statement be determined without painstakingly checking the rest of the program?

The second problem occurs if this number is used in other calculations throughout the program and must be changed periodically. Assuming the number is an interest rate, what if the rate changes from 12.9 percent to 13.2 percent? The programmer will have to search through the source code for every occurrence of the number.

Both of these problems can be addressed by using *named constants*. A named constant, also called a *constant variable*, is like a variable, but its content is read-only and cannot be changed while the program is running. Here is a definition of a named constant:

```
const double INTEREST_RATE = 0.129;
```

It looks just like a regular variable definition except that the word `const` appears before the data type name. The key word `const` is a qualifier that tells the compiler to make the variable read-only. This ensures that its value will remain constant throughout the program's execution. If any statement in the program attempts to change its value, an error results when the program is compiled. A named constant can have any legal C++ identifier name, but many programmers use all uppercase letters in the name, as we have done here, to distinguish it from a regular variable.

When a named constant is defined it must be initialized with a value. It cannot be defined and then later assigned a value with an assignment statement.

```
const double INTEREST_RATE;      // illegal
INTEREST_RATE = 0.129;           // illegal
```

An added advantage of using named constants is that they make programs more self-documenting. If the named constant `INTEREST_RATE` has been correctly defined, the program statement

```
newAmount = balance * 0.129;
```

can be changed to read

```
newAmount = balance * INTEREST_RATE;
```

A new programmer can read the second statement and better understand what is happening. It is evident that `balance` is being multiplied by the interest rate. Another advantage to this approach is that widespread changes can easily be made to the program. No matter how many places the interest rate is used in the program, if the rate changes the programmer only has to change one line of code—the statement that defines and initializes the named constant. The following line of code, for example, would be used to set a new interest rate of 13.2 percent.

```
const double INTEREST_RATE = 0.132;
```

All that has to be done then is to recompile the program and every statement that uses `INTEREST_RATE` will use the new value.

It is also useful to define named constants for common values that are difficult to remember. For example, Program 3-6 calculated the area of a circle. The number 3.14159 is used for pi in the formula. This value could easily be defined as a named constant, as shown in line 9 of Program 3-11.

Program 3-11

```
1  // This program calculates the area of a circle. The formula for the
2  // area of a circle is PI times the radius squared. PI is 3.14159.
3  #include <iostream>
4  #include <cmath>                  // Needed for the pow function
5  using namespace std;
6
7  int main()
8  {
9     const double PI = 3.14159;     // PI is a named constant
10    double area, radius;
11
12    cout << "This program calculates the area of a circle.\n";
13
14    // Get the radius
15    cout << "What is the radius of the circle? ";
16    cin  >> radius;
17
18    // Compute and display the area
19    area = PI * pow(radius, 2);
20    cout << "The area is " << area << endl;
21    return 0;
22 }
```

The `#define` Directive

The older C-style method of creating named constants is with the `#define` preprocessor directive. Although it is preferable to use the `const` modifier, there are programs with the `#define` directive still in use. In addition, the `#define` directive has other uses, so it is important to understand. Program 3-12 shows how the preprocessor can be used to create a named constant.

Program 3-12

```
1  // This program calculates the area of a circle. The formula for the
2  // area of a circle is PI times the radius squared. PI is 3.14159.
3  #include <iostream>
4  #include <cmath>                 // Needed for the pow function
5  using namespace std;
6
7  #define PI 3.14159               // PI is "defined" to be 3.14159
8
9  int main()
10 {
11    double area, radius;
12
13    cout << "This program calculates the area of a circle.\n";
14
15    // Get the radius
16    cout << "What is the radius of the circle? ";
17    cin  >> radius;
18
19    // Compute and display the area
20    area = PI * pow(radius, 2);
21    cout << "The area is " << area << endl;
22    return 0;
23 }
```

Remember, the preprocessor scans your program before it is compiled. It looks for directives, which are lines that begin with the # symbol. Preprocessor directives cause your source code to be modified prior to being compiled. Line 7 of Program 3-12 uses the following `#define` directive:

```
#define PI 3.14159
```

The word `PI` is a named constant and 3.14159 is its value. Anytime `PI` is used in the program, it will be replaced by the value 3.14159. The code on line 20 that reads

```
area = PI * pow(radius, 2);
```

will be sent to the compiler as

```
area = 3.14159 * pow(radius, 2);
```

If there had been a line that read

```
cout << PI << endl;
```

it would be compiled as

```
cout << 3.14159 << endl;
```

It is important to realize the difference between constant variables created with the key word `const` and constants created with the `#define` directive. Constant variables are defined like regular variables. They have a data type and a specific storage location in memory. They are like regular variables in every way except that you cannot change their

value while the program is running. Constants created with the `#define` directive, however, are not variables at all, but text substitutions. Each occurrence of the named constant in your source code is removed and the value of the constant is written in its place when it is sent to the compiler.

Be careful not to put a semicolon at the end of a `#define` directive. If you used a semicolon it would actually become part of the value of the constant. If the `#define` directive in line 7 of Program 3-12 had read like this,

```
#define PI 3.14159;
```

the mathematical statement

```
area = PI * pow(radius, 2);
```

would have been modified to read

```
area = 3.14159; * pow(radius, 2);
```

Because of the semicolon, the preprocessor would have created a syntax error in the statement and the compiler would have given an error message when trying to process this statement.



NOTE: `#define` directives are intended for the preprocessor and C++ statements are intended for the compiler. The preprocessor does not look for semicolons to terminate directives.



Checkpoint

3.17 Write statements using the `const` qualifier to create named constants for the following literal values:

Constant Value	Description
`2.71828`	Euler's number (known in mathematics as *e*)
`5.256E5`	Number of minutes in a year
`32.2`	The gravitational acceleration constant (in feet per second^2)
`9.8`	The gravitational acceleration constant (in meters per second^2)
`1609`	Number of meters in a mile

3.18 Write `#define` directives for the literal values listed in question 3.17.

3.19 Assuming the user enters 6 in response to the question, what will the following program display on the screen?

```
#include <iostream>
using namespace std;

#define GREETING1 "This program calculates the number "
#define GREETING2 "of candy pieces sold."
#define QUESTION "How many jars of candy have you sold? "
#define RESULTS "The number of pieces sold: "
#define YOUR_COMMISSION "Candy pieces you get for commission: "
#define COMMISSION_RATE .20
```

```
int main()
{
   const int PIECES_PER_JAR = 1860;
   int jars, pieces;
   double commission;

   cout << GREETING1;
   cout << GREETING2 << endl;
   cout << QUESTION;
   cin  >> jars;

   pieces = jars * PIECES_PER_JAR;
   cout << RESULTS << pieces << endl;
   commission = pieces * COMMISSION_RATE;
   cout << YOUR_COMMISSION << commission << endl;
   return 0;
}
```

3.20 Complete the following program skeleton so it properly converts a speed entered in miles per hour to feet per second. One mile per hour is 1.467 feet per second.

```
#include <iostream>
using namespace std;

int main()
{
   // Define a named constant called CONVERSION
   // with the value 1.467.
   double milesPerHour, feetPerSecond;

   cout << "This program converts miles per hour to\n";
   cout << "feet per second.\n";
   cout << "Enter a speed in MPH: ";
   cin  >> milesPerHour;
   // Insert a mathematical statement here to
   // calculate feet per second and assign the result
   // to the feetPerSecond variable.
   cout << "That is " << feetPerSecond << " feet per second.\n";
   return 0;
}
```

3.7 Multiple and Combined Assignment

CONCEPT: Multiple assignment means to assign the same value to several variables with one statement.

C++ allows you to assign a value to multiple variables at once. If a program has several variables, such as `a`, `b`, `c`, and `d`, and each variable needs to be assigned a value, such as 12, the following statement may be constructed:

```
a = b = c = d = 12;
```

The value 12 will be assigned to each variable listed in the statement. This works because the assignment operations are carried out from right to left. First 12 is assigned to `d`. Then

d's value, now a 12, is assigned to c. Then c's value is assigned to b, and finally b's value is assigned to a.

Program 3-13 uses the following multiple assignment statement to store a value entered by the user into two different variables:

```
store1 = store2 = begInv;
```

Combined Assignment Operators



VideoNote
Combined Assignment Operators

Quite often programs have assignment statements of the following form:

```
number = number + 1;
```

The expression on the right side of the assignment operator gives the value of number plus 1. The result is then assigned to number, replacing the value that was previously stored there. Effectively, this statement adds 1 to number. In a similar fashion, the following statement subtracts 5 from number.

```
number = number – 5;
```

If you have never seen this type of statement before, it might cause some initial confusion because the same variable name appears on both sides of the assignment operator. Table 3-8 shows other examples of statements written this way.

Table 3-8 Assignment Statements that Change a Variable's Value (Assume x = 6)

Statement	What It Does	Value of x After the Statement
`x = x + 4;`	Adds 4 to x	10
`x = x – 3;`	Subtracts 3 from x	3
`x = x * 10;`	Multiplies x by 10	60
`x = x / 2;`	Divides x by 2	3
`x = x % 4`	Makes x the remainder of x / 4	2

Because these types of operations are so common in programming, C++ offers a special set of operators designed specifically for these jobs. Table 3-9 shows the *combined assignment operators*, also known as *compound operators* or *arithmetic assignment operators*.

Table 3-9 Combined Assignment Operators

Operator	Example Usage	Equivalent To
`+=`	`x += 5;`	`x = x + 5;`
`-=`	`y -= 2;`	`y = y - 2;`
`*=`	`z *= 10;`	`z = z * 10;`
`/=`	`a /= b;`	`a = a / b;`
`%=`	`c %= 3;`	`c = c % 3;`

As you can see, the combined assignment operators do not require the programmer to type the variable name twice. Also, they give a clear indication of what is happening in the statement. Program 3-13 uses combined assignment operators.

Program 3-13

```
1  // This program tracks the inventory of two widget stores.
2  // It illustrates the use of multiple and combined assignment.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int begInv,    // Beginning inventory for both stores
9         sold,      // Number of widgets sold
10        store1,    // Store 1's inventory
11        store2;    // Store 2's inventory
12
13    // Get the beginning inventory for the two stores
14    cout << "One week ago, 2 new widget stores opened\n";
15    cout << "at the same time with the same beginning\n";
16    cout << "inventory. What was the beginning inventory? ";
17    cin  >> begInv;
18
19    // Set each store's inventory
20    store1 = store2 = begInv;
21
22    // Get the number of widgets sold at each store
23    cout << "How many widgets has store 1 sold? ";
24    cin  >> sold;
25    store1 -= sold;     // Adjust store 1's inventory
26
27    cout << "How many widgets has store 2 sold? ";
28    cin  >> sold;
29    store2 -= sold;     // Adjust store 2's inventory
30
31    // Display each store's current inventory
32    cout << "\nThe current inventory of each store:\n";
33    cout << "Store 1: " << store1 << endl;
34    cout << "Store 2: " << store2 << endl;
35    return 0;
36 }
```

Program Output with Example Input Shown in Bold

```
One week ago, 2 new widget stores opened
at the same time with the same beginning
inventory. What was the beginning inventory? 100[Enter]
How many widgets has store 1 sold? 25[Enter]
How many widgets has store 2 sold? 15[Enter]

The current inventory of each store:
Store 1: 75
Store 2: 85
```

More elaborate statements may be expressed with the combined assignment operators. Here is an example:

```
result *= a + 5;
```

In this statement, `result` is multiplied by the sum of `a + 5`. Notice that the precedence of the combined assignment operators is lower than that of the regular arithmetic operators. The above statement is equivalent to

```
result = result * (a + 5);
```

which is different from

```
result = result * a + 5;
```

Table 3-10 shows additional examples using combined assignment operators.

Table 3-10 Examples Using Combined Assignment Operators and Arithmetic Operators

Example Usage	Equivalent To
`x += b + 5;`	`x = x + (b + 5);`
`y -= a * 2;`	`y = y - (a * 2);`
`z *= 10 - c;`	`z = z * (10 - c);`
`a /= b + c;`	`a = a / (b + c);`
`c %= d - 3;`	`c = c % (d - 3);`



Checkpoint

3.21 Write a multiple assignment statement that assigns 0 to the variables `total`, `subtotal`, `tax`, and `shipping`.

3.22 Write statements using combined assignment operators to perform the following:

A) Add 6 to `x`.
B) Subtract 4 from `amount`.
C) Multiply `y` by 4.
D) Divide `total` by 27.
E) Store in `x` the remainder of `x` divided by 7.
F) Add `y * 5` to `x`.
G) Subtract `discount` times 4 from `total`.
H) Multiply `increase` by `salesRep` times 5.
I) Divide `profit` by `shares` minus 1000.

3.23 What will the following program display?

```
#include <iostream>
using namespace std;

int main()
{
   int unus, duo, tres;

   unus = duo = tres = 5;
   unus += 4;
```

```
    duo *= 2;
    tres -= 4;
    unus /= 3;
    duo += tres;
    cout << unus << endl;
    cout << duo << endl;
    cout << tres << endl;
    return 0;
}
```

3.8 Formatting Output

CONCEPT: **cout provides ways to format data as it is being displayed. This affects the way data appears on the screen.**

The same data can be printed or displayed in several different ways. For example, all of the following numbers have the same value, although they look different:

```
720
720.0
720.00000000
7.2e+2
+720.0
```

The way a value is printed is called its *formatting*. The `cout` object has a standard way of formatting variables of each data type. Sometimes, however, you need more control over the way data is displayed. Consider Program 3-14, for example, which displays three rows of numbers with spaces between each one.

Program 3-14

```
1  // This program displays three rows of numbers.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int num1 = 2897, num2 = 5,    num3 = 837,
8         num4 = 34,   num5 = 7,    num6 = 1623,
9         num7 = 390,  num8 = 3456, num9 = 12;
10
11    // Display the first row of numbers
12    cout << num1 << "  " << num2 << "  " << num3 << endl;
13
14    // Display the second row of numbers
15    cout << num4 << "  " << num5 << "  " << num6 << endl;
16
17    // Display the third row of numbers
18    cout << num7 << "  " << num8 << "  " << num9 << endl;
19
20    return 0;
21 }
```

(program continues)

Program 3-14 *(continued)*

Program Output

```
2897  5  837
34  7  1623
390  3456  12
```

Unfortunately, the numbers do not line up in columns. This is because some of the numbers, such as 5 and 7, occupy one position on the screen, while others occupy two or three positions. `cout` uses just the number of spaces needed to print each number.

To remedy this, `cout` offers a way of specifying the minimum number of spaces to use for each number. A stream manipulator, `setw`, can be used to establish print fields of a specified width. Here is an example of how it is used:

```
value = 23;
cout << setw(5) << value;
```

The number inside the parentheses after the word `setw` specifies the *field width* for the value immediately following it. The field width is the minimum number of character positions, or spaces, on the screen to print the value in. In our example, the number 23 will be displayed in a field of five spaces.

To further clarify how this works, look at the following statements:

```
value = 23;
cout << "(" << setw(5) << value << ")";
```

This will produce the following output:

```
(   23)
```

Because the number did not use the entire field, `cout` filled the extra three positions with blank spaces. Because the number appears on the right side of the field with blank spaces "padding" it in front, it is said to be *right-justified*.

Program 3-15 shows how the numbers in Program 3-14 can be printed in columns that line up perfectly by using `setw`. In addition, because we used a `setw(6)`, and the largest number has four digits, the numbers will be separated without having to print a string constant containing blanks between the numbers.

Program 3-15

```
1  // This program uses setw to display three rows of numbers so they align.
2  #include <iostream>
3  #include <iomanip>         // Header file needed to use setw
4  using namespace std;
5
6  int main()
7  {
8     int num1 = 2897, num2 = 5,    num3 = 837,
9         num4 = 34,   num5 = 7,    num6 = 1623,
10        num7 = 390,  num8 = 3456, num9 = 12;
```

(program continues)

Program 3-15 *(continued)*

```
11
12    // Display the first row of numbers
13    cout << setw(6) << num1 << setw(6) << num2 << setw(6) << num3 << endl;
14
15    // Display the second row of numbers
16    cout << setw(6) << num4 << setw(6) << num5 << setw(6) << num6 << endl;
17
18    // Display the third row of numbers
19    cout << setw(6) << num7 << setw(6) << num8 << setw(6) << num9 << endl;
20
21    return 0;
22 }
```

Program Output

```
  2897     5   837
    34     7  1623
   390  3456    12
```



NOTE: A new header file, `iomanip`, is included on line 3 of Program 3-15. It must be used in any program that uses `setw`.

Notice that a `setw` manipulator is used with each value. This is because `setw` only establishes a field width for the value immediately following it. After that value is printed, `cout` goes back to its default method of printing.

You might wonder what will happen if the number is too large to fit in the field, as in the following statement:

```
value = 18397;
cout << setw(2) << value;
```

In cases like this, `cout` will print the entire number. `setw` only specifies the minimum number of positions in the print field. Any number larger than the minimum will cause `cout` to override the `setw` value.

You may specify the field width of any type of data. Program 3-16 shows `setw` being used with an integer, a floating-point number, and a string.

Program 3-16

```
1 // This program demonstrates the setw manipulator
2 // being used with variables of various data types.
3 #include <iostream>
4 #include <iomanip>
5 #include <string>
6 using namespace std;
7
```

(program continues)

Program 3-16 *(continued)*

```
8  int main()
9  {
10    int intValue = 3928;
11    double doubleValue = 91.5;
12    string stringObjectValue = "Jill Q. Jones";
13
14    cout << "(" << setw(5) << intValue << ")" << endl;
15    cout << "(" << setw(8) << doubleValue << ")" << endl;
16    cout << "(" << setw(16) << stringObjectValue << ")" << endl;
17    return 0;
18 }
```

Program Output

```
( 3928)
(    91.5)
(   Jill Q. Jones)
```

Program 3-16 illustrates the following points:

- The field width of a floating-point number includes a position for the decimal point.
- The field width of a string includes all characters in the string, including spaces.
- The values printed in the field are right-justified by default. This means they are aligned with the right side of the print field, and any blanks that must be used to pad it are inserted in front of the value.

The `setprecision` Manipulator

Floating-point values may be rounded to a number of *significant digits*, or *precision*, which is the total number of digits that appear before and after the decimal point. You can control the number of significant digits with which floating-point values are displayed by using the `setprecision` manipulator. Program 3-17 shows the results of a division operation displayed with different numbers of significant digits.

Program 3-17

```
1  // This program demonstrates how the setprecision manipulator
2  // affects the way a floating-point value is displayed.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     double, number1 = 132.364, number2 = 26.91;
10    double quotient = number1 / number2;
11
12    cout << quotient << endl;
13    cout << setprecision(5) << quotient << endl;
```

(program continues)

Program 3-17 *(continued)*

```
14    cout << setprecision(4) << quotient << endl;
15    cout << setprecision(3) << quotient << endl;
16    cout << setprecision(2) << quotient << endl;
17    cout << setprecision(1) << quotient << endl;
18    return 0;
19 }
```

Program Output

```
4.91877
4.9188
4.919
4.92
4.9
5
```



NOTE: With prestandard compilers, your output may be different from that shown in Program 3-17.

The first value in Program 3-17 is displayed in line 12 without the `setprecision` manipulator. (By default, the system displays floating-point values with six significant digits.) The subsequent `cout` statements print the same value, but rounded to five, four, three, two, and one significant digits. Notice that, unlike `setw`, `setprecision` does not count the decimal point. When we used `setprecision(5)`, for example, the output contained five significant digits, which required six positions to print 4.9188.

If the value of a number is expressed in fewer digits of precision than specified by `setprecision`, the manipulator will have no effect. In the following statements, the value of `dollars` only has four digits of precision, so the number printed by both `cout` statements is 24.51.

```
double dollars = 24.51;
cout << dollars << endl;                          // displays 24.51
cout << setprecision(5) << dollars << endl;       // displays 24.51
```

Table 3-11 shows how `setprecision` affects the way various values are displayed. Notice that when fewer digits are to be displayed than the number holds, `setprecision` rounds, rather than truncates, the number. Notice also that trailing zeros are omitted. Therefore, for example, 21.40 displays as 21.4 even though `setprecision(5)` is specified.

Table 3-11 The `setprecision` Manipulator

Number	Manipulator	Value Displayed
28.92786	setprecision(3)	28.9
21.40	setprecision(5)	21.4
109.50	setprecision(4)	109.5
34.78596	setprecision(2)	35

Unlike field width, the precision setting remains in effect until it is changed to some other value. As with all formatting manipulators, you must include the header file `iomanip` to use `setprecision`.

Program 3-18 shows how the `setw` and `setprecision` manipulators may be combined to control the way floating-point numbers are displayed.

Program 3-18

```
1  // This program asks for sales figures for three days.
2  // The total sales are calculated and displayed in a table.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     double day1, day2, day3, total;
10
11    // Get the sales for each day
12    cout << "Enter the sales for day 1: ";
13    cin  >> day1;
14    cout << "Enter the sales for day 2: ";
15    cin  >> day2;
16    cout << "Enter the sales for day 3: ";
17    cin  >> day3;
18
19    // Calculate total sales
20    total = day1 + day2 + day3;
21
22    // Display the sales figures
23    cout << "\nSales Figures\n";
24    cout << "-------------\n";
25    cout << setprecision(5);
26    cout << "Day 1: " << setw(8) << day1 << endl;
27    cout << "Day 2: " << setw(8) << day2 << endl;
28    cout << "Day 3: " << setw(8) << day3 << endl;
29    cout << "Total: " << setw(8) << total << endl;
30    return 0;
31 }
```

Program Output with Example Input Shown in Bold

```
Enter the sales for day 1: 321.57[Enter]
Enter the sales for day 2: 269.60[Enter]
Enter the sales for day 3: 307.00[Enter]

Sales Figures
-------------
Day 1:   321.57
Day 2:    269.6
Day 3:      307
Total:   898.17
```

The output created by Program 3-18, as we directed, allows a maximum of five significant digits to be displayed and is printed right justified in a field width of eight characters. However, the result is clearly not what is desired. In just a moment, we'll look at another manipulator that provides additional control over the format of the output.

The `fixed` Manipulator

If a number is too large to print using the number of digits specified with `setprecision`, many systems print it in scientific notation. For example, here is the output of Program 3-18 with larger numbers being input.

```
Enter the sales for day 1: 145678.99[Enter]
Enter the sales for day 2: 205614.85[Enter]
Enter the sales for day 3: 198645.22[Enter]

Sales Figures
-------------
Day 1: 1.4568e+005
Day 2: 2.0561e+005
Day 3: 1.9865e+005
Total: 5.4994e+005
```

To prevent this, you can use another stream manipulator, `fixed`, which indicates that floating-point output should be printed in *fixed-point*, or decimal, *notation*.

```
cout << fixed;
```

What is perhaps most important about the `fixed` manipulator, however, is that when the `setprecision` manipulator is used in conjunction with `fixed`, it behaves in a new way. It specifies the number of digits to be displayed after the decimal point of a floating-point number, rather than the total number of digits to be displayed. This is usually what we want. For example, if we rewrite line 25 of Program 3-18 as

```
cout << fixed << setprecision(2);
```

and rerun the program using the same sample data, we get the following results:

```
Enter the sales for day 1: 321.57[Enter]
Enter the sales for day 2: 269.60[Enter]
Enter the sales for day 3: 307.00[Enter]

Sales Figures
-------------
Day 1:   321.57
Day 2:   269.60
Day 3:   307.00
Total:   898.17
```

By using `fixed` and `setprecision` together, we get the desired output. Notice in this case, however, we set the precision to 2, the number of decimal places we wish to see, not to 5.

The `showpoint` Manipulator

Another useful manipulator is `showpoint`, which indicates that a decimal point should be printed for a floating-point number, even if the value being displayed has no decimal

digits. Program 3-19 illustrates the use of `fixed`, `showpoint`, and `setprecision`. As with `setprecision`, the `fixed` and `showpoint` manipulators remain in effect until the programmer explicitly changes them.

Program 3-19

```
1  // This program illustrates how the fixed, showpoint, and
2  // setprecision manipulators operate when used together.
3  #include <iostream>
4  #include <iomanip>                // Needed to use stream manipulators
5  using namespace std;
6
7  int main()
8  {
9     double amount = 125.0;
10
11    cout << setw(6) << amount << endl;
12
13    cout << showpoint;
14    cout << setw(6) << amount << endl;
15
16    cout << fixed << showpoint << setprecision(2);
17    cout << setw(6) << amount << endl;
18    return 0;
19 }
```

Program Output

```
   125
125.000
125.00
```

When `amount` is printed the first time, in line 11, the `showpoint` and `setprecision` manipulators have not yet been set. Therefore, since the value being displayed requires no decimal digits, only 125 is displayed. When `amount` is printed the second time, in line 14, the `showpoint` manipulator has been set, so a decimal point followed by zeroes is displayed. However, since the `setprecision` manipulator has not yet been set, we have no control over how many zeroes are to be printed, and 125.000 is displayed. Finally, when `amount` is printed the third time, in line 17, the `fixed` and `setprecision` manipulators have both been set, specifying that exactly two decimal digits are to be printed, so 125.00 is displayed.

Actually, when the `fixed` and `setprecision` manipulators are both used, it is not necessary to use the `showpoint` manipulator. For example,

```
cout << fixed << setprecision(2);
```

will automatically display a decimal point before the two decimal digits. However, many programmers prefer to use it anyway as shown here:

```
cout << fixed << showpoint << setprecision(2);
```

The `left` and `right` Manipulators

Normally, as you have seen, output is right-justified. This means if the field it prints in is larger than the value being displayed, it is printed on the far right of the field, with leading

blanks. There are times when you may wish to force a value to print on the left side of its field, padded by blanks on the right. To do this you can use the `left` manipulator. It remains in effect until you use a `right` manipulator to set it back. These manipulators can be used with any type of value, even a string. Program 3-20 illustrates the `left` and `right` manipulators. It also illustrates that the `fixed`, `showpoint`, and `setprecision` manipulators have no effect on integers, only on floating-point numbers.

Program 3-20

```
1  // This program illustrates the use of the left and right manipulators.
2  #include <iostream>
3  #include <iomanip>        // Needed to use stream manipulators
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string month1 = "January",
10           month2 = "February",
11           month3 = "March";
12
13    int days1 = 31,
14        days2 = 28,
15        days3 = 31;
16
17    double high1 = 22.6,
18           high2 = 37.4,
19           high3 = 53.9;
20
21    cout << fixed << showpoint << setprecision(1);
22    cout << "Month        Days    High\n";
23
24    cout << left  << setw(12) << month1
25         << right << setw(4)  << days1 << setw(9) << high1 << endl;
26    cout << left  << setw(12) << month2
27         << right << setw(4)  << days2 << setw(9) << high2 << endl;
28    cout << left  << setw(12) << month3
29         << right << setw(4)  << days3 << setw(9) << high3 << endl;
30
31    return 0;
32 }
```

Program Output

```
Month         Days    High
January         31     22.6
February        28     37.4
March           31     53.9
```

Chapter 13 introduces additional stream manipulators and output formatting methods. However, the six manipulators we have covered in this chapter are normally sufficient to produce the output you desire. Table 3-12 summarizes these six manipulators.

Table 3-12 Output Stream Manipulators

Stream Manipulator	Description
`setw(n)`	Sets the display field width to size `n`.
`fixed`	Displays floating-point numbers in fixed (i.e., decimal) form.
`showpoint`	Displays the decimal point and trailing zeroes for floating-point numbers even if there is no fractional part.
`setprecision(n)`	If the `fixed` manipulator is *not* set, `n` indicates the total number of significant digits to display. If the `fixed` manipulator is set, `n` indicates the number of decimal digits to display.
`left`	Causes subsequent output to be left-justified.
`right`	Causes subsequent output to be right-justified.



Checkpoint

3.24 Write `cout` statements with stream manipulators that perform the following:

A) Display the number 34.789 in a field of nine spaces with two decimal places of precision.

B) Display the number 7.0 in a field of five spaces with three decimal places of precision. The decimal point and any trailing zeroes should be displayed.

C) Display the number 5.789e+12 in fixed-point notation.

D) Display the number 67 left-justified in a field of seven spaces.

3.25 The following program skeleton asks for an angle in degrees and converts it to radians. The formatting of the final output is left to you.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const double PI = 3.14159;
   double degrees, radians;

   cout << "Enter an angle in degrees and I will convert it\n";
   cout << "to radians for you: ";
   cin  >> degrees;
   radians = degrees * PI / 180;

   // Display the value in radians left-justified, in fixed-point
   // notation, with four decimal places of precision, in a field
   // seven spaces wide.
   return 0;
}
```

3.9 Working with Characters and String Objects

CONCEPT: Special functions exist for working with characters and string objects.

In Chapter 2 you were introduced to characters and to string objects. A `char` variable can hold only one character, whereas a variable defined as a string can hold a whole set of characters. The following variable definitions and initializations illustrate this.

```
char letter1 = 'A',
     letter2 = 'B';
string name1 = "Mark Twain",
       name2 = "Samuel Clemens";
```

As with numeric data types, characters and strings can be assigned values.

```
letter2 = letter1;        // Now letter2's value is 'A'
name2 = name1;            // Now name2's value is "Mark Twain"
```

They can also be displayed with the `cout` statement. The following line of code outputs a character variable, a string constant, and a string object.

```
cout << letter1 << ". " << name1 << endl;
```

The output produced is

```
A. Mark Twain
```

Inputting characters and strings, however, is a little trickier than reading in numeric values.

Inputting a String

Although it is possible to use `cin` with the >> operator to input strings, it can cause problems you need to be aware of. When `cin` reads data it passes over and ignores any leading *whitespace* characters (spaces, tabs, or line breaks). However, once it comes to the first nonblank character and starts reading, it stops reading when it gets to the next whitespace character. If we use the following statement

```
cin >> name1;
```

we can input "Mark", or even " Mark", but not "Mark Twain" because `cin` cannot input strings that contain embedded spaces.

Program 3-21 illustrates this problem.

Program 3-21

```
1 // This program illustrates a problem that can occur if
2 // cin is used to read character data into a string object.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
```

(program continues)

Program 3-21 *(continued)*

```
 7 int main()
 8 {
 9    string name;
10     string city;
11
12     cout << "Please enter your name: ";
13     cin  >> name;
14     cout << "Enter the city you live in: ";
15     cin  >> city;
16
17     cout << "Hello, " << name << endl;
18     cout << "You live in " << city << endl;
19     return 0;
20 }
```

Program Output with Example Input Shown in Bold

```
Please enter your name. John Doe[Enter]
Enter the city you live in: Hello, John
You live in Doe
```

Notice that the user was never given the opportunity to enter the city. In the first input statement, when `cin` came to the space between `John` and `Doe`, it stopped reading, storing just `John` as the value of `name`. In the second input statement, `cin` used the leftover characters it found in the keyboard buffer and stored `Doe` as the value of `city`.

To solve this problem, C++ provides a special function called `getline`. This function will read in an entire line, including leading and embedded spaces, and store it in a string object. The `getline` function looks like the following, where `cin` is the input stream we are reading from and `inputLine` is the name of the `string` variable receiving the input string.

```
getline(cin, inputLine);
```

Program 3-22 illustrates using the `getline` function.

Program 3-22

```
 1 // This program illustrates using the getline function
 2 // to read character data into a string object.
 3 #include <iostream>
 4 #include <string>
 5 using namespace std;
 6
 7 int main()
 8 {
 9     string name;
10     string city;
11
12     cout << "Please enter your name: ";
13     getline(cin, name);
14     cout << "Enter the city you live in: ";
15     getline(cin, city);
```

(program continues)

Program 3-22 *(continued)*

```
16
17      cout << "Hello, " << name << endl;
18      cout << "You live in " << city << endl;
19      return 0;
20  }
```

Program Output with Example Input Shown in Bold

```
Please enter your name. John Doe[Enter]
Enter the city you live in: Chicago[Enter]
Hello, John Doe
You live in Chicago
```

Inputting a Character

Sometimes you want to read only a single character of input. For example, some programs display a menu of items for the user to choose from. Often the selections will be denoted by the letters A, B, C, and so forth. The user chooses an item from the menu by typing a character. The simplest way to read a single character is with `cin` and the >> operator, as illustrated in Program 3-23.

Program 3-23

```
1  // This program reads a single character into a char variable.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     char ch;
8
9     cout << "Type a character and press Enter: ";
10    cin  >> ch;
11    cout << "You entered " << ch << endl;
12    return 0;
13 }
```

Program Output with Example Input Shown in Bold

```
Type a character and press Enter: A[Enter]
You entered A
```

Using `cin.get`

As with string input, however, there are times when using `cin` >> to read a character does not do what we want. For example, because it passes over all leading whitespace, it is impossible to input just a blank or [Enter] with `cin` >>. The program will not continue past the `cin` statement until some character other than the spacebar, the tab key, or the [Enter] key has been pressed. (Once such a character is entered, the [Enter] key must still be pressed before the program can continue to the next statement.) Thus, programs that ask the user to `"Press the enter key to continue."` cannot use the >> operator to read only the pressing of the [Enter] key.

In those situations, a `cin` function called `get` becomes useful. The `get` function reads a single character, including any whitespace character. If the program needs to store the character being read, the `get` function can be called in either of the following ways. In both cases, `ch` is the name of the variable that the character is being read into.

```
cin.get(ch);
ch = cin.get();
```

If the program is using the `get` function simply to hold the screen until the [Enter] key is pressed and does not need to store the character, the function can also be called like this:

```
cin.get();
```

Program 3-24 illustrates all three ways to use the `get` function.

Program 3-24

```
1  // This program demonstrates three ways
2  // to use cin.get() to pause a program.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8      char ch;
9
10     cout << "This program has paused. Press Enter to continue.";
11     cin.get(ch);
12     cout << "It has paused a second time. Please press Enter again.";
13     ch = cin.get();
14     cout << "It has paused a third time.  Please press Enter again.";
15     cin.get();
16     cout << "Thank you!";
17     return 0;
18 }
```

Program Output with Example Input Shown in Bold

```
This program has paused. Press Enter to continue.[Enter]
It has paused a second time. Please press Enter again.[Enter]
It has paused a third time.  Please press Enter again.[Enter]
Thank you!
```

Mixing `cin >>` and `cin.get`

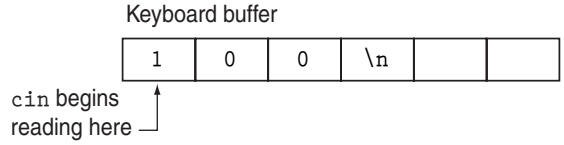
Mixing `cin >>` with `cin.get` can cause an annoying and hard-to-find problem. For example, look at the following code segment. The lines are numbered for reference.

```
1 char ch;                        // Define a character variable
2 int number;                     // Define an integer variable
3 cout << "Enter a number: ";
4 cin  >> number;                 // Read an integer
3 cout << "Enter a character: ";
6 ch = cin.get();                 // Read a character
7 cout << "Thank You!\n";
```

These statements allow the user to enter a number, but not a character. It will appear that the `cin.get` statement on line 6 has been skipped.

This happens because both `cin >>` and `cin.get` read the user's keystrokes from the keyboard buffer. After entering a number in response to the first prompt, the user presses the [Enter] key. Pressing this key causes a newline character (`'\n'`) to be stored in the keyboard buffer. For example, suppose the user enters 100 and presses [Enter]. The input will be stored in the keyboard buffer as shown in Figure 3-3.

Figure 3-3



When the `cin >>` statement reads data from the keyboard buffer, it stops reading at the newline character. In our example, 100 is read in and stored in the `number` variable. The newline character is left in the keyboard buffer. However, `cin.get` always reads the next character in the buffer, no matter what it is, without skipping over whitespace. It only waits for the user to input a value if the keyboard buffer is empty. When `cin.get` finds the newline character in the buffer, it uses it and does not wait for the user to input another value. You can remedy this situation by using the `cin.ignore` function, described in the following section.

Using `cin.ignore`

To solve this problem, the `cin.ignore` function can be used. This function tells the `cin` object to skip characters in the keyboard buffer. Here is its general form:

```
cin.ignore(n, c);
```

The arguments shown in the parentheses are optional. If they are used, `n` is an integer and `c` is a character. They tell `cin` to skip `n` number of characters, or until the character `c` is encountered. For example, the following statement causes `cin` to skip the next 20 characters or until a newline is encountered, whichever comes first:

```
cin.ignore(20,'\n');
```

If no arguments are used, `cin` will only skip the very next character. Here's an example:

```
cin.ignore();
```

The statements that mix `cin >>` and `cin.get` can be repaired by inserting a `cin.ignore` statement after the `cin >>` statement:

```
cout << "Enter a number: ";
cin  >> number;
cin.ignore();           // Skip the newline character
cout << "Enter a character: ";
cin.get(ch);
cout << "Thank You!" << endl;
```

Useful String Functions and Operators

The C++ `string` class provides a number of functions, called member functions, for working with strings. One that is particularly useful is the `length` function, which tells you how many characters there are in a string. Here is an example of how to use it.

```
string state = "New Jersey";
int size = state.length();
```

The `size` variable now holds the value 10. Notice that a blank space is a character and is counted just like any other character. On the other hand, notice that the '\0' null character you learned about in Chapter 2 that marks the end of a string constant, is not counted.

The `string` class also has special operators for working with strings. One of them is the + operator.

You have already encountered the + operator to add two numeric quantities. Because strings cannot be added, when this operator is used with string operands it *concatenates* them, or joins them together. Assume we have the following definitions and initializations in a program.

```
string greeting1 = "Hello ",
       greeting2;
string name1     = "World";
string name2     = "People";
```

The following statements illustrate how string concatenation works.

```
greeting2 = greeting1 + name1; // greeting2 now holds "Hello World"
greeting1 = greeting1 + name2; // greeting1 now holds "Hello People"
```

Notice that the string stored in `greeting1` has a blank as its last character. If the blank were not there, `greeting2` would have been assigned the string `"HelloWorld"`.

The last statement could also have been written using the += combined assignment operator.

```
greeting1 += name2;
```

You will learn about other useful `string` class functions and operators in Chapter 12.

3.10 Using C-Strings

CONCEPT: C-strings provide another way to store and work with strings.

In C, and in C++ prior to the introduction of the `string` class, strings were stored as a set of individual characters. A group of contiguous 1-byte memory cells was set up to hold them, with each cell holding just one character of the string. A group of memory cells like this is called an *array*. You will learn more about arrays in Chapter 8, but for now all you need to know is how to set one up and use it to hold and work with the characters that make up a string.

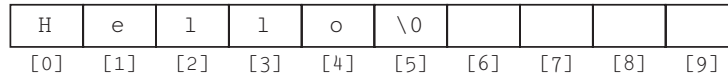
Because this was the way to create a string variable in C, a string defined in this manner is called a *C-string*. Here is a statement that defines `word` to be an array of characters that will hold a C-string and initializes it to `"Hello"`.

```
char word[10] = "Hello";
```

Notice that the way we define `word` is similar to the way we define any other variable. The data type is specified first and then the variable name is given. The only difference is the `[10]` that follows the name of the variable. This is called a *size declarator*. It tells how many memory cells to set up to hold the characters in the C-string.

As with string constants, the null character is automatically appended to the end of a C-string to mark its end. Figure 3-4 shows what the contents of the `word` variable would look like in memory.

Figure 3-4



Because one space must be reserved for the null terminator, `word` can only hold a string of up to nine characters.

Like string objects, C-strings can have their contents input using `cin`, and they can have their contents displayed using `cout`. This is illustrated in Program 3-25. Because `name` is defined in line 8 to have 12 memory cells, it can store a name of up to 11 characters. No special header file is needed to use C-strings.

Program 3-25

```
1  // This program uses cin >> to read a word into a C-string.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     const int SIZE = 12;
8     char name[SIZE];        // name is a set of 12 memory cells
9
10    cout << "Please enter your first name." << endl;
11    cin  >> name;
12    cout << "Hello, " << name << endl;
13    return 0;
14 }
```

Program Output with Example Input Shown in Bold

```
Please enter your first name.
Sebastian[Enter]
Hello, Sebastian
```

Except for inputting and displaying them with `cin >>` and `cout <<`, almost everything else about using string objects and C-strings is different. This is because the `string` class includes functions and operators that save the programmer having to worry about many of the details of working with strings. When using C-strings, however, it is the responsibility of the programmer to handle these things.

Because C-strings are harder to work with than string objects, you might be wondering why you are learning about them. There are two reasons. First, you are apt to encounter older programs that use them, so you need to understand them. Second, even though strings can now be declared as string objects in most cases, there are still times when only C-strings will work. You will be introduced to some of these cases later in the book.

Assigning a Value to a C-String

The first way in which using a C-string differs from using a string object is that, except for initializing it at the time of its definition, it cannot be assigned a value using the assignment operator. In Program 3-25 we could not, for example, replace the `cin` statement with the following line of code.

```
name = "Sebastian";                    // Wrong!
```

Instead, to assign a value to a C-string, we must use a function called `strcpy` (pronounced *string copy*) to copy the contents of one string into another. In the following line of code `Cstring` is the name of the variable receiving the value, and `value` is either a string constant or the name of another C-string variable.

```
strcpy(Cstring, value);
```

Program 3-26 shows how the `strcpy` function works.

Program 3-26

```
1  // This program uses the strcpy function to copy one C-string to another.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     const int SIZE = 12;
8     char name1[SIZE],
9          name2[SIZE];
10
11    strcpy(name1, "Sebastian");
12    cout << "name1 now holds the string " << name1 << endl;
13
14    strcpy(name2, name1);
15    cout << "name2 now also holds the string " << name2 << endl;
16
17    return 0;
18 }
```

Program Output

```
name1 now holds the string Sebastian
name2 now also holds the string Sebastian
```

Keeping Track of a How Much a C-String Can Hold

Another crucial way in which using a C-string differs from using a string object involves the memory allocated for it. With a string object, you do not have to worry about there being too little memory to hold a string you wish to place in it. If the storage space allocated to the string object is too small, the `string` class functions will make sure more memory is allocated to it. With C-strings this is not the case. The number of memory cells set aside to hold a C-string remains whatever size you originally set it to in the definition statement. It is the job of the programmer to ensure that the number of characters placed in it does not exceed the storage space. If the programmer uses `cin` to read a value into a C-string and the user types in more characters

than it can hold, `cin` will store all the characters anyway. The ones that don't fit will spill over into the following memory cells, overwriting whatever was previously stored there. This type of error, known as a *buffer overrun*, can lead to serious problems.

One way to prevent this from happening is to use the `setw` stream manipulator. This manipulator, which we used earlier in this chapter to format output, can also be used to control the number of characters that `cin >>` inputs on its next read, as illustrated here:

```
char word[5];
cin >> setw(5) >> word;
```

Another way to do the same thing is by using the `cin width` function.

```
char word[5];
cin.width(5);
cin >> word;
```

In both cases the field width specified is 5 and `cin` will read, at most, one character less than this, leaving room for the null character at the end. Program 3-27 illustrates the use of the `setw` manipulator with `cin`, while Program 3-28 uses its `width` function. Both programs produce the same output.

Program 3-27

```
1  // This program uses setw with the cin object.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  int main()
7  {
8     const int SIZE = 5;
9     char word[SIZE];
10
11    cout << "Enter a word: ";
12    cin  >> setw(SIZE) >> word;
13    cout << "You entered " << word << endl;
14
15    return 0;
16 }
```

Program 3-28

```
1  // This program uses cin's width function.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  int main()
7  {
8     const int SIZE = 5;
9     char word[SIZE];
```

(program continues)

Program 3-28 *(continued)*

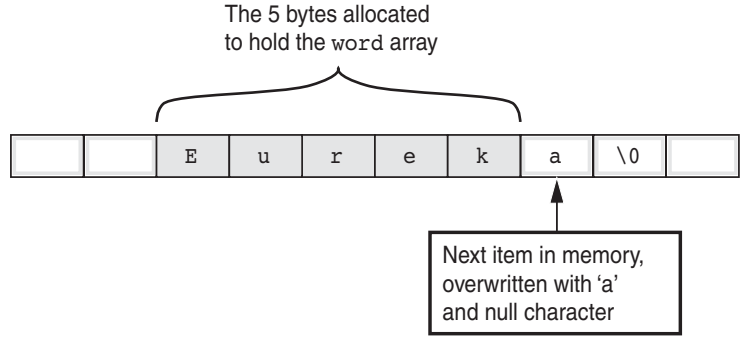
```
10
11    cout << "Enter a word: ";
12    cin.width(SIZE);
13    cin  >> word;
14    cout << "You entered " << word << endl;
15
16    return 0;
17 }
```

Program Output for Programs 3-27 and 3-28 with Example Input Shown in Bold

```
Enter a word: Eureka[Enter]
You entered Eure
```

In Program 3-28, `cin` only reads and stores four characters into `word`. If the field width had not been specified, `cin` would have written the entire word "Eureka" into memory, overflowing the space set up to hold `word`. Figure 3-5 illustrates the way memory would have been affected by this. The shaded area is the 5 bytes of memory allocated to hold the C-string.

Figure 3-5



There are three important points to remember about the way `cin` handles field widths:

- The field width only pertains to the very next item entered by the user.
- To leave space for the '\0' character, the maximum number of characters read and stored will be one less than the size specified.
- If `cin` comes to a whitespace character before reading the specified number of characters, it will stop reading.

Reading a Line of Input

Still another way in which using C-strings differs from using string objects is that you must use a different set of functions when working with them. To read a line of input, for example, you must use `cin.getline` rather than `getline`. These two names look a lot alike, but they are two different functions and are not interchangeable. Like `getline`, `cin.getline` allows you to read in a string containing spaces. It will continue reading until it has read the maximum specified number of characters, or until the [Enter] key is pressed. Here is an example of how it is used:

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

The `getline` function takes two arguments separated by a comma. The first argument is the name of the array that the string is to be stored in. In the statement above, the name of the array is `sentence`. The second argument is the size of the array. `cin` will read up to one character less than this number, leaving room for the null terminator. This eliminates the need for using the `setw` manipulator or the `width` function. The statement above will read up to 19 characters. The null terminator will automatically be placed in the array, after the last character. Program 3-29 shows the `getline` function being used to read a sentence of up to 80 characters.

Program 3-29

```
1  // This program demonstrates cin's getline function
2  // to read a line of text into a C-string.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int SIZE = 81;
9     char sentence[SIZE];
10
11    cout << "Enter a sentence: ";
12    cin.getline(sentence, SIZE);
13    cout << "You entered " << sentence << endl;
14    return 0;
15 }
```

Program Output with Example Input Shown in Bold

```
Enter a sentence: To be, or not to be, that is the question.[Enter]
You entered To be, or not to be, that is the question.
```

Later chapters cover more on C-strings and how they differ from string objects.



Checkpoint

3.26 Will the following string constant fit in the space allocated for `name`? Why or why not?

```
char name[4] = "John";
```

3.27 If a program contains the definition `string name;` indicate whether each of the following lettered program statements is legal or illegal.

A) `cin >> name;`
B) `cin.getline(name, 20);`
C) `cout << name;`
D) `name = "John";`

3.28 If a program contains the definition `char name[20];` indicate whether each of the following lettered program statements is legal or illegal.

A) `cin >> name;`
B) `cin.getline(name, 20);`
C) `cout << name;`
D) `name = "John";`

3.11 More Mathematical Library Functions

CONCEPT: The C++ run-time library provides several functions for performing complex mathematical operations.

Earlier in this chapter you learned to use the `pow` function to raise a number to a power. The C++ library has numerous other functions that perform specialized mathematical operations. These functions are useful in scientific and special purpose programs. Table 3-13 shows several of these, each of which requires the `cmath` header file.

Table 3-13 Selected Mathematical Library Functions

Function	Example	Description
`abs`	`y = abs(x);`	Returns the absolute value of the argument. The argument and the return value are integers.
`cos`	`y = cos(x);`	Returns the cosine of the argument. The argument should be an angle expressed in radians. The return type and the argument are doubles.
`exp`	`y = exp(x);`	Computes the exponential function of the argument, which is `x`. The return type and the argument are doubles.
`fmod`	`y = fmod(x, z);`	Returns, as a double, the remainder of the first argument divided by the second argument. Works like the modulus operator, but the arguments are doubles. (The modulus operator only works with integers.) Take care not to pass zero as the second argument. Doing so would cause division by zero.
`log`	`y = log(x);`	Returns the natural logarithm of the argument. The return type and the argument are doubles.
`log10`	`y = log10(x);`	Returns the base-10 logarithm of the argument. The return type and the argument are doubles.
`round`	`y = round(x);`	Returns the argument rounded to the nearest whole number. The return value is an integer.
`sin`	`y = sin(x);`	Returns the sine of the argument. The argument should be an angle expressed in radians. The return type and the argument are doubles.
`sqrt`	`y = sqrt(x);`	Returns the square root of the argument. The return type and argument are doubles. The argument must be zero or greater.
`tan`	`y = tan(x);`	Returns the tangent of the argument. The argument should be an angle expressed in radians. The return type and the argument are doubles.

With the exception of the `abs` and `round` functions, all of the functions listed in Table 3-13 take one or more `double` arguments and return a `double` value. However, most C++ compilers

allow them to be called with `int` arguments as well. So, for example, both of the following will work to print the square root of 30.

```
cout << sqrt(30.0);   // Displays 5.47723
cout << sqrt(30);     // Displays 5.47723
```

Program 3-30 shows the `sqrt` function being used to find the hypotenuse of a right triangle. The program uses the following formula, taken from the Pythagorean theorem:

$$c = \sqrt{a^2 + b^2}$$

In the formula, c is the length of the hypotenuse, and a and b are the lengths of the other sides of the triangle.

Program 3-30

```
1  // This program inputs the lengths of the two sides of a right
2  // triangle, then calculates and displays the length of the hypotenuse.
3  #include <iostream>
4  #include <cmath>        // Needed to use the sqrt function
5  using namespace std;
6
7  int main()
8  {
9     double a, b, c;
10
11    // Get the length of the two sides
12    cout << "Enter the length of side a: ";
13    cin  >> a;
14    cout << "Enter the length of side b: ";
15    cin  >> b;
16
17    // Compute and display the length of the hypotenuse
18    c = sqrt(pow(a, 2.0) + pow(b, 2.0));
19
20    cout << "The length of the hypotenuse is ";
21    cout << c << endl;
22    return 0;
23 }
```

Program Output with Example Input Shown in Bold

```
Enter the length of side a: 5.0[Enter]
Enter the length of side b: 12.0[Enter]
The length of the hypotenuse is 13
```

The following statement, taken from line 18 of Program 3-30, calculates the square root of the sum of the squares of the triangle's two sides:

```
c = sqrt(pow(a, 2.0) + pow(b, 2.0));
```

Notice that the following mathematical expression is used as the `sqrt` function's argument:

```
pow(a, 2.0) + pow(b, 2.0)
```

This expression calls the pow function twice: once to calculate the square of a and again to calculate the square of b. These two squares are then added together, and the sum is sent to the sqrt function.

Random Numbers

Some programs need to use randomly generated numbers. The C++ library has a function called rand() for this purpose. To use the rand() function, you must include the cstdlib header file in your program. The number returned by the function is an int. Here is an example of how it is used.

```
randomNum = rand();
```

However, the numbers returned by the function are really *pseudorandom*. This means they have the appearance and properties of random numbers, but in reality are not random. They are actually generated with an algorithm. The algorithm needs a starting value, called a *seed*, to generate the numbers. If it is not given one, it will produce the same stream of numbers each time it is run. Program 3-31 illustrates this.

Program 3-31

```
1  // This program demonstrates what happens in C++ if you
2  // try to generate random numbers without setting a "seed".
3  #include <iostream>
4  #include <cstdlib>
5  using namespace std;
6
7  int main ()
8  {
9     int num1, num2, num3;  // These hold the 3 random numbers
10
11    // Now generate and print three random numbers
12    num1 = rand();
13    num2 = rand();
14    num3 = rand();
15    cout << num1 << "      " << num2 << "      " << num3 << endl;
16    return 0;
17 }
```

Program Output from Run 1

```
41      18467      6334
```

Program Output from Run 2

```
41      18467      6334
```

Program 3-32 illustrates what happens when random numbers are generated after giving the generator a seed to start with. In C++ this is done by calling the srand() function and passing it a positive integer. The srand() function should be called only once for the whole program.

Notice in line 10 of the program that the variable created to hold the seed is unsigned. As you may recall, this data type holds only non-negative integers. This is the data type the srand function expects to receive when it is called and it guarantees that no negative numbers will be sent to srand.

Program 3-32

```
1  // This program demonstrates random numbers, providing
2  // a "seed" for the random number generator.
3  #include <iostream>
4  #include <cstdlib>
5  using namespace std;
6
7  int main()
8  {
9     int num1, num2, num3;  // These hold the 3 random numbers
10    unsigned seed;         // Random generator seed
11
12     // Get a "seed" value from the user
13    cout << "Enter a seed value: ";
14    cin  >> seed;
15
16    // Set the random generator seed before calling rand()
17    srand(seed);
18
19    // Now generate and print three random numbers
20    num1 = rand();
21    num2 = rand();
22    num3 = rand();
23    cout << num1 << "      " << num2 << "      " << num3 << endl;
24    return 0;
25 }
```

Program Output with Example Input Shown in Bold

Run 1:

```
Enter a seed value: 19[Enter]
100      15331      209
```

Run 2:

```
Enter a seed value: 171[Enter]
597      10689      28587
```

As you can see from the Program 3-32 output, when the program is run with a different seed, a different stream of random numbers is generated. However, if we run the program a third time using 19 or 171 as the seed again, we will get exactly the same numbers we did the first time.



NOTE: The stream of random numbers generated on your computer system may be different.

Notice that on line 14 of Program 3-32 `cin` is used to get a value from the user for the random number generator seed. Another common practice for getting a seed value is to call the `time` function, which is part of the standard library. This function returns the number of seconds that have elapsed since midnight, January 1, 1970. To use the `time` function in a program you must include the `ctime` header file, and you must pass `0` as an argument to the function when you call it. The following code segment illustrates how to "seed" the random number generator with a value obtained this way.

```
seed = time(0);
srand(seed);
```

If you wish to limit the range of the random number to an integer between 1 and `maxRange`, use the following formula.

```
randomNum = 1 + rand() % maxRange;
```

For example, if you wish to generate a random number in the range of 1 through 6 to represent the roll of a dice, you would use

```
dice = 1 + rand() % 6;
```



NOTE: The `mod` operation gives us the remainder of an integer divide. When the integer returned by `rand()` is divided by 6, the remainder will be a number between 0 and 5. Because we want a number between 1 and 6, we simply add 1 to the result.



Checkpoint

3.29 Assume the variables `angle1` and `angle2` hold angles stored in radians. Write a statement that adds the sine of `angle1` to the cosine of `angle2` and stores the result in the variable `x`.

3.30 To find the cube root (the third root) of a number, raise it to the power of 1/3. To find the fourth root of a number, raise it to the power of 1/4. Write a statement that will find the fifth root of the variable `x` and store the result in the variable `y`.

3.31 The cosecant of the angle a is

$$\frac{1}{\sin a}$$

Write a statement that calculates the cosecant of the angle stored in the variable `a` and stores it in the variable `y`.

3.12 Introduction to Files

CONCEPT: Program input can be read from a file and program output can be written to a file.

The programs you have written so far require you to re-enter data each time the program runs. This is because the data stored in RAM disappears once the program stops running or the computer is shut down. If a program is to retain data between the times it runs, it must have a way of saving it. Data is saved in a file, which is usually stored on a computer's disk. Once the data is saved by writing it into a file, it will remain there after the program stops running. The data can then be retrieved and used at a later time.

There are five steps that must be taken when a file is used by a program:

1. Include the header file needed to perform file input/output.
2. Define a file stream object.

3. Open the file.
4. Use the file.
5. Close the file.

Step 1: Include the header file needed to perform file input/output.

Just as `cin` and `cout` require the `iostream` file to be included in the program, C++ file access requires another header file. The file `fstream` contains all the definitions necessary for file operations. It is included with the following statement:

```
#include <fstream>
```

Step 2: Define a file stream object.

The next step in setting up a program to perform file I/O is to define one or more *file stream objects*. They are called stream objects because a file can be thought of as a stream of data. File stream objects work very much like `cin` and `cout` objects. A stream of data can be sent to `cout`, which causes values to be displayed on the screen. A stream of data can be read from the keyboard by `cin` and stored in variables. Likewise, streams of data can be sent to a file stream object, which writes the data to a file. Data that is read from a file flows from a file stream object into other variables.

The `fstream` header file contains definitions for the data types `ofstream`, `ifstream`, and `fstream`. Before a C++ program can work with a file, it must define an object of one of these data types. The object will be associated with an actual file stored on some secondary storage medium, and the operations that may be performed on that file depend on which of these three data types you pick for the file stream object. Table 3-14 lists and describes file stream data types.

Table 3-14 File Stream Data Types

File Stream Data Type	Description
`ofstream`	Output file stream. This data type can be used to open *output* files and write data to them. If the file does not yet exist, the `open` operation will automatically create it. If the file already exists, the `open` operation will destroy it and create a new, empty file of the same name in its place. With the `ofstream` data type, data may only be copied from variables to the file, but not vice versa.
`ifstream`	Input file stream. This data type can be used to open existing *input* files and read data from them into memory. With the `ifstream` data type, data may only be copied from the file into variables, not but vice versa.
`fstream`	File stream. This data type can be used to open files, write data to them, and read data from them. With the `fstream` data type, data may be copied from variables into a file, or from a file into variables.



NOTE: In this section we only discuss the `ofstream` and `ifstream` types. The `fstream` type is covered in Chapter 13.

Here are example statements that define `ofstream` and `ifstream` objects:

```
ofstream outputFile;
ifstream inputFile;
```

These two file stream objects, `outputFile` and `inputFile`, could have been named using any legal C++ identifier names. However, as is good programming practice, they were given descriptive names that clarify their use. The `outputFile` object is of the `ofstream` type, so data can be written to any file associated with it. The `inputFile` object is of the `ifstream` type, so data can be read from any file it is associated with.

Step 3: Open the file.

Before data can be written to or read from a file, the file must be opened. Outside of the C++ program, a file is identified by its name. Inside a C++ program, however, a file is identified by a stream object. The object and the file name are linked when the file is opened.

Files can be opened through the `open` function that exists for file stream objects. Assume `inputFile` is an `ifstream` object, defined as

```
ifstream inputFile;
```

The following statement uses `inputFile` to open a file named `customer.dat`:

```
inputFile.open("customer.dat");     // Open an input file
```

The argument to the open function in this statement is the name of the file. This links the file `customer.dat` with the stream object `inputFile`. Until `inputFile` is associated with another file, any operations performed with it will be carried out on the file `customer.dat`.

It is also possible to define a file stream object and open a file all in one statement. Here is an example:

```
ifstream inputFile("customer.dat");
```

In our example `open` statement, the `customer.dat` file was specified as a simple file name, with no path given. When no path is given, the program will look for the file in a default directory. If the program is being executed from the command line, the default directory is the current directory. If the program is being executed from within an integrated development environment (IDE), the default directory depends on the particular compiler you are using. Further information on default directories can be found in Appendices L and M on wxDev-C++ and Microsoft Visual C++, and your instructor can provide you with specific information for your particular system. If the file you want to open is not in the default directory, you will need to specify its location as well as its name.

For example, on a Windows system the following statement opens file `C:\data\inventory.dat` and associates it with `outputFile`.

```
outputFile.open("C:\\data\\invetory.dat");
```

Notice the use of the double back slashes in the file's path. This is because, as mentioned earlier, two back slashes are needed to represent one backslash in a string.



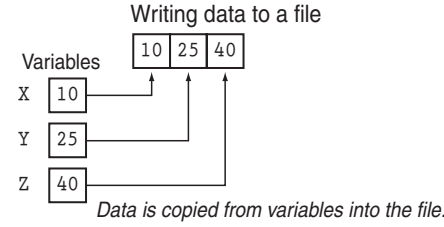
NOTE: Some systems cannot handle file names that contain spaces. In this case, the entire pathname should be enclosed in an additional set of quotation marks. For example, if the file above had been named `Truck Inventory`, it could be opened like this:

```
outputFile.open("\"C:\\data\\Truck Inventory.dat\"");
```

Step 4: Use the file.

Now that the file is open and can be accessed through a file stream object, you are ready to use it. When a program is actively working with data, the data is located in random-access memory, usually in variables. When data is written into a file, it is copied from variables into the file. This is illustrated in Figure 3-6.

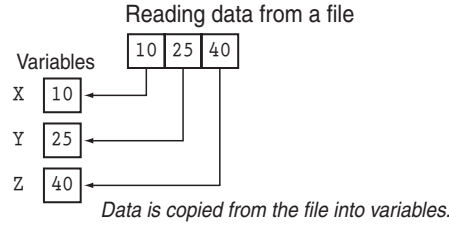
Figure 3-6



Data is copied from variables into the file.

When data is read from a file, it is copied from the file into variables. Figure 3-7 illustrates this.

Figure 3-7



Data is copied from the file into variables.

Writing information to a file

You already know how to use the stream insertion operator (`<<`) with the `cout` object to write information to the screen. It can also be used with file stream objects to write information to a file. Assuming `outputFile` is a file stream object, the following statement demonstrates using the << operator to write a string to a file:

```
outputFile << "I love C++ programming";
```

As you can see, the statement looks like a `cout` statement, except the file stream object name replaces `cout`. Here is a statement that writes both a string and the contents of a variable to a file:

```
outputFile << "Price: " << Price;
```

This statement writes the stream of information to `outputFile` exactly as `cout` would write it to the screen.

Program 3-33 demonstrates opening a file, writing information to it, and closing it.

Program 3-33

```
1  // This program uses the << operator to write information to a file.
2  #include <iostream>
3  #include <fstream>                    // Needed to use files
4  using namespace std;
5
6  int main()
7  {
8     ofstream outputFile;
9     outputFile.open("demofile.txt");
10
11    cout << "Now writing information to the file.\n";
12    // Write 3 great names to the file
13    outputFile << "Bach\n";
14    outputFile << "Beethoven\n";
15    outputFile << "Mozart\n";
16
17    // Close the file
18    outputFile.close();
19    cout << "Done.\n";
20    return 0;
21 }
```

Program Screen Output

```
Now writing information to the file.
Done.
```

Output to File demofile.txt

```
Bach
Beethoven
Mozart
```

Output files have two common uses. The first is to hold computer program output. When a program writes output to a file, rather than to the computer screen, it can be saved for later viewing and printing. This is often done with reports that would not easily fit on one screen or that must be printed more than once. The second common use of output files is to hold data generated by one program that will later be read in by another program. Because these are simple text files, they can be viewed and printed with any text editor.

Reading information from a file

The >> operator not only reads user input from the `cin` object, but it can also be used to read data from a file. Assuming `inFile` is a file stream object, the following statement shows the >> operator reading data from the file into the variable `name`:

```
inFile >> name;
```

In Program 3-33, the file `demofile.txt` was created and the following list of names was stored there.

```
Bach
Beethoven
Mozart
```

Program 3-34 demonstrates the use of the >> operator to read the names from the file and store them in a variable.

Program 3-34

```
1  // This program uses the >> operator to read information from a file.
2  #include <iostream>
3  #include <fstream>                  // Needed to use files
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     ifstream inFile;
10    string name;
11
12    inFile.open("demofile.txt");
13    cout << "Reading information from the file.\n\n";
14
15    inFile >> name;                  // Read name 1 from the file
16    cout << name << endl;            // Display name 1
17
18    inFile >> name;                  // Read name 2 from the file
19    cout << name << endl;            // Display name 2
20
21    inFile >> name;                  // Read name 3 from the file
22    cout << name << endl;            // Display name 3
23
24    inFile.close();                  // Close the file
25    cout << "\nDone.\n";
26    return 0;
27 }
```

Program Screen Output

```
Reading information from the file.

Bach
Beethoven
Mozart

Done.
```

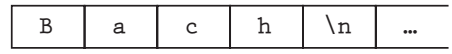
Data is read from files in a sequential manner. When a file is first opened, the file stream object's *read position* is at the first byte of the file. The first read operation extracts data starting at the first byte. As data is read, the file stream object's read position advances through the file.

When the >> operator extracts data from a file, it expects to read pieces of data that are separated by whitespace characters (spaces, tabs, or newlines). In Program 3-34, the following statement reads a string from the file:

```
inFile >> name;
```

The >> operator extracts a string in this case because `name` is a string object. Figure 3-8 shows the first five bytes in the file:

Figure 3-8



The >> operator will extract all of the characters up to the newline, so "Bach" is the first string read from the file. After "Bach" is extracted, the file stream object will be positioned so the following read operation would extract the string "Beethoven." This procedure is followed until all three strings have been read from the file.

The file of data read in by Program 3-34 was created by Program 3-33. However, this is not the only way to create a data file. A data file can be created with any text editor (such as Windows WordPad). This is often done when a program has a substantial amount of input. Placing the data in a text file ahead of time and then having the program read the data from the file saves the user having to enter the data when the program is run. Program 3-35, which finds the area of four rectangles, illustrates reading data from a text file named `dimensions.txt`, which was previously created with a text editor. Here is a sample of the file's contents. Each pair of numbers is the length and width of a different rectangle.

```
10    2
5     7
6     20
8     3
```

Program 3-35

```
1  // This program uses the >> operator to read rectangle dimensions
2  // from a file. It demonstrates that, as with cin, more than one
3  // value can be read in from a file with a single statement.
4  #include <iostream>
5  #include <fstream>
6  using namespace std;
7
8  int main()
9  {
10    ifstream inFile;
11    int length, width;
12
13    inFile.open("dimensions.txt");
14    cout << "Reading dimensions of 4 rectangles from the file.\n\n";
15
16    // Process rectangle 1
17    inFile >> length >> width;
18    cout << "Area of rectangle 1: " << (length * width) << endl;
19
20    // Process rectangle 2
21    inFile >> length >> width;
22    cout << "Area of rectangle 2: " << (length * width) << endl;
```

(program continues)

Program 3-35 *(continued)*

```
23
24    // Process rectangle 3
25    inFile >> length >> width;
26    cout << "Area of rectangle 3: " << (length * width) << endl;
27
28    // Process rectangle 4
29    inFile >> length >> width;
30    cout << "Area of rectangle 4: " << (length * width) << endl;
31
32    // Close the file
33    inFile.close();
34    cout << "Done.\n";
35
36    return 0;
37 }
```

Program Output with Example Input Shown in Bold

```
Reading dimensions of 4 rectangles from the file.

Area of rectangle 1: 20
Area of rectangle 2: 35
Area of rectangle 3: 120
Area of rectangle 4: 24
Done
```

Step 5: Close the file.

The opposite of opening a file is closing it. You have probably noticed that in Programs 3-33 through 3-35 the files were closed once the program was finished using them. Although a program's files are automatically closed when the program shuts down, it is a good programming practice to write statements that explicitly close them. Here are two reasons a program should close files when it is finished using them:

- Most operating systems temporarily store information in a *file buffer* before it is written to a file. A file buffer is a small holding section of memory that file-bound information is first written to. When the buffer is filled, all the information stored there is written to the file. This technique improves the system's performance. Closing a file causes any unsaved information that may still be held in a buffer to be saved to its file. This means the information will be in the file if you need to read it later in the same program.
- Some operating systems limit the number of files that may be open at one time. When a program keeps open files that are no longer being used, it uses more of the operating system's resources than necessary.

Calling the file stream object's `close` function closes a file. Here is an example:

```
outputFile.close();
```



Checkpoint

3.32 What header file must be included in a program to use files?

3.33 What is the difference between a file stream object with the data type `ifstream` and one with the data type `ofstream`?

3.34 Which program statement links a file stream object with an actual disk file?

3.35 Assuming `dataFile` is an `ofstream` object associated with a disk file named `payroll.dat`, which of the following statements would write the value of the `salary` variable to the file?

A) `cout << salary;`
B) `ofstream << salary;`
C) `dataFile << salary;`
D) `payroll.dat << salary;`

3.13 Focus on Debugging: *Hand Tracing a Program*

Hand tracing is a debugging process where you pretend that you are the computer executing a program. You step through each of the program's statements one by one. As you look at a statement, you record the contents that each variable will have after the statement executes. This process is often helpful in finding mathematical mistakes and other logic errors.

To hand trace a program you construct a chart with a column for each variable. The rows in the chart correspond to the lines in the program. For example, Program 3-36 is shown with a hand trace chart. The program uses the following four variables: `num1`, `num2`, `num3`, and `avg`. Notice that the hand trace chart has a column for each variable and a row for each line of code in function `main`.

Program 3-36 ***(with hand trace chart empty)***

```
1  // This program asks for three numbers, then displays
2  // their average. It contains a bug. Can you find it?
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     double num1, num2, num3, avg;
9
10    cout << "Enter the first number: ";
11    cin  >> num1;
12    cout << "Enter the second number: ";
13    cin  >> num2;
14    cout << "Enter the third number: ";
15    cin  >> num3;
16    avg = num1 + num2 + num3 / 3;
17    cout << "The average is " << avg << endl;
18  return 0;
19 }
```

num1	num2	num3	avg

(program continues)

Program 3-36 ***(with hand trace chart empty)****(continued)*

Program Output with Example Input Shown in Bold

```
Enter the first number:  10[Enter]
Enter the second number: 20[Enter]
Enter the third number:  30[Enter]
The average is 40
```

Notice that the program runs, but it displays an incorrect average. The correct average of 10, 20, and 30 is 20, not 40. To find the error we will hand trace the program.

To hand trace a program, you step through each statement, observe the operation that is taking place, and then record the contents of the variables after the statement executes. After the hand trace is complete, the chart will appear as follows. We have written question marks in the chart where we do not yet know the contents of a variable.

Program 3-36 ***(with hand trace chart filled in)***

```
1  // This program asks for three numbers, then displays
2  // their average. It contains a bug. Can you find it?
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     double num1, num2, num3, avg;
9
10    cout << "Enter the first number: ";
11    cin  >> num1;
12    cout << "Enter the second number: ";
13    cin  >> num2;
14    cout << "Enter the third number: ";
15    cin  >> num3;
16    avg = num1 + num2 + num3 / 3;
17    cout << "The average is " << avg << endl;
18    return 0;
19 }
```

num1	num2	num3	avg
?	?	?	?
?	?	?	?
10	?	?	?
10	?	?	?
10	20	?	?
10	20	?	?
10	20	30	?
10	20	30	40
10	20	30	40

Do you see the error? By examining the statement on line 16 that computes the average, we find a mistake. The division operation takes place before the addition operations, so we must rewrite that statement as

```
avg = (num1 + num2 + num3) / 3;
```

Hand tracing is a simple process that focuses your attention on each statement in a program. Often this helps you locate errors that are not obvious.

3.14 Green Fields Landscaping Case Study—Part 1

Problem Statement

One of the services provided by Green Fields Landscaping is the sale and delivery of mulch, which is measured and priced by the cubic yard. You have been asked to create a program that will determine the number of cubic yards of mulch the customer needs and the total price.

Program Design

Program Steps

The program must carry out the following general steps (This list of steps is sometimes called General Pseudocode):

1. Set the price for a cubic yard of mulch (currently 22.00).
2. Ask the user to input the number of square feet to be covered and the depth of the mulch to be spread over this area.
3. Calculate the number of cubic feet of mulch needed.
4. Calculate the number of cubic yards of mulch needed.
5. Calculate the total price for the mulch.
6. Display the results.

Variables whose values will be input

```
double squareFeet            // square feet of land to be covered
int    depth                 // how many inches deep the mulch is to be spread
```

Variables whose values will be output

```
double cubicYards            // number of cubic yards of mulch needed
double totalPrice            // total price for all the cubic yards ordered
```

Program Constants

```
double PRICE_PER_CUBIC_YD    // the price for 1 delivered cubic yard of mulch
```

Additional Variables

```
double cubicFeet             // number of cubic feet of mulch needed
```

Detailed Pseudocode (including actual variable names and needed calculations)

```
PRICE_PER_CUBIC_YD = 22.00
Input squareFeet             // with prompt
Input depth                  // with prompt
cubicFeet = squareFeet * (depth / 12.0)
```

```
cubicYards = cubicFeet / 27
totalPrice = cubicYards * PRICE_PER_CUBIC_YD
Display cubicYards, PRICE_PER_CUBIC_YD, and totalPrice
```

The Program

The next step, after the pseudocode has been checked for logic errors, is to expand the pseudocode into the final program. This is shown in Program 3-37.

Program 3-37

```
1  // Program used by Green Fields Landscaping to determine the number of
2  // cubic yards of mulch a customer needs and its price.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  const double PRICE_PER_CUBIC_YD = 22.00;
8
9  int main()
10 {
11    double squareFeet;   // square feet of land to be covered
12    int    depth;        // inches deep the mulch is to be spread
13    double cubicFeet,    // number of cubic feet of mulch needed
14           cubicYards,   // number of cubic yards of mulch needed
15           totalPrice;   // total price for all the cubic yards ordered
16
17    // Get inputs
18    cout << "Number of square feet to be covered with mulch: ";
19    cin  >> squareFeet;
20    cout << "Number of inches deep: ";
21    cin  >> depth;
22
23    // Perform calculations
24    cubicFeet = squareFeet * (depth / 12.0);
25    cubicYards = cubicFeet / 27;
26    totalPrice = cubicYards * PRICE_PER_CUBIC_YD;
27
28    // Display outputs
29    cout << "\n Number of cubic yards needed: " << cubicYards << endl;
30    cout << fixed << showpoint << setprecision(2);
31    cout << "Price per cubic yard: $" << setw(7)
32         << PRICE_PER_CUBIC_YD << endl;
33    cout << "Total price:          $" << setw(7)
34         << totalPrice << endl << endl;
35
36    return 0;
37 }
```

(program continues)

Program 3-37 *(continued)*

Program Output with Example Input Shown in Bold

```
Number of square feet to be covered with mulch: 270[Enter]
Number of inches deep: 12[Enter]

Number of cubic yards needed: 10
Price per cubic yard: $  22.00
Total price:          $ 220.00
```

Program Output with Different Example Input Shown in Bold

```
Number of square feet to be covered with mulch: 800[Enter]
Number of inches deep: 3[Enter]

Number of cubic yards needed: 7.40741
Price per cubic yard: $  22.00
Total price:          $ 162.96
```

General Crates, Inc., Case Study

The following additional case study, which contains applications of material introduced in Chapter 3, can be found on the student CD.

This case study develops a program that accepts the dimensions on a crate to be built and outputs information on its volume, building cost, selling cost, and profit. The case study illustrates the major program development steps: initial problem statement, program design using hierarchy charts and pseudocode, development of the algorithm needed to create the outputs, source code for the final working program, and output created by running the program with several test cases.

3.15 Tying It All Together: *Word Game*

With the programming knowledge you have learned so far, you can start constructing simple games. Here is one that creates a program to play a word game. It will ask the player to enter the following:

- their name (`name`)
- the name of a city (`city`)
- a fun activity (`activity`)
- a type of animal (`animal`)
- a food or product you can buy (`product`)
- an adjective noun (`petname`)
- a number between 10 and 50 (`age`)
- a number between 0 and 15 (`kids`)

Then it will display a story using those words.

Program 3-38

```
// This program uses strings to play a word game.
#include <iostream>
#include <string>
using namespace std;

int main()
{  // Stored strings
   string s1 = "There once was a person named ",
          s2 = " who lived in ",
          s3 = "\nand who loved ",
          s4 = ". At the age of ",
          s5 = ", ",
          s6 = " graduated \nfrom high school and went to work in a ",
          s7 = " factory.\n",
          s8 = " got married and had ",
          s9 = " children and a pet ",
          s10= " named ",
          s11= ".\nEvery weekend the family and ",
          s12= " had fun ",
          s13= " together.";

   // Values input by the user
   string name, city, activity, animal, product, petName;
   int age, kids;

   cout << "Enter the following information and I\'ll "
        << "tell you a story.\n\n";
   cout << "Your name: ";
   getline(cin, name);

   cout << "The name of a city: ";
   getline(cin, city);

   cout << "A physical activity (e.g. jogging, playing baseball): ";
   getline(cin, activity);

   cout << "An animal: ";
   getline(cin, animal);

   cout << "A food or product you can buy: ";
   getline(cin, product);

   cout << "An adjective noun (e.g. blue car): ";
   getline(cin, petName);

   cout << "A number between 10 and 50: ";
   cin  >> age;

   cout << "A number between 0 and 15: ";
   cin  >> kids;

```

(program continues)

Program 3-38 *(continued)*

```
52      cout << endl << s1 << name << s2 << city << s3 << activity;
53      cout << s4 << age << s5 << name << s6 << product << s7;
54      cout << name << s8 << kids << s9 << animal << s10 << petName;
55      cout << s11 << petName << s12 << activity << s13 << endl;
56
57     return 0;
58  }
```

Sample Run with User Input Shown in Bold

```
Enter the following information and I'll tell you a story.

Your name: Joe[Enter]
The name of a city: Honolulu[Enter]
A physical activity (e.g. jogging, playing baseball): scuba diving[Enter]
An animal: bear[Enter]
A food or product you can buy: potato chips[Enter]
An adjective noun (e.g. blue car): dish rag[Enter]
A number between 10 and 50: 20[Enter]
A number between 0 and 15: 10[Enter]

There once was a person named Joe who lived in Honolulu
and who loved scuba diving. At the age of 20, Joe graduated
from high school and went to work in a potato chips factory.
Joe got married and had 10 children and a pet bear named dish rag.
Every weekend the family and dish rag had fun scuba diving together.
```

Try running this program with a variety of inputs. Then try modifying it to make up new stories.

Review Questions and Exercises

Short Answer

1. Assume a string object has been defined as follows:

   ```
   string description;
   ```

   A) Write a `cin` statement that reads in a one word description.
   B) Write a statement that reads in a description that can contain multiple words separated by blanks.

2. Write a definition statement for a character array large enough to hold any of the following strings:

   ```
   "Billy Bob's Pizza"
   "Downtown Auto Supplies"
   "Betty Smith School of Architecture"
   "ABC Cabinet Company"
   ```

3. Assume the array `name` is defined as follows:

   ```
   char name[25];
   ```

A) Using a stream manipulator, write a `cin` statement that will read a string into `name`, but will read no more characters than `name` can hold.
B) Using the `getline` function, write a `cin` statement that will read a string into `name` but that will read no more characters than `name` can hold.

4. Assume the following variables are defined:

```
int age;
double pay;
char section;
```

Write a single `cin` statement that will read input into each of these variables.

5. What header files must be included in the following program?

```
int main()
{
   double amount = 89.7;
   cout << fixed << showpoint << setprecision(1);
   cout << setw(8) << amount << endl;
   return 0;
}
```

6. Write a definition statement for a character array named `city`. It should be large enough to hold a string 30 characters in length.

7. Assume the following preprocessor directive appears in a program:

```
#define SIZE 12
```

How will the preprocessor rewrite the following lines?

A) `price = SIZE * unitCost;`
B) `cout << setw(SIZE) << 98.7;`
C) `cout << SIZE;`

8. Complete the following table by writing the value of each expression in the Value column.

Expression	Value
`28 / 4 - 2`	
`6 + 12 * 2 - 8`	
`4 + 8 * 2`	
`6 + 17 % 3 - 2`	
`2 + 22 * (9 - 7)`	
`(8 + 7) * 2`	
`(16 + 7) % 2 - 1`	
`12 / (10 - 6)`	
`(19 - 3) * (2 + 2) / 4`	

9. Write C++ expressions for the following algebraic expressions:

A) $a = 12x$

B) $z = 5x + 14y + 6k$

C) $y = x^4$

D) $g = \frac{h + 12}{4k}$

E) $c = \frac{a^3}{b^2k^4}$

10. Assume a program has the following variable definitions

```
int units;
float mass;
double weight;
```

and the following statement:

```
weight = mass * units;
```

Which automatic data type conversions will take place?

11. Assume a program has the following variable definitions

```
int a, b = 2;
double c = 4.3;
```

and the following statement:

```
a = b * c;
```

What value will be stored in `a`?

12. Assume that `qty` and `salesReps` are both integers. Use a type cast expression to rewrite the following statement so it will no longer perform integer division.

```
unitsEach = qty / salesReps;
```

13. Rewrite the following variable definition so the variable is a named constant with the value 12.

```
int rate;
```

14. Complete the following table by writing statements with combined assignment operators in the right-hand column. The statements should be equivalent to the statements in the left-hand column.

Statements with Assignment Operator	Statements with Combined Assignment Operator
`x = x + 5;`	
`total = total + subtotal;`	
`dist = dist / rep;`	
`ppl = ppl * period;`	
`inv = inv - shrinkage;`	
`num = num % 2;`	

15. Write a multiple assignment statement that can be used instead of the following group of assignment statements:

```
east = 1;
west = 1;
north = 1;
south = 1;
```

16. Replace the following statements with a single statement that initializes `sum` to 0 at the time it is defined.

```
int sum;
sum = 0;
```

17. Is the following code legal? Why or why not?

```
const int DAYS_IN_WEEK;
DAYS_IN_WEEK = 7;
```

18. Write a `cout` statement so the variable `divSales` is displayed in a field of eight spaces, in fixed-point notation, with a decimal point and two decimal digits.

19. Write a `cout` statement so the variable `profit` is displayed in a field of 12 spaces, in fixed-point notation, with a decimal point and four decimal digits.

20. What header file must be included
    A) to perform mathematical functions like `sqrt`?
    B) to use files?
    C) to use stream manipluators like `setprecision`?

Algorithm Workbench

21. A bowling alley is offering a prize to the bowler whose average score from bowling three games is the lowest. Write a pseudocode algorithm for a program that inputs three bowling scores and calculates and displays their average.

22. Pet World offers a 15% discount to senior citizens. Write a pseudocode algorithm for a program that inputs the amount of a sale, then calculates and displays both the amount the customer saves and the amount they must pay.

23. A retail store grants its customers a maximum amount of credit. Each customer's available credit is his or her maximum amount of credit minus the amount of credit used. Write a pseudocode algorithm for a program that asks for a customer's maximum credit and amount of credit used, then calculates and displays the customer's available credit.

24. Little Italy Pizza charges $12.00 for a 12-inch diameter sausage pizza and $14.00 for a 14-inch diameter sausage pizza. Write the pseudocode algorithm that calculates and displays how much each of these earns the establishment per square inch of pizza sold. (Hint: you will need to first calculate how many square inches there are in each pizza.)

Predict the Output

25. Trace the following programs and tell what each will display. (Some require a calculator.)
    A) *(Assume the user enters 38711. Use a calculator.)*

```
#include <iostream>
using namespace std;
```

```
int main()
{
   double salary, monthly;

   cout << "What is your annual salary? ";
   cin  >> salary;
   monthly = static_cast<int>(salary) / 12;
   cout << "Your monthly wages are " << monthly << endl;
   return 0;
}
```

B)
```
#include <iostream>
using namespace std;

int main()
{
   long x, y, z;
   x = y = z = 4;
   x += 2;
   y -= 1;
   z *= 3;
   cout << x << " " << y << " " << z << endl;
   return 0;
}
```

C)
```
#include <iostream>
using namespace std;
#define WHO "Columbus"
#define DID "sailed"
#define WHAT "the ocean blue."

int main()
{
   const int WHEN = 1492;
   cout << "In " << WHEN << " " << WHO << " "
        << DID << " " << WHAT << endl;
   return 0;
}
```

26. A) *(Assume the user enters George Washington.)*

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
int main()
{
   string userInput;

   cout << "What is your name? ";
   cin  >> userInput;
   cout << "Hello " << userInput << endl;
   return 0;
}
```

B) *(Assume the user enters George Washington.)*

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

int main()
{
   string userInput;

   cout << "What is your name? ";
   getline(cin, userInput);
   cout << "Hello " << userInput << endl;
   return 0;
}
```

C) *(Assume the user enters 36720152. Use a calculator.)*

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   long seconds;
   double minutes, hours, days, months, years;

   cout << "Enter the number of seconds that have\n";
   cout << "elapsed since some time in the past and\n";
   cout << "I will tell you how many minutes, hours,\n";
   cout << "days, months, and years have passed: ";
   cin  >> seconds;
   minutes = seconds / 60;
   hours = minutes / 60;
   days = hours / 24;
   years = days / 365;
   months = years * 12;

   cout << fixed << showpoint << setprecision(4) << left;
   cout << "Minutes: " << setw(6) << minutes << endl;
   cout << "Hours: "   << setw(6) << hours << endl;
   cout << "Days: "    << setw(6) << days << endl;
   cout << "Months: "  << setw(6) << months << endl;
   cout << "Years: "   << setw(6) << years << endl;
   return 0;
}
```

Find the Errors

27. Each of the following programs has some errors. Locate as many as you can.

A)

```
using namespace std;
int main()
{
   double number1, number2, sum;
```

```
    Cout << "Enter a number: ";
    Cin  << number1;
    Cout << "Enter another number: ";
    Cin  << number2;
    number1 + number2 = sum;
    Cout "The sum of the two numbers is " << sum
    return 0;
}
```

B)
```
#include <iostream>
using namespace std;
int main()
{
    int number1, number2;
    double quotient;
    cout << "Enter two numbers and I will divide\n";
    cout << "the first by the second for you.\n";
    cin  >> number1, number2;
    quotient = double<static_cast>(number1)/number2;
    cout << quotient
}
```

28. A)
```
#include <iostream>;
using namespace std;
int main()
{
    const int number1, number2, product;

    cout << "Enter two numbers and I will multiply\n";
    cout << "them for you.\n";
    cin  >> number1 >> number2;
    product = number1 * number2;
    cout << product
    return 0;
}
```

B)
```
#include <iostream>;
using namespace std;
main
{
    int number1, number2;

    cout << "Enter two numbers and I will multiply\n"
    cout << "them by 50 for you.\n"
    cin  >> number1 >> number2;
    number1 =* 50;
    number2 =* 50;
    return 0;
    cout << number1 << " " << number2;
}
```

29. A)
```
#include <iostream>;
using namespace std;
main
{
    double number, half;
```

```
    cout << "Enter a number and I will divide it\n"
    cout << "in half for you.\n"
    cin  >> number1;
    half =/ 2;
}
```

B)
```
#include <iostream>;
using namespace std;
int main()
{
    char name, go;

    cout << "Enter your name: ";
    cin.width(20);
    cin.getline >> name;
    cout << "Hi " << name << endl;
    cout "Press the ENTER key to end this program.";
    cin  >> go;
    return 0;
}
```

Soft Skills

Often programmers work in teams with other programmers to develop a piece of software. It is important that the team members be able to communicate clearly with one another.

30. Suppose you and a fellow student have been assigned to develop together the pizza cost program described in Problem 24. You have developed a pseudocode algorithm for the program and emailed it to your partner, but he does not understand how it works. Write a paragraph that you might email back clearly explaining how the algorithm works, what steps must be done, why they must be done in a particular order, and why the calculations you have specified in the pseudocode are the correct ones to use. Write your answer using full English sentences with correct spelling and grammar.

Programming Challenges

1. Miles per Gallon

Write a program that calculates a car's gas mileage. The program should ask the user to enter the number of gallons of gas the car can hold and the number of miles it can be driven on a full tank. It should then calculate and display the number of miles per gallon the car gets.



VideoNote
Solving the Stadium Seating Problem

2. Stadium Seating

There are three seating categories at a stadium. For a softball game, Class A seats cost $15, Class B seats cost $12, and Class C seats cost $9. Write a program that asks how many tickets for each class of seats were sold, then displays the amount of income generated from ticket sales. Format your dollar amount in a fixed-point notation with two decimal points and make sure the decimal point is always displayed.

3. Housing Costs

Write a program that asks the user to enter their *monthly* costs for each of the following housing related expenses:

- rent or mortgage payment
- utilities
- phones
- cable

The program should then display the total monthly cost of these expenses, and the total annual cost of these expenses.

4. How Much Insurance?

Many financial experts advise property owners to insure their homes or buildings for at least 80 percent of the amount it would cost to replace the structure. Write a program that asks the user to enter the replacement cost of a building and then displays the minimum amount of insurance that should be purchased for the property.

5. Batting Average

Write a program to find a baseball player's batting average. The program should ask the user to enter the number of times the player was at bat and the number of hits he got. It should then display his batting average to 4 decimal places.

6. Test Average

Write a program that asks for five test scores. The program should calculate the average test score and display it. The number displayed should be formatted in fixed-point notation, with one decimal point of precision.

7. Average Rainfall

Write a program that calculates the average monthly rainfall for three months. The program should ask the user to enter the name of each month, such as June or July, and the amount of rain (in inches) that fell that month. The program should display a message similar to the following:

```
The average monthly rainfall for June, July, and August was 6.72 inches.
```

8. Box Office

A movie theater only keeps a percentage of the revenue earned from ticket sales. The remainder goes to the distibutor. Write a program that calculates a theater's gross and net box office profit for a night. The program should ask for the name of the movie, and how many adult and child tickets were sold. (The price of an adult ticket is $6.00 and a child's ticket is $3.00.) It should display a report similar to the following:

```
Movie Name:                  "Wheels of Fury"
Adult Tickets Sold:             382
Child Tickets Sold:             127
Gross Box Office Profit:        $ 2673.00
Amount Paid to Distributor:   - $ 2138.40
Net Box Office Profit:          $  534.60
```

Assume the theater keeps 20 percent of the gross box office profit.

myCodeMate

9. How Many Widgets?

The Yukon Widget Company manufactures widgets that weigh 9.2 pounds each. Write a program that calculates how many widgets are stacked on a pallet, based on the total weight of the pallet. The program should ask the user how much the pallet weighs by itself and with the widgets stacked on it. It should then calculate and display the number of widgets stacked on the pallet.

10. How many Calories?

A bag of cookies holds 40 cookies. The calorie information on the bag claims that there are 10 "servings" in the bag and that a serving equals 300 calories. Write a program that asks the user to input how many cookies they actually ate and then reports how many total calories were consumed.

11. Celsius to Fahrenheit

Write a program that converts Celsius temperatures to Fahrenheit temperatures. The formula is

$$F = \frac{9}{5}C + 32$$

where F is the Fahrenheit temperature and C is the Celsius temperature. The program should prompt the user to input a Celsius temperature and should display the corresponding Farenheit temperature.

12. Currency

Write a program that will convert U.S. dollar amounts to Japanese yen and to euros, storing the conversion factors in the constant variables YEN_PER_DOLLAR and EUROS_PER_DOLLAR. To get the most up-to-date exchange rates, search the Internet using the term "currency exchange rate" or "currency converter". If you cannot find the most recent exchange rates, use the following:

1 Dollar = .952 Yen
1 Dollar = .7175 Euros

13. Monthly Sales Tax

A retail company must file a monthly sales tax report listing the sales for the month and the amount of sales tax collected. Write a program that asks for the month, the year, and the total amount collected at the cash register (that is, sales plus sales tax). Assume the state sales tax is 4 percent and the county sales tax is 2 percent.

If the total amount collected is known and the total sales tax is 6 percent, the amount of product sales may be calculated as

$$S = \frac{T}{1.06}$$

where S is the product sales and T is the total income (product sales plus sales tax).

The program should display a report similar to the following:

```
Month: March 2008
--------------------
Total Collected:     $ 26572.89
Sales:               $ 25068.76
County Sales Tax:    $   501.38
State Sales Tax:     $  1002.75
Total Sales Tax:     $  1504.13
```

myCodeMate

14. Property Tax

Madison County collects property taxes on the assessed value of property, which is 60 percent of its actual value. For example, if a house is valued at $158,000 its assessed value is $94,800. This is the amount the homeowner pays tax on. At last year's tax rate of $2.64 for each $100 of assessed value, the annual property tax for this house would be $2502.72. Write a program that asks the user to input the actual value of a piece of property and the current tax rate for each $100 of assessed value. The program should then calculate and report how much annual property tax the homeowner will be charged for this property.

15. Senior Citizen Property Tax

Madison County provides a $5000 homeowner exemption for its senior citizens. For example, if their house is valued at $158,000 its assessed value would be $94,800, as explained above. However they would only pay tax on $89,800. At last year's tax rate of $2.64 for each $100 of assessed value, their property tax would be $2370.72. In addition to the tax break, senior citizens are allowed to pay their property tax in 4 equal payments. The quarterly payment due on this property would be $592.68. Write a program that asks the user to input the actual value of a piece of property and the current tax rate for each $100 of assessed value. The program should then calculate and report how much annual property tax a senior homeowner will be charged for this property and what their quarterly tax bill will be.

16. Math Tutor

Write a program that can be used as a math tutor for a young student. The program should display two random numbers between 1 and 9 to be added, such as

```
  2
+ 1
```

After the student has entered an answer and pressed the [Enter] key, the program should display the correct answer so the student can see if his or her answer is correct.

17. Interest Earned

Assuming there are no deposits other than the original investment, the balance in a savings account after one year may be calculated as

$$\text{Amount} = \text{Principal} * \left(1 + \frac{\text{Rate}}{\text{T}}\right)^{\text{T}}$$

where Principal is the balance in the account, Rate is the annual interest rate, and T is the number of times the interest is compounded during a year. (e.g., T is 4 if the interest is compounded quarterly.)

Write a program that asks for the principal, the interest rate, and the number of times the interest is compounded. It should display a report similar to the following:

```
Interest Rate:            4.25%
Times Compounded:           12
Principal:           $ 1000.00
Interest:            $   43.33
Final balance:       $ 1043.33
```

18. Monthly Payments

The monthly payment on a loan may be calculated by the following formula:

$$\text{Payment} = \frac{\text{Rate} * (1 + \text{Rate})^{N}}{(1 + \text{Rate})^{N} - 1} * L$$

Rate is the monthly interest rate, which is the annual interest rate divided by 12. (A 12 percent annual interest would be 1 percent monthly interest.) *N* is the number of payments and `L` is the amount of the loan. Write a program that asks for these values and displays a report similar to the following:

```
Loan Amount:              $ 10000.00
Monthly Interest Rate:            1%
Number of Payments:               36
Monthly Payment:          $   332.14
Amount Paid Back:         $ 11957.15
Interest Paid:            $  1957.15
```

myCodeMate

19. Pizza Slices

Joe's Pizza Palace needs a program to calculate the number of slices a pizza of any size can be divided into. The program should perform the following steps:

A) Ask the user for the diameter of the pizza in inches.
B) Calculate the number of slices that may be taken from a pizza of that size if each slice has an area of 14.125 square inches.
C) Display a message telling the number of slices.

The number of square inches in the total pizza can be calculated with this formula:

$$\text{Area} = \pi r^2$$

where variable r is the radius of the pizza and π is the Greek letter PI. In your program make PI a named constant with the value 3.14. Display the number of slices as a whole number (i.e., with no decimals).

20. How Many Pizzas?

Modify the program you wrote in Programming Challenge 19 so that it reports the number of pizzas you need to buy for a party if each person attending is expected to eat an average of 4 slices. The program should ask the user for the number of people who will be at the party and for the diameter of the pizzas to be ordered. It should then calculate and display the number of pizzas to purchase. Because it is impossible to buy a part of a pizza, the number of required pizzas should be displayed as a whole number.

21. Angle Calculator

Write a program that asks the user for an angle, entered in radians. The program should then display the sine, cosine, and tangent of the angle. (Use the `sin`, `cos`, and `tan` library functions to determine these values.) The output should be displayed in fixed-point notation, rounded to four decimal places of precision.

22. Stock Transaction Program

Last month Joe purchased 100 shares of stock. Here are the details of the purchase:

- When Joe purchased the stock, he paid $32.87 per share.
- Joe paid his stock broker a commission that amounted to 2% of the amount he paid for the stock.

Two months later Joe sold the stock. Here are the details of the sale:

- He sold the stock for $33.92 per share.
- He paid his stock broker another commission that amounted to 2% of the amount he received for the stock.

Write a program that displays the following information:

- The amount of money Joe paid for the stock.
- The amount of commission Joe paid his broker when he bought the stock.
- The amount that Joe sold the stock for.
- The amount of commission Joe paid his broker when he sold the stock.
- The amount of profit or loss that Joe had after selling his stock and paying both broker commissions.

23. Using Files—Storing and Retrieving Numbers

For this assignment you will write two programs:

Program 1 Write a program that asks the user to enter five floating-point numbers. The program should create a file and save all five numbers to the file.

Program 2 Write a program that opens the file created by Program 1, reads the five numbers, and displays them. The program should also calculate and display the sum of the five numbers.

24. Using Files—Monthly Sales Tax Modification

Modify the program you wrote for Programming Challenge 13 (Monthly Sales Tax) so it writes its output to a file instead of to the screen. After the program has finished running, print the contents of the file to hand in.

25. Using Files—Average Rainfall Modification

Modify the program you wrote for Programming Challenge 7 (Average Rainfall) so it reads its input from a file instead of from the keyboard. Sample data to test your program can be found in the `rainfall.dat` file.

CHAPTER 4

Making Decisions

TOPICS

4.1 Relational Operators
4.2 The `if` Statement
4.3 The `if/else` Statement
4.4 The `if/else if` Statement
4.5 Menu-Driven Programs
4.6 Nested `if` Statements
4.7 Logical Operators
4.8 Validating User Input
4.9 More About Variable Definitions and Scope
4.10 Comparing Characters and Strings
4.11 The Conditional Operator
4.12 The `switch` Statement
4.13 Enumerated Data Types
4.14 Testing for File Open Errors
4.15 Focus on Testing and Debugging: *Validating Output Results*
4.16 Green Fields Landscaping Case Study—Part 2
4.17 Tying It All Together: *Fortune Teller*

4.1 Relational Operators

CONCEPT: **Relational operators allow you to compare numeric and `char` values and determine whether one is greater than, less than, equal to, or not equal to another.**

So far, the programs you have written follow this simple scheme:

- Gather input from the user.
- Perform one or more calculations.
- Display the results on the screen.

Computers are good at performing calculations, but they are also quite adept at comparing values to determine if one is greater than, less than, or equal to, the other. These types of operations are valuable for tasks such as examining sales figures, determining profit and loss, checking a number to ensure it is within an acceptable range, and validating the input given by a user.

Numeric data is compared in C++ by using relational operators. Characters can also be compared with these operators, because characters are considered numeric values in C++. Each relational operator determines whether a specific relationship exists between two values. For example, the greater-than operator (>) determines if a value is greater than another. The equality operator (==) determines if two values are equal. Table 4-1 lists all of C++'s relational operators.

Table 4-1 Relational Operators

Relational Operators	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to



NOTE: All the relational operators are binary operators with left-to-right associativity. Recall that associativity is the order in which an operator works with its operands.

All of the relational operators are binary. This means they use two operands. Here is an example of an expression using the greater-than operator:

```
x > y
```

This expression is called a *relational expression*. It is used to determine whether `x` is greater than `y`. The following expression determines whether `x` is less than `y`:

```
x < y
```

The Value of a Relationship

So, how are relational expressions used in a program? Remember, all expressions have a value. Relational expressions are *Boolean expressions*, which means their value can only be *true* or *false*. If `x` is greater than `y`, the expression `x > y` will be true, while the expression `y == x` will be false.

The == operator determines whether the operand on its left is equal to the operand on its right. If both operands have the same value, the expression is true. Assuming that `a` is 4, the following expression is true:

```
a == 4
```

But the following is false:

```
a == 2
```



WARNING! Notice the equality operator is two = symbols together. Don't confuse this operator with the assignment operator, which is one = symbol. The == operator determines if a variable is equal to another value, but the = operator assigns the value on the operator's right to the variable on its left. There will be more about this later in the chapter.

A couple of the relational operators actually test for two relationships. The >= operator determines whether the operand on its left is greater than *or* equal to the operand on the right. Assuming that `a` is 4, `b` is 6, and `c` is 4, both of the following expressions are true:

```
b >= a
a >= c
```

But the following is false:

```
a >= 5
```

The <= operator determines whether the operand on its left is less than *or* equal to the operand on its right. Once again, assuming that `a` is 4, `b` is 6, and `c` is 4, both of the following expressions are true:

```
a <= c
b <= 10
```

But the following is false:

```
b <= a
```

The last relational operator is `!=`, which is the not-equal operator. It determines whether the operand on its left is not equal to the operand on its right, which is the opposite of the == operator. As before, assuming `a` is 4, `b` is 6, and `c` is 4, both of the following expressions are true:

```
a != b
b != c
```

These expressions are true because `a` is *not* equal to `b` and `b` is *not* equal to `c`. But the following expression is false because `a` *is* equal to `c`:

```
a != c
```

Table 4-2 shows other relational expressions and their true or false values.

Table 4-2 Example Relational Expressions (Assume `x` is 10 and `y` is 7.)

Expression	Value
`x < y`	false, because `x` is not less than `y`.
`x > y`	true, because `x` is greater than `y`.
`x >= y`	true, because `x` is greater than or equal to `y`.
`x <= y`	false, because `x` is not less than or equal to `y`.
`y != x`	true, because `y` is not equal to `x`.

What Is Truth?

If a relational expression can evaluate to either true or false, how are those values represented internally in a program? How does a computer store *true* in memory? How does it store *false*?

As you saw in Program 2-16, those two abstract states are converted to numbers. This can be confusing, especially for new programmers, because in C++ zero is considered false and any non-zero value is considered true. The C++ keyword `false` is stored as 0 and the keyword `true` is stored as 1. And when a relational expression is false it evaluates to 0. However, when a relational expression is true it does not always evaluate to 1. Though it usually does, it can actually evaluate to any non-zero value.

To illustrate this more fully, look at Program 4-1.

Program 4-1

```
1 // This program displays the values of true and false states.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7    bool trueValue, falseValue;
8    int x = 5, y = 10;
9
10   trueValue = (x < y);
11   falseValue = (y == x);
12
13   cout << "True  is " << trueValue << endl;
14   cout << "False is " << falseValue << endl;
15   return 0;
16 }
```

Program Output

```
True  is 1
False is 0
```

Let's examine the statements containing the relational expressions a little closer:

```
trueValue = (x < y);
falseValue = (y == x);
```

These statements may seem odd because they are assigning the value of a comparison to a variable. In the first statement, the variable `trueValue` is being assigned the result of `x < y`. Because `x` is less than `y`, the expression is true, and the variable `trueValue` is assigned a non-zero value. In the second statement, the expression `y == x` is false, so the variable `falseValue` is set to 0.

When writing statements such as these, most programmers enclose the relational expression in parentheses, as shown above, to make it clearer.

Parentheses are not actually required, however, because even without them the relational operation is carried out before the assignment operation is performed. This occurs because relational operators have a higher precedence than the assignment operator. Likewise, arithmetic operators have a higher precedence than relational operators.

The statement

```
result = x < y - 8;
```

is equivalent to the statement

```
result = x < (y - 8);
```

In both cases, `y - 8` is evaluated first. Then this value is compared to x. Notice, however, how much clearer the second statement is. It is always a good idea to place parentheses around an arithmetic expression when its result will be used in a relational expression.

Table 4-3 shows examples of other statements that include relational expressions.

Table 4-3 Statements that Include Relational Expressions (Assume `x` is 10, `y` is 7, and `z` is an `int` or `bool`.)

Statement	Outcome
`z = x < y`	`z` is assigned 0 because `x` is not less than `y`.
`cout << (x > y);`	Displays 1 because `x` is greater than `y`.
`z = (x >= y);`	`z` is assigned 1 because `x` is greater than or equal to `y`.
`cout << (x <= y);`	Displays 0 because `x` is not less than or equal to `y`.
`z = (y != x);`	`z` is assigned 1 because `y` is not equal to `x`.
`cout << (x == (y + 3));`	Displays 1 because `x` is equal to `y + 3`.

Relational operators also have a precedence order among themselves. The two operators that test for equality or lack of equality (== and `!=`) have the same precedence as each other. The four other relational operators, which test relative size, have the same precedence as each other. These four relative relational operators have a higher precedence than the two equality relational operators. Table 4-4 shows the precedence of relational operators.

Table 4-4 Precedence of Relational Operators (Highest to Lowest)

`>  >=  <  <=`
`==  !=`

Here is an example of how this is applied. If `a = 9`, `b = 24`, and `c = 0`, the following statement would cause a 1 to be printed.

```
cout << (c == a > b);
```

Because of the relative precedence of the operators in this expression, `a > b` would be evaluated first. Since 9 is not greater than 24, it would evaluate to false, or 0. Then `c == 0` would be evaluated. Since `c` does equal 0, this would evaluate to true, or 1. So a 1 would be inserted into the output stream and printed.

In this chapter's remaining sections you will see how to get the most from relational expressions by using them in statements that take action based on the results of the comparison.



Checkpoint

4.1 Assuming `x` is 5, `y` is 6, and `z` is 8, indicate whether each of the following relational expressions is true or false:

A) `x == 5`
B) `7 <= (x + 2)`
C) `z > 4`
D) `(2 + x) != y`
E) `z != 4`
F) `x >= 0`
G) `x <= (y * 2)`

4.2 Indicate whether each of the following statements about relational expressions is correct or incorrect.

A) `x <= y` is the same as `y > x`.
B) `x != y` is the same as `y >= x`.
C) `x >= y` is the same as `y <= x`.

4.3 Answer the following questions with a yes or no.

A) If it is true that `x > y` and it is also true that `x < z`, does that mean `y < z` is true?
B) If it is true that `x >= y` and it is also true that `z == x`, does that mean that `z == y` is true?
C) If it is true that `x != y` and it is also true that `x != z`, does that mean that `z != y` is true?

4.4 What will the following program segment display?

```
int a = 0, b = 2, x = 4, y = 0;

cout << (a == b) << endl;
cout << (a != y) << endl;
cout << (b <= x) << endl;
cout << (y > a)  << endl;
```

4.2 The `if` Statement

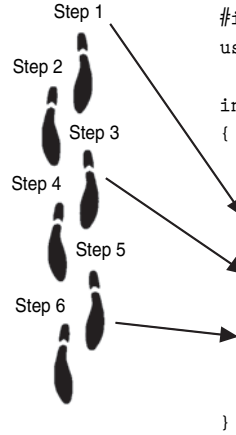
CONCEPT: The `if` statement can cause other statements to execute only under certain conditions.



VideoNote
Using an `if` Statement

You might think of the statements in a procedural program as individual steps taken as you are walking down a road. To reach the destination, you must start at the beginning and take each step, one after the other, until you reach the destination. The programs you have written so far are like a "path" of execution for the program to follow.

Figure 4-1

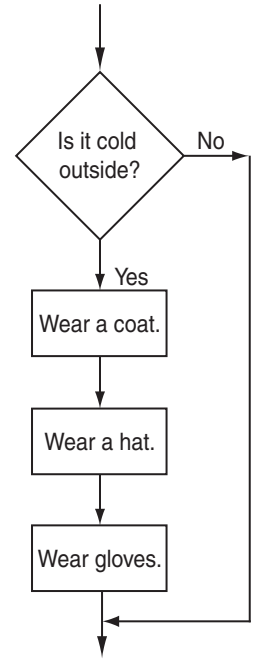



```
// A program to calculate the area of a rectangle

#include <iostream>
using namespace std;

int main()
{
   double length, width, area;

   cout << "Enter the length of the rectangle: ";
   cin  >> length;
   cout << "Enter the width of the rectangle: ";
   cin  >> width;
   area = length * width;
   cout << "The area is: " << area << endl;
   return 0;
}
```

The type of code in Figure 4-1 is called a *sequence structure* because the statements are executed in sequence, one after another, without branching off in another direction. Programs often need more than one path of execution, however. Many algorithms require a program to execute some statements only under certain circumstances. This can be accomplished with a *decision structure*.

In a decision structure's simplest form a specific action, or set of actions, is taken only when a specific condition exists. If the condition does not exist, the actions are not performed. The flowchart in Figure 4-2 shows the logic of a decision structure. The diamond symbol represents a yes/no question or a true/false condition. If the answer to the question is yes (or if the condition is true), the program flow follows one path which leads to the actions being performed. If the answer to the question is no (or the condition is false), the program flow follows another path which skips the actions.

Figure 4-2



In the flowchart, the actions "Wear a coat", "Wear a hat", and "Wear gloves" are performed only when it is cold outside. If it is not cold outside, these actions are skipped. The actions are *conditionally executed* because they are performed only when a certain condition (cold outside) exists.

We perform mental tests like these every day. Here are some other examples:

If the car is low on gas, stop at a service station and get gas.
If it's raining outside, go inside.
If you're hungry, get something to eat.

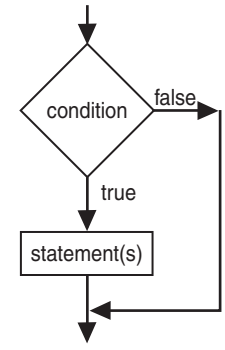
The most common way to code a decision structure in C++ is with the `if` statement. Figure 4-3 shows the general format of the `if` statement and a flowchart visually depicting how it works.

Figure 4-3

```
if (condition)
{
     statement 1;
     statement 2;
        .
        .
     statement n;
}
```

or

```
if (condition)  {
     statement 1;
     statement 2;
        .
        .
     statement n;
}
```



Notice that the statements inside the body of the `if` construct are contained within a set of curly braces. The opening brace must be located after the `if` *condition* and before the first statement in the body. However, while following this requirement, different programmers choose different places to locate it. The two most common placements are shown in Figure 4-3. This book uses the form shown on the left. Your instructor will tell you what form he or she wants you to use.

Program 4-2 illustrates the use of an `if` statement. The user enters three test scores and the program calculates their average. If the average equals 100, the program congratulates the user on earning a perfect score.

Program 4-2

```
1  // This program correctly averages 3 test scores.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  int main()
7  {
8     int score1, score2, score3;
9     double average;
10
11    // Get the three test scores
12    cout << "Enter 3 test scores and I will average them: ";
13    cin  >> score1 >> score2 >> score3;
14
15    // Calculate and display the average score
16    average = (score1 + score2 + score3) / 3.0;
17    cout << fixed << showpoint << setprecision(1);
18    cout << "Your average is " << average << endl;
19
20    // If the average equals 100, congratulate the user
21    if (average == 100)
22    {  cout << "Congratulations! ";
23       cout << "That's a perfect score!\n";
24    }
25    return 0;
26 }
```

(program continues)

Program 4-2 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter 3 test scores and I will average them: 80 90 70[Enter]
Your average is 80.0
```

Program Output with Other Example Input Shown in Bold

```
Enter 3 test scores and I will average them: 100 100 100[Enter]
Your average is 100.0
Congratulations! That's a perfect score!
```

Let's look more closely at lines 21–24 of Program 4-2, which cause the congratulatory message to be printed.

```
if (average == 100)
{   cout << "Congratulations! ";
    cout << "That's a perfect score!\n";
}
```

There are four important things to notice. First, the word `if`, which begins the statement, is a C++ key word and must be written in lowercase. Second, the condition to be tested (`average == 100`) must be enclosed inside parentheses. Third, there is *no* semi-colon after the test condition, even though there *is* a semi-colon after each action associated with the `if` construct. We will explain why shortly. And finally, the block of statements to be conditionally executed is surrounded by curly braces. This is required whenever two or more actions are associated with an `if` statement.

If the block of statements to be conditionally executed contains only one statement, the braces can be omitted. For example, in Program 4-2 if the two `cout` statements were combined into one statement, they could be written as shown here.

```
if (average == 100)
   cout << "Congratulations! That's a perfect score!\n";
```

However, some instructors prefer that you *always* place braces around a conditionally executed block, even when it consists of only one statement.

Table 4-5 shows other examples of `if` statements and their outcomes.

Table 4-5 Example `if` Statements

Statements	Outcome

`if (hours > 40)`
`{  overTime = true;`
`   payRate *= 2;`
`}`
	Assigns `true` to Boolean variable `overTime` and doubles `payRate` only when `hours` is greater than 40. Because there is more than one statement in the conditionally executed block, braces `{}` are required.

`if (temperature > 32)` `   freezing = false;`	Assigns `false` to Boolean variable `freezing` only when `temperature` is greater than 32. Because there is only one statement in the conditionally executed block, braces `{}` are optional.

Programming Style and the `if` Statement

Even though `if` statements usually span more than one line, they are technically one long statement. For instance, the following `if` statements are identical except in style:

```
if (a >= 100)
   cout << "The number is out of range.\n";

if (a >= 100) cout << "The number is out of range.\n";
```

The first of these two `if` statements is considered to be better style because it is easier to read. By indenting the conditionally executed statement or block of statements you are causing it to stand out visually. This is so you can tell at a glance what part of the program the `if` statement executes. This is a standard way of writing `if` statements and is the method you should use. Here are two important style rules you should adopt for writing `if` statements:

- The conditionally executed statement(s) should begin on the line after the `if` statement.
- The conditionally executed statement(s) should be indented one "level" from the `if` statement.



NOTE: In most editors, each time you press the tab key, you are indenting one level.

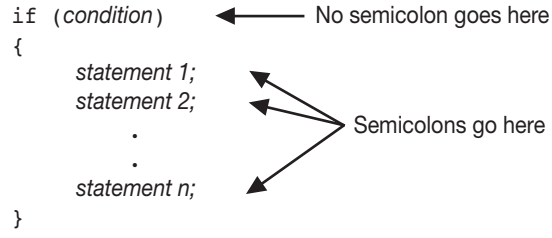
Three Common Errors to Watch Out For

When writing `if` statements, there are three common errors you must watch out for.

1. Misplaced semicolons
2. Missing braces
3. Confusing = with ==

Be Careful with Semicolons

Semicolons do not mark the end of a line, but the end of a complete C++ statement. The `if` statement isn't complete without the conditionally executed statement that comes after it. So, you must not put a semicolon after the `if (`*`condition`*`)` portion of an `if` statement.



If you inadvertently put a semicolon after the `if` part, the compiler will assume you are placing a null statement there. The *null statement* is an empty statement that does nothing. This will prematurely terminate the `if` statement, which disconnects it from the block of

statements that follows it. These statements will then always execute. For example, notice what would have happened in Program 4-2 if the `if` statement had been prematurely terminated with a semicolon, as shown here.

```
if (average == 100);     // Error. The semicolon terminates
{                        // the if statement prematurely.
    cout << "Congratulations! ";
    cout << "That's a perfect score!\n";
}
```

Output of Revised Program 4-2 with Example Input Shown in Bold

```
Enter 3 test scores and I will average them: 80 90 70[Enter]
Your average is 80.0
Congratulations! That's a perfect score!
```

Because the `if` statement ends when the premature semicolon is encountered, the `cout` statements inside the braces are considered to be separate statements following the `if`, rather than statements belonging to the `if`. Therefore, they always execute, regardless of whether `average` equals 100 or not. This erroneous version of the program can be found on the student CD as Program 4-2B.



NOTE: Indentation and spacing are for the human reader of a program, not the computer. Even though the `cout` statements following the `if` statement in this example are indented, the semicolon still terminates the `if` statement.

Don't Forget the Braces

If you intend to conditionally execute a block of statements, rather than just one statement, with an `if` statement, don't forget the braces. Without a set of braces, the `if` statement only executes the very next statement. Any following statements are considered to be outside the `if` statement and will always be executed, even when the `if` condition is false. For example, notice what would have happened in the original Program 4-2 if the braces enclosing the two `cout` statements to be conditionally executed had been omitted.

```
if (average == 100)
    cout << "Congratulations! ";          // There are no braces.
    cout << "That's a perfect score!\n";  // This is outside the if.
```

Output of Program 4-2 Revised a Second Time with Example Input Shown in Bold

```
Enter 3 test scores and I will average them: 80 90 70[Enter]
Your average is 80.0
That's a perfect score!
```

With no braces around the set of statement to be conditionally executed, only the first of these statements is considered to belong to the `if` statement. Because the condition in our test case `(average == 100)` was false, the `Congratulations!` message was skipped. However the `cout` statement that prints `That's a perfect score!` was executed, as it would be every time, regardless of whether `average` equals 100 or not. This erroneous version of the program can be found on the student CD as Program 4-2C.

Not All Operators Are "Equal"

Earlier you saw a warning not to confuse the equality operator (==) with the assignment operator (=), as in the following statement:

```
if (x = 2)        // Caution here!
    cout << "It is True!";
```

This statement does not determine whether x is equal to 2; instead it assigns x the value 2! Furthermore, the cout statement will *always* be executed because the expression x = 2 evaluates to 2, which C++ considers true.

This occurs because the value of an assignment expression is the value being assigned to the variable on the left side of the = operator. Therefore the value of the expression x = 2 is 2. Earlier you learned that C++ stores the value true as 1. But it actually considers all non-zero values, not just 1, to be true. Thus 2 represents a true condition.

Let's examine this more closely by looking at yet another variation of the original Program 4-2. This time notice what would have happened if the equal-to relational operator in the if condition had been replaced by the assignment operator, as shown here.

```
if (average = 100)        // Error. This assigns 100 to average.
{
    cout << "Congratulations! ";
    cout << "That's a perfect score!\n";
}
```

Output of Program 4-2 Revised a Third Time with Example Input Shown in Bold

```
Enter 3 test scores and I will average them: 80 90 70[Enter]
Your average is 80.0
Congratulations! That's a perfect score!
```

Rather than being compared to 100, average is assigned the value 100 in the if statement. This causes the if test to evaluate to 100, which is considered true. Therefore the two cout statements will execute every time, regardless of what test scores are entered by the user. This erroneous version of the program can be found on the student CD as Program 4-2D.

More About Truth

Now that you've gotten your feet wet with relational expressions and if statements, let's look further at the subject of truth. You have seen that a relational expression has the value 1 when it is true and 0 when false. You have also seen that while 0 is considered false, all values other than 0 are considered true. This means that any value, even a negative number, represents true as long as it is not 0.

Just as in real life, truth is a complicated thing. Here is a summary of the rules you have seen so far:

- When a relational expression is true, it has a non-zero value, which in most cases is represented by the value 1.
- When a relational expression is false, it has the value 0.
- An expression that has the value 0 is considered false by the if statement. This includes the bool value false, which is equivalent to 0.
- An expression that has *any* value other than 0 is considered true. This includes the bool value true, which is equivalent to 1.

The fact that the if statement considers any nonzero value as true opens many possibilities. Relational expressions are not the only conditions that may be tested. For example, the following is a legal if statement in C++:

```
if (value)
   cout << "It is True!";
```

This if statement is not testing a relational expression. It is testing the contents of a variable. If the variable, value, contains any number other than 0, the message "It is True!" will be displayed. If value is set to 0, however, the cout statement will be skipped. Here is another example:

```
if (x + y)
   cout << "It is True!";
```

In this statement the sum of x and y is tested. If the sum is 0, the expression is considered false; otherwise it is considered true. You may also use the return value of a function call as a conditional expression. Here is an example that uses the pow function:

```
if (pow(a, b))
   cout << "It is True!";
```

This if statement uses the pow function to raise a to the power of b. If the result is anything other than 0, the cout statement will be executed.

Flags

A *flag* is a variable that signals whether or not some condition currently exists in a program. Because bool variables hold the values true and false, they are the perfect type of variables to use for flags. When the flag variable is set to true, it means the condition does exist. When the flag variable is set to false, it means that the condition does not exist, at least not yet.

Suppose we wanted to change Program 4-2 to find out if *any* of the scores read in equals 100, rather than finding out if the average equals 100. We could use a Boolean variable, scored100, initialized to false.

```
bool scored100 = false;   // No score of 100 found yet
```

Then, after reading in the three scores, we could test them, like this:

```
if (score1 == 100)
   scored100 = true;
if (score2 == 100)
   scored100 = true;
if (score3 == 100)
   scored100 = true;
```

Finally, we could replace the lines that test for a perfect average, with the following code.

```
if (scored100)
   cout << "You got 100% on at least one of the tests.\n";
```

Comparing Floating-Point Numbers

Testing floating-point numbers for equality can sometimes give erroneous results. Because of a lack of precision or round-off errors, a number that should be mathematically equal to

another might not be. In Program 4-3, 6 is multiplied by 0.666667, a decimal version of 2/3. Of course, 6 times 2/3 is 4. The program, however, disagrees.

Program 4-3

```
1  // This program demonstrates how a lack of precision in
2  // floating-point numbers can make equality comparisons unreliable.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     double result = .666667 * 6.0;
9
10    // 2/3 of 6 should be 4 and, if you print result, 4 is displayed.
11    cout << "result = " << result << endl;
12
13    // However, internally result is NOT precisely equal to 4.
14    if (result == 4.0)
15       cout << "result DOES equal 4!" << endl;
16    else
17       cout << "result DOES NOT equal 4!" << endl;
18
19    return 0;
20 }
```

Program Output

```
result = 4
result DOES NOT equal 4!
```

Typically, the value in `result` will be a number just short of 4, like 3.999996. To prevent errors like this, it is wise to stick with greater-than and less-than comparisons when using floating-point numbers. For example, instead of testing if the result equals 4.0, you could test to see if it is very close to 4.0. Program 4-4 demonstrates this technique.

Program 4-4

```
1  // This program demonstrates how to safely test a floating-point number
2  // to see if it is, for all practical purposes, equal to some value.
3  #include <iostream>
4  #include <cmath>
5  using namespace std;
6
7  int main()
8  {
9     double result = .666667 * 6.0;
10
11    // 2/3 of 6 should be 4 and, if you print result, 4 is displayed.
12    cout << "result = " << result << endl;
13
```

(program continues)

Program 4-4 *(continued)*

```
14    // However, internally result is NOT precisely equal to 4.
15    // So test to see if it is "close" to 4.
16    if (abs(result - 4.0 < .0001))
17       cout << "result DOES equal 4!" << endl;
18    else
19       cout << "result DOES NOT equal 4!" << endl;
20
21    return 0;
22 }
```

Program Output

```
result = 4
result DOES equal 4!
```

Line 16 of the program uses the `abs` function introduced in Chapter 2. Recall that it returns the absolute value of the argument. By using it, we ensure that the test condition will be true if the difference between `result` and 4.0 is less than .0001, regardless of which one is larger.



Checkpoint

4.5 True or False: Both of the following `if` statements perform the same operation.

```
if (sales > 10000)
   commissionRate = 0.15;

if (sales > 10000) commissionRate = 0.15;
```

4.6 True or false: Both of the following `if` statements perform the same operation.

```
if (calls == 20)
   rate *= 0.5;

if (calls = 20)
   rate *= 0.5;
```

4.7 Although the following code segments are syntactically correct, each contains an error. Locate the error and indicate what is wrong.

A)
```
hours = 12;
if (hours > 40);
   cout << hours << " hours qualifies for over-time.\n";
```

B)
```
interestRate = .05;
if (interestRate = .07)
   cout << "This account is earning the maximum rate.\n";
```

C)
```
interestRate = .05;
if (interestRate > .07)
   cout << "This account earns a $10 bonus.\n";
   balance += 10.0;
```

4.8 Write an `if` statement that assigns 0 to `x` if `y` is equal to 20.

4.9 Write an `if` statement that multiplies `payRate` by 1.5 if `hours` is greater than 40.

4.10 Write an `if` statement that assigns .20 to `commission` if `sales` is greater than or equal to 10,000.00.

4.11 Write an `if` statement that sets `fees` to 50 if flag variable `max` is set to true.

4.3 The `if/else` Statement

CONCEPT: The **`if/else`** statement will execute one set of statements when the **`if`** condition is true, and another set when the condition is false.

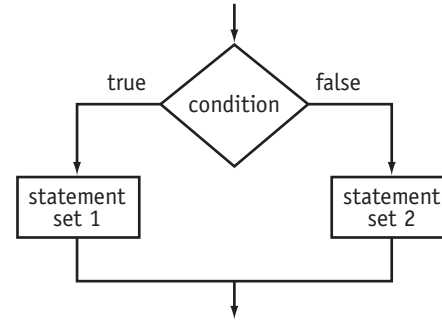


VideoNote
Using an `if/else` Statement

The `if/else` statement is an expansion of the `if` statement. Figure 4-4 shows the general format of this statement and a flowchart visually depicting how it works.

Figure 4-4

```
if (condition)
{
      statement set 1;
}
else
{
      statement set 2;
}
```



As with the `if` statement, a condition is tested. If the condition is true, a block containing one or more statements is executed. If the condition is false, however, a different group of statements is executed. Program 4-5 uses the `if/else` statement along with the modulus operator to determine if a number is odd or even.

Program 4-5

```
1  // This program uses the modulus operator to determine
2  // if a number is odd or even. If the number is evenly divisible
3  // by 2, it is an even number. A remainder indicates it is odd.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     int number;
10
11    cout << "Enter an integer and I will tell you if it\n";
12    cout << "is odd or even. ";
13    cin  >> number;
14
```

(program continues)

Program 4-5 *(continued)*

```
15    if (number % 2 == 0)
16        cout << number << " is even.\n";
17    else
18        cout << number << " is odd.\n";
19    return 0;
20 }
```

Program Output with Example Input Shown in Bold

```
Enter an integer and I will tell you if it
is odd or even. 17[Enter]
17 is odd.
```

The `else` part at the end of the `if` statement specifies one or more statements that are to be executed when the condition is false. When `number % 2` does not equal 0, a message is printed indicating the number is odd. Note that the program will only take one of the two paths in the `if/else` statement. If you think of the statements in a computer program as steps taken down a road, consider the `if/else` statement as a fork in the road. It causes program execution to follow one of two mutually exclusive paths.

Notice the programming style used to construct the `if/else` statement. The word `else` is at the same level of indention as `if`. The statements whose execution are controlled by the `if` and by the `else` are indented one level. This makes the two paths of execution that may be followed visually clear to anyone reading the code.

As with the `if` part, if you don't use braces the `else` part controls a single statement. If you wish to execute more than one statement with the `else` part, place these statements inside a set of braces. Program 4-6 illustrates this. It also illustrates a way to handle a classic programming problem: division by zero.

Division by zero is mathematically impossible to perform and it normally causes a program to crash. This means the program will prematurely stop running, sometimes with an error message. Program 4-6 shows a way to test the value of a divisor before the division takes place. On line 15 the value of `num2` is tested. If the user enters anything other than zero, the lines controlled by the `if` are executed, allowing the division to be performed and the result to be displayed. But if the user enters a zero for `num2`, the lines controlled by the `else` are executed instead, causing an error message to be displayed.

Program 4-6

```
1  // This program makes sure that the divisor is not
2  // equal to 0 before it performs a divide operation.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     double num1, num2, quotient;
9
```

(program continues)

Program 4-6 *(continued)*

```
10    // Get the two numbers
11    cout << "Enter two numbers: ";
12    cin  >> num1 >> num2;
13
14    // If num2 is not zero, perform the division.
15    if (num2 != 0)
16    {
17       quotient = num1 / num2;
18       cout << "The quotient of " << num1 << " divided by "
19            << num2 << " is " << quotient << ".\n";
20    }
21    else
22    {
23       cout << "Division by zero is not possible.\n"
24            << "Please run the program again and enter "
25            << "a number other than zero.\n";
26    }
27    return 0;
28 }
```

Program Output with Example Input Shown in Bold

```
Enter two numbers: 10 0[Enter]
Division by zero is not possible.
Please run the program again and enter a number other than zero.
```



Checkpoint

4.12 True or false: The following `if/else` statements cause the same output to display.

A)
```
if (x > y)
   cout << "x is the greater.\n";
else
   cout << "x is not the greater.\n";
```

B)
```
if (y <= x)
   cout << "x is not the greater.\n";
else
   cout << "x is the greater.\n";
```

4.13 Write an `if/else` statement that assigns 1 to `x` if `y` is equal to 100. Otherwise it should assign 0 to `x`.

4.14 Write an `if/else` statement that assigns 0.10 to `commission` unless `sales` is greater than or equal to 50,000.00, in which case it assigns 0.20 to `commission`.

4.15 Complete the following program skeleton so it computes the correct sales tax. If the customer is an in-state resident, `taxRate` should be set to .05. If the customer is an out-of-state resident, `taxRate` should be set to 0.

```
#include <iostream>
using namespace std;
```

```
int main()
{
   double taxRate, saleAmount;
   char residence;

   cout << "Enter the amount of the sale: ";
   cin  >> saleAmount;
   cout << "Enter I for in-state residence or O for out-of-\n";
   cout << "state: ";
   cin.get(residence);

   // Write code here that assigns 0 to taxRate if char
   // variable residence is set to the letter 'O' or .05 to
   // taxRate if residence is set to 'I'.

   saleAmount += saleAmount * taxRate;
   cout << "The total is " << saleAmount;
   return 0;
}
```

4.16 Will the `if/else` statement shown below function exactly the same as the two separate `if` statements?

```
if (x < y)              if (x < y)
   cout << 1;              cout << 1;
if (x > y)              else
   cout << 2;              cout << 2;
```

4.4 The `if/else if` Statement

CONCEPT: The `if/else if` statement is a chain of `if` statements. They perform their tests, one after the other, until one of them is found to be true.



VideoNote
Using an `if/else if` Statement

We make certain mental decisions by using sets of different but related rules. For example, we might decide the type of coat or jacket to wear by consulting the following rules:

if it is very cold, wear a heavy coat,
else, if it is chilly, wear a light jacket,
else, if it is windy, wear a windbreaker,
else, if it is hot, wear no jacket.

The purpose of these rules is to determine which type of outer garment to wear. If it is cold, the first rule dictates that a heavy coat must be worn. All the other rules are then ignored. If the first rule doesn't apply, however (if it isn't cold), then the second rule is consulted. If that rule doesn't apply, the third rule is consulted, and so forth.

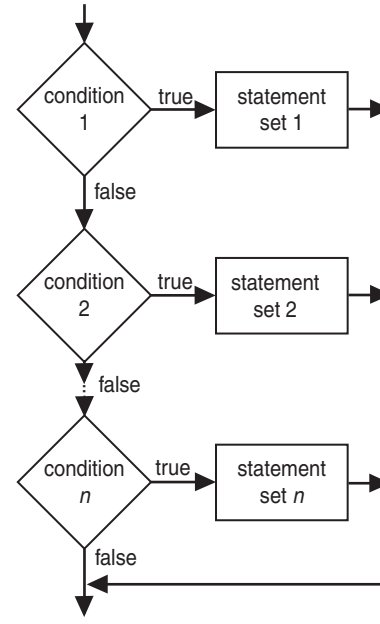
The way these rules are connected is very important. If they were consulted individually, we might go out of the house wearing the wrong jacket or, possibly, more than one jacket. For instance, if it is windy, the third rule says to wear a windbreaker. What if it is both windy and very cold? Will we wear a windbreaker? A heavy coat? Both? Because of the order that the

rules are consulted in, the first rule will determine that a heavy coat is needed. The third rule will not be consulted, and we will go outside wearing the most appropriate garment.

This type of decision making is also very common in programming. In C++ it can be accomplished through the `if/else if` statement. Figure 4-5 shows its format and a flowchart visually depicting how it works.

Figure 4-5

```
if (condition 1)
{
      statement set 1;
}
else if (condition 2)
{
      statement set 2;
}
         .
         .

else if (condition n)
{
      statement set n;
}
```



This construction is like a chain of `if/else` statements. The `else` part of one statement is linked to the `if` part of another. When put together this way, the chain of `if/else`s becomes one long statement. Program 4-7 shows an example. The user is asked to enter a numeric test score and the program displays the letter grade earned.

Program 4-7

```
1  // This program uses an if/else if statement to assign a
2  // letter grade (A, B, C, D, or F) to a numeric test score.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int testScore;    // Holds a numeric test score
9     char grade;       // Holds a letter grade
10
11    // Get the numeric score
12    cout << "Enter your numeric test score and I will\n";
13    cout << "tell you the letter grade you earned: ";
14    cin  >> testScore;
15
```

(program continues)

Program 4-7 *(continued)*

```
16    // Determine the letter grade
17    if (testScore < 60)
18       grade = 'F';
19    else if (testScore < 70)
20       grade = 'D';
21    else if (testScore < 80)
22       grade = 'C';
23    else if (testScore < 90)
24       grade = 'B';
25    else if (testScore <= 100)
26       grade = 'A';
27
28    // Display the letter grade
29    cout << "Your grade is " << grade << ".\n";
30    return 0;
31 }
```

Program Output with Example Input Shown in Bold

```
Enter your numeric test score and I will
tell you the letter grade you earned: 88[Enter]
Your grade is B.
```

As with other forms of the `if` statement, braces are required in an `if/else if` whenever there is more than one statement in a conditionally executed block. Otherwise they are optional. Because each of the conditionally executed blocks of code in Program 4-7 contains only one statement, braces were not used.

The `if/else if` statement has a number of notable characteristics. Let's analyze how it works in Program 4-7. First, the relational expression `testScore < 60` is tested on line 17.

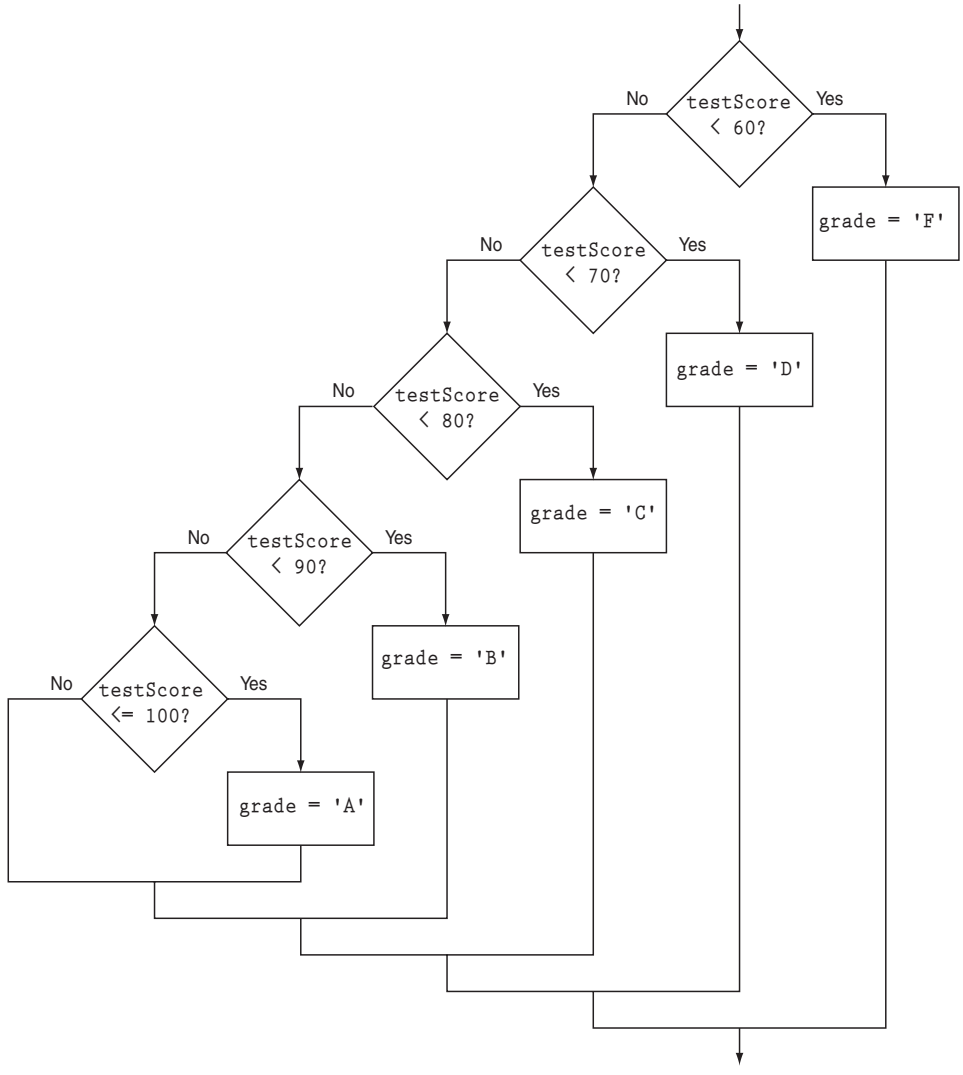
```
if (testScore < 60)
   grade = 'F';
```

If `testScore` is less than 60, the letter '`F`' is assigned to `grade` and the rest of the linked `if` statements are skipped. If `testScore` is not less than 60, the `else` part takes over and causes the next `if` condition to be tested on line 19.

```
else if (testScore < 70)
   grade = 'D';
```

The first `if` statement filtered out all of the grades less than 60, so when this next `if` statement executes, `testScore` will have a value of 60 or greater. If `testScore` is less than 70, the letter 'D' is assigned to `grade` and the rest of the `if/else if` statement is ignored. This chain of events continues until one of the conditional expressions is found true or the end of the statement is encountered. In either case, the program resumes at the statement immediately following the `if/else if` statement. This is the `cout` statement on line 29 that prints the grade. Figure 4-6 shows the paths that may be taken by the `if/else if` statement.

Figure 4-6



Each `if` condition in the structure depends on all the `if` conditions before it being false. The statements following a particular `else if` are executed when the conditional expression following the `else if` is true and all previous conditional expressions are false. To demonstrate how this interconnection works, let's look at Program 4-8, which uses independent `if` statements instead of an `if/else if` statement.

Program 4-8

```
1 // This program illustrates a bug that occurs when independent if/else
2 // statements are used to assign a letter grade to a numeric test score.
3 #include <iostream>
4 using namespace std;
5
```

(program continues)

Program 4-8 *(continued)*

```
6  int main()
7  {
8     int testScore;     // Holds a numeric test score
9     char grade;        // Holds a letter grade
10
11    // Get the numeric score
12    cout << "Enter your test score and I will tell you\n";
13    cout << "the letter grade you earned: ";
14    cin  >> testScore;
15
16    // Determine the letter grade. What grade will be assigned?
17    if (testScore < 60)
18       grade = 'F';
19    if (testScore < 70)
20       grade = 'D';
21    if (testScore < 80)
22       grade = 'C';
23    if (testScore < 90)
24       grade = 'B';
25    if (testScore <= 100)
26       grade = 'A';
27
28    // Display the letter grade
29    cout << "Your grade is " << grade << ".\n";
30    return 0;
31 }
```

Program Output with Example Input Shown in Bold

```
Enter your numeric test score and I will tell you
the letter grade you earned: 40[Enter]
Your grade is A.
```

In Program 4-8, all the `if` statements execute because they are individual statements. In the example output, `testScore` is assigned the value 40, yet the student receives an A. Here is what happens. Because the student's score is less than 60, the first `if` statement causes `'F'` to be assigned to `grade`. However, because the next `if` statement is not connected to the first through an `else`, it executes as well. Since `testScore` is also less than 70, it causes `'D'` to be assigned to `grade`, replacing the `'F'` that was previously stored there. This continues until all the `if` statements have executed. The last one will cause `'A'` to be assigned to `grade`. (Most students prefer this method since `'A'` is the only grade it gives out!)

Using a Trailing `else`

A trailing `else`, placed at the end of an `if/else if` statement, provides a default set of actions when none of the `if` expressions are true. It is often used to catch errors. This feature would be helpful, for example, in Program 4-7. What happens in the current version of that program if the user enters a test score greater than 100? The `if/else if` statement handles all scores through 100, but none greater. If the user enters 104, for example, the program does not give any letter grade because there is no code to handle a score greater

than 100. Assuming that 100 is the highest score, we can fix the program by placing an `else` at the end of the `if/else if` statement. This is shown in Program 4-9.

Program 4-9

```
// This program uses an if/else if statement to assign a
// letter grade (A, B, C, D, or F) to a numeric test score.
// A trailing else has been added to catch test scores > 100.
#include <iostream>
using namespace std;

int main()
{
   int testScore;          // Holds a numeric test score
   char grade;             // Holds a letter grade
   bool goodScore = true;  // Is the score valid?

   // Get the numeric score
   cout << "Enter your numeric test score and I will\n";
   cout << "tell you the letter grade you earned: ";
   cin  >> testScore;

   // Determine the letter grade
   if (testScore < 60)
      grade = 'F';
   else if (testScore < 70)
      grade = 'D';
   else if (testScore < 80)
      grade = 'C';
   else if (testScore < 90)
      grade = 'B';
   else if (testScore <= 100)
      grade = 'A';
   else
      goodScore = false;   // The score was greater than 100

   // If the score is valid, display the corresponding letter grade;
   // otherwise, display an error message
   if (goodScore)
      cout << "Your grade is " << grade << endl;
   else
      cout << "We do not give scores higher than 100.\n";

   return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter your numeric test score and I will
tell you the letter grade you earned: 104 [Enter]
We do not give scores higher than 100.
```

The trailing `else` catches any value that "falls through the cracks." It provides a default response when none of the `if`s find a true condition.

4.5 Menu-Driven Programs

CONCEPT: **A *menu* is a set of choices presented to the user. A *menu-driven program* allows the user to determine the course of action by selecting it from the menu.**

A menu is a screen displaying a set of choices the user selects from. For example, a program that keeps a mailing list might give you the following menu:

1. Add a name to the list.
2. Remove a name from the list.
3. Change a name in the list.
4. Print the list.
5. Quit the program.

The user selects one of the operations by entering its number. Entering 4, for example, causes the mailing list to be printed, and entering 5 causes the program to end. The `if/else if` structure can be used to set up such a menu. After the user enters a number, it compares the number to the available selections and executes the statements that perform that operation.

Program 4-10 calculates the charges for membership in a health club. The club has three membership packages to choose from: standard adult membership, child membership, and senior citizen membership. The program presents a menu that allows the user to choose the desired package and then calculates the cost of the membership.

Program 4-10

```
1  // This menu-driven program uses an if/else statement to carry
2  // out the correct set of actions based on the user's menu choice.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     // Constants for membership rates
10    const double ADULT_RATE  = 40.0;
11    const double CHILD_RATE  = 20.0;
12    const double SENIOR_RATE = 30.0;
13
14    int choice;           // Menu choice
15    int months;           // Number of months
16    double charges;       // Monthly charges
17
18    // Display the menu and get the user's choice
19    cout << "   Health Club Membership Menu\n\n";
20    cout << "1. Standard Adult Membership\n";
21    cout << "2. Child Membership\n";
22    cout << "3. Senior Citizen Membership\n";
23    cout << "4. Quit the Program\n\n";
```

(program continues)

Program 4-10 *(continued)*

```
24    cout << "Enter your choice: ";
25    cin  >> choice;
26
27    // Set the numeric output formatting
28    cout << fixed << showpoint << setprecision(2);
29
30    // Use the menu selection to execute the correct set of actions
31    if (choice == 1)
32    {  cout << "\nFor how many months? ";
33       cin  >> months;
34       charges = months * ADULT_RATE;
35       cout << "The total charges are $" << charges << endl;
36    }
37    else if (choice == 2)
38    {  cout << "\nFor how many months? ";
39       cin  >> months;
40       charges = months * CHILD_RATE;
41       cout << "The total charges are $" << charges << endl;
42    }
43    else if (choice == 3)
44    {  cout << "For how many months? ";
45       cin  >> months;
46       charges = months * SENIOR_RATE;
47       cout << "The total charges are $" << charges << endl;
48    }
49    else if (choice != 4)
50    {  cout << "The valid choices are 1 through 4. Run the\n"
51            << "program again and select one of those.\n";
52    }
53    return 0;
54 }
```

Program Output with Example Input Shown in Bold

```
   Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: 3[Enter]
For how many months? 6[Enter]
The total charges are $180.00
```

Notice that three `double` constants `ADULT_RATE`, `CHILD_RATE`, and `SENIOR_RATE` are defined in lines 10 through 12. These constants hold the monthly membership rates for adult, child, and senior citizen memberships. Also notice that the program lets the user know when an invalid choice is made. If a number other than 1, 2, 3, or 4 is entered, an error message is printed. This is known as *input validation*.

4.6 Nested `if` Statements

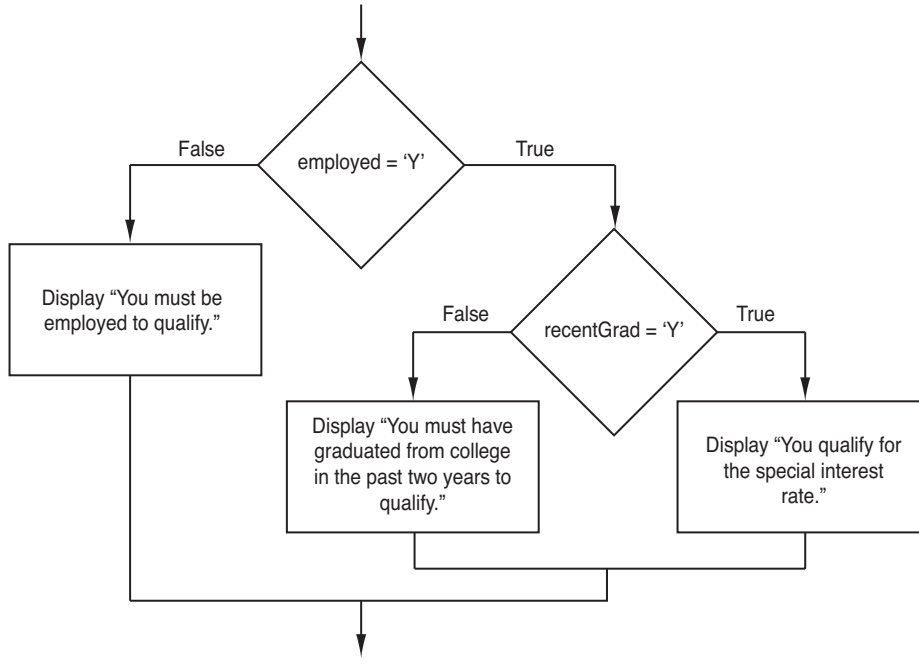
CONCEPT: **To test more than one condition, an `if` statement can be nested inside another `if` statement.**

It is possible for one `if` statement or `if/else` statement to be placed inside another one. This construct, called a *nested if*, allows you to test more than one condition to determine which block of code should be executed. For example, consider a banking program that determines whether a bank customer qualifies for a special, low interest rate on a loan. To qualify, two conditions must exist:

1. The customer must be currently employed.
2. The customer must have recently graduated from college (in the past two years).

Figure 4-7 shows a flowchart for an algorithm that could be used in such a program.

Figure 4-7



If we follow the flow of execution in the flowchart, we see that first the expression `employed == 'Y'` is tested. If this expression is false, there is no need to perform any other tests. We know that the customer does not qualify for the special interest rate. If the expression is true, however, we need to test the second condition. This is done with a nested decision structure that tests the expression `recentGrad == 'Y'`. If this

expression is true, then the customer qualifies for the special interest rate. If this expression is false, the customer does not qualify. Program 4-11 shows the code that corresponds to the logic of the flowchart. It nests one `if/else` statement inside another one.

Program 4-11

```
// This program determines whether a loan applicant qualifies for
// a special loan interest rate. It uses nested if/else statements.
#include <iostream>
using namespace std;

int main()
{
   char employed,        // Currently employed? (Y or N)
        recentGrad;      // Recent college graduate? (Y or N)

   // Is the applicant employed and a recent college graduate?
   cout << "Answer the following questions\n";
   cout << "with either Y for Yes or N for No.\n";

   cout << "Are you employed? ";
   cin  >> employed;
   cout << "Have you graduated from college in the past two years? ";
   cin  >> recentGrad;

   // Determine the applicant's loan qualifications
   if (employed == 'Y')
   {
      if (recentGrad == 'Y')         // Employed and a recent grad
      {
         cout << "You qualify for the special interest rate.\n";
      }
      else                           // Employed but not a recent grad
      {
         cout << "You must have graduated from college in the past\n";
         cout << "two years to qualify for the special interest rate.\n";
      }
   }
   else                              // Not employed
   {
      cout << "You must be employed to qualify for the "
           << "special interest rate. \n";
   }
   return 0;
}
```

(program continues)

Program 4-11 *(continued)*

Program Output with Example Input Shown in Bold

```
Answer the following questions
with either Y for Yes or N for No.
Are you employed? N[Enter]
Have you graduated from college in the past two years? Y[Enter]
You must be employed to qualify for the special interest rate.
```

Program Output with Other Example Input Shown in Bold

```
Answer the following questions
with either Y for Yes or N for No.
Are you employed? Y[Enter]
Have you graduated from college in the past two years? N[Enter]
You must have graduated from college in the past
two years to qualify for the special interest rate.
```

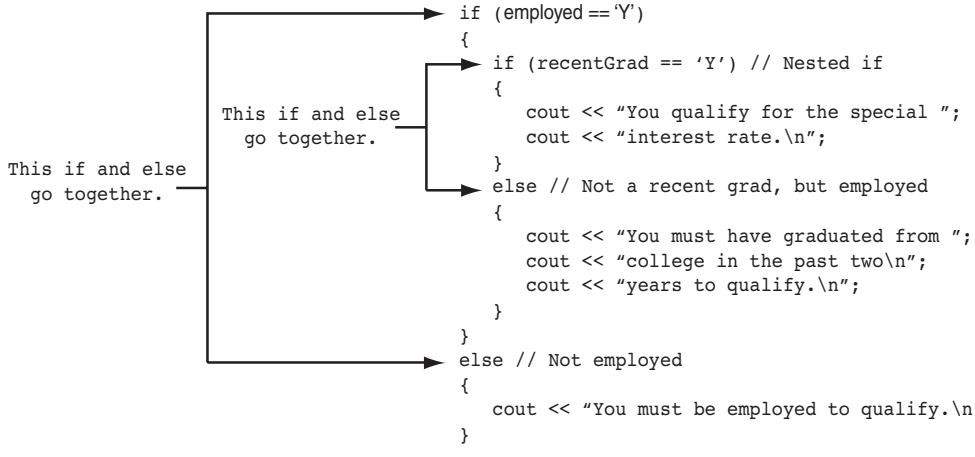
Program Output with Other Example Input Shown in Bold

```
Answer the following questions
with either Y for Yes or N for No.
Are you employed? Y[Enter]
Have you graduated from college in the past two years? Y[Enter]
You qualify for the special interest rate.
```

Let's take a closer look at this program. The `if` statement that begins on line 21 tests the expression `employed == 'Y'`. If the expression is true, the inner `if` statement that begins on line 23 is executed. If the expression is false, the program jumps to line 33 and executes the statements in the outer `else` block instead.

When you are debugging a program with nested `if/else` statements, it's important to know which `if` statement each `else` goes with. The rule for matching each `else` with the proper `if` is this: An `else` goes with the closest previous `if` statement that doesn't already have its own `else`. This is easier to see when the statements are properly indented. Figure 4-8 shows lines similar to lines 21 through 37 of Program 4-11. It illustrates how each `else` should line up with the `if` it belongs to. These visual cues are important because nested `if` statements can be very long and complex.

Figure 4-8





Checkpoint

4.17 Program 4-9 asks the user for a numeric test score and displays the letter grade for that score. Modify it so an error message is displayed if the user enters a test score less than 0.

4.18 What will the following program segment display?

```
int funny = 7, serious = 15;

funny = serious % 2;
if (funny != 1)
{  funny = 0;
   serious = 0;
}
else if (funny == 2)
{  funny = 10;
   serious = 10;
}
else
{  funny = 1;
   serious = 1;
}
cout << funny << serious << endl;
```

4.19 The following program is used in a bookstore to determine how many discount coupons a customer gets. Complete the table that appears after the program.

```
#include <iostream>
using namespace std;
int main()
{
   int numBooks, numCoupons;

   cout << "How many books are being purchased? ";
   cin  >> numBooks;

   if (numBooks < 1)
      numCoupons = 0;
   else if (numBooks < 3)
      numCoupons = 1;
   else if (numBooks < 5)
      numCoupons = 2;
   else
      numCoupons = 3;
   cout << "The number of coupons to give is " << numCoupons
        << endl;
   return 0;
}
```

If the customer purchases this many books...	...This many coupons are given.
1	
2	
3	
4	
5	
10	

4.20 Write nested `if` statements that perform the following test: If `amount1` is greater than 10 and `amount2` is less than 100, display the greater of the two.

4.7 Logical Operators

CONCEPT: Logical operators connect two or more relational expressions into one or reverse the logic of an expression.

In the previous section you saw how a program tests two conditions with two `if` statements. In this section you will see how to use logical operators to combine two or more relational expressions into one. Table 4-6 lists C++'s logical operators.

Table 4-6 Logical Operators

Operator	Meaning	Effect
`&&`	AND	Connects two expressions into one. Both expressions must be true for the overall expression to be true.
`||`	OR	Connects two expressions into one. One or both expressions must be true for the overall expression to be true. It is only necessary for one to be true, and it does not matter which.
`!`	NOT	Reverses the "truth" of an expression. It makes a true expression false, and a false expression true.

The `&&` Operator

The `&&` operator is known as the logical AND operator. It takes two expressions as operands and creates an expression that is true only when both sub-expressions are true. Here is an example of an `if` statement that uses the `&&` operator:

```
if ((temperature < 20) && (minutes > 12))
   cout << "The temperature is in the danger zone.";
```

Notice that both of the expressions being ANDed together are complete expressions that evaluate to true or false. First `temperature < 20` is evaluated to produce a true or false result. Then `minutes > 12` is evaluated to produce a true or false result. Then, finally, these two results are ANDed together to arrive at a final result for the entire expression.

The `cout` statement will only be executed if `temperature` is less than 20 AND `minutes` is greater than 12. If either relational test is false, the entire expression is false and the `cout` statement is not executed.

Table 4-7 shows a truth table for the `&&` operator. The truth table lists all the possible combinations of values that two expressions may have, and the resulting value returned by the `&&` operator connecting the two expressions. As the table shows, both sub-expressions must be true for the `&&` operator to return a true value.

Table 4-7 Logical AND

Expression	Value of the Expression
`false && false`	`false (0)`
`false && true`	`false (0)`
`true && false`	`false (0)`
`true && true`	`true (1)`



NOTE: If the sub-expression on the left side of an `&&` operator is false, the expression on the right side will not be checked. Because the entire expression is false if even just one of the sub-expressions is false, it would waste CPU time to check the remaining expression. This is called *short circuit evaluation*.

The `&&` operator can be used to simplify programs that otherwise would use nested `if` statements. Program 4-12 is similar to Program 4-11, which determines if a bank customer qualifies for a special interest rate. Program 4-12 uses a logical operator.

Program 4-12

```
1  // This program determines whether a loan applicant qualifies for
2  // a special loan interest rate. It uses the && logical operator.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     char  employed,     // Currently employed? (Y or N)
9           recentGrad;   // Recent college graduate? (Y or N)
10
11    // Is the applicant employed and a recent college graduate?
12    cout << "Answer the following questions\n";
13    cout << "with either Y for Yes or N for No.\n";
14
15    cout << "Are you employed? ";
16    cin  >> employed;
17    cout << "Have you graduated from college in the past two years? ";
18    cin  >> recentGrad;
19
```

(program continues)

Program 4-12 *(continued)*

```
20    // Determine the applicant's loan qualifications
21    if ((employed == 'Y') && (recentGrad == 'Y'))  // Uses logical AND
22       cout << "\nYou qualify for the special interest rate.\n";
23    else
24    {  cout << "\nYou must be employed and have graduated\n";
25       cout << "from college in the past two years to qualify\n";
26       cout << "for the special interest rate.\n";
27    }
28    return 0;
29 }
```

Program Output with Example Input Shown in Bold

```
Answer the following questions
with either Y for Yes or N for No.
Are you employed? Y[Enter]
Have you graduated from college in the past two years? N[Enter]

You must be employed and have graduated
from college in the past two years to qualify
for the special interest rate.
```

Note that while this program is similar to Program 4-11, it is not the exact logical equivalent. In Program 4-12 the message `"You qualify for the special interest rate"` is displayed when both the expressions `employed == 'Y'` and `recentGrad == 'Y'` are true. If either of these are false the following message displays: `"You must be employed and have graduated from college in the past two years to qualify for the special interest rate."` Program 4-11 displays different messages when the loan applicant does not qualify.

The || Operator

The `||` operator is known as the logical OR operator. It takes two expressions as operands and creates an expression that is true when either of the sub-expressions are true. Here is an example of an `if` statement that uses the `||` operator:

```
if ((temperature < 20) || (temperature > 100))
    cout << "The temperature is in the danger zone.";
```

The `cout` statement will be executed if `temperature` is less than 20 OR `temperature` is greater than 100. If either relational test is true, the entire expression is true and the `cout` statement is executed.



NOTE: The two things being ORed should both be logical expressions that evaluate to true or false. It would *not* be correct to write the `if` condition as

```
if (temperature < 20 || > 100).
```



NOTE: There is no || key on the computer keyboard. Use two | symbols. This symbol is on the backslash key. Press Shift and backslash to type it.

Table 4-8 shows a truth table for the || operator.

Table 4-8 Logical OR

Expression	Value of the Expression
`false || false`	`false (0)`
`false || true`	`true (1)`
`true || false`	`true (1)`
`true || true`	`true (1)`

All it takes for an OR expression to be true is for one of the sub-expressions to be true. It doesn't matter if the other sub-expression is false or true.



NOTE: The || operator also performs short circuit evaluation. If the sub-expression on the left side of an || operator is true, the sub-expression on the right side will not be checked. Because it is only necessary for one of the sub-expressions to be true for the whole expression to evaluate to true, it would waste CPU time to check the remaining expression.

Program 4-13 performs different tests to qualify a person for a loan. This one determines if the customer earns at least $35,000 per year or has been employed for more than five years.

Program 4-13

```
1  // This program determines whether or not an applicant qualifies
2  // for a loan. To qualify they must make at least $35,000 a year
3  // or have been at their current job for more than 5 years.
4  // It uses the logical || operator.
5  #include <iostream>
6  using namespace std;
7
8  int main()
9  {
10    double income;      // Annual income
11    int years;          // Years at the current job
12
13    // Get annual income and years on the job
14    cout << "What is your annual income? ";
15    cin  >> income;
16    cout << "How many years have you worked at your current job? ";
17    cin  >> years;
18
```

(program continues)

Program 4-13 *(continued)*

```
19    // Determine if the applicant qualifies for a loan
20    if ((income >= 35000) || (years > 5))   // Uses logical OR
21       cout << "You qualify for a loan.\n";
22    else
23    {  cout << "You must earn at least $35,000 or have been employed\n";
24       cout << "for more than 5 years to qualify for a loan.\n";
25    }
26    return 0;
27 }
```

Program Output with Example Input Shown in Bold

```
What is your annual income? 40000[Enter]
How many years have you worked at your current job? 2[Enter]
You qualify for a loan.
```

Program Output with Other Example Input Shown in Bold

```
What is your annual income? 20000[Enter]
How many years have you worked at your current job? 7[Enter]
You qualify for a loan.
```

Program Output with Other Example Input Shown in Bold

```
What is your annual income? 30000[Enter]
How many years have you worked at your current job? 3[Enter]
You must earn at least $35,000 or have been employed
for more than 5 years to qualify for a loan.
```

The message `"You qualify for a loan."` is displayed when either or both the expressions `income >= 35000` or `years > 5` are true. If both of these are false, the disqualifying message is printed.

The ! Operator

The `!` operator performs a logical NOT operation. It takes an operand and reverses its truth or falsehood. In other words, if the expression is true, the `!` operator returns false, and if the expression is false, it returns true. Here is an `if` statement using the `!` operator:

```
if (!(temperature > 100))
   cout << "You are below the maximum temperature.\n";
```

First, the expression `(temperature > 100)` is tested to be true or false. Then the `!` operator is applied to that value. If the expression `(temperature > 100)` is true, the `!` operator returns false. If it is false, the `!` operator returns true. In the example, it is equivalent to asking "is the temperature not greater than 100?"

Table 4-9 shows a truth table for the `!` operator.

Table 4-9 Logical NOT

Expression	Value of the Expression
`!false`	`true (1)`
`!true`	`false (0)`

Program 4-14 performs the same task as Program 4-13. The `if` statement, however, uses the `!` operator to determine if the user does *not* make at least $35,000 or has *not* been on the job more than five years.

Program 4-14

```
1  // This program determines whether or not an applicant qualifies
2  // for a loan. To qualify they must make at least $35,000 a year
3  // or have been at their current job for more than 5 years. It uses
4  // the ! logical operator to reverse the logic of the if statement.
5  #include <iostream>
6  using namespace std;
7
8  int main()
9  {
10    double income;      // Annual income
11    int years;          // Years at the current job
12
13    // Get annual income and years on the job
14    cout << "What is your annual income? ";
15    cin  >> income;
16    cout << "How many years have you worked at your current job? ";
17    cin  >> years;
18
19    // Determine if the applicant qualifies for a loan
20    if (!((income >= 35000) || (years > 5)))  // Uses logical NOT
21    {
22       cout << "You must earn at least $35,000 or have been employed\n";
23       cout << "for more than 5 years to qualify for a loan.\n";
24    }
25    else
26       cout << "You qualify for a loan.\n";
27    return 0;
28 }
```

Program Output 4-14 is the same as that of Program 4-13.

Boolean Variables and the ! Operator

An interesting feature of a Boolean variable is that its value can be tested just by naming it. Suppose `moreData` is a Boolean variable. Then the test

```
if (moreData == true)
```

can be written simply as

```
if (moreData)
```

and the test

```
if (moreData == false)
```

can be written simply as

```
if (!moreData)
```

In fact, this second way of testing the value of a Boolean variable is preferable. This is because the C++ constant `true` always has the value 1, but a condition that evaluates to

true may have *any* non-zero value. For example, C++ has a function called `isalpha()`, which tests whether or not a character is an alphabetic character. As you would expect, the test `isalpha('?')` evaluates to false and the test `isalpha('x')` evaluates to true. However, for some alphabetic characters, this function returns a value other than 1 to represent true. Program 4-15 illustrates this.

Program 4-15

```
1  // This program illustrates what can happen when a
2  // Boolean value is compared to the C++ constant true.
3  #include <iostream>
4  #include <cctype>          // Needed to use the isalpha function
5  using namespace std;
6
7  int main()
8  {
9      cout << "Is '?' an alphabetic character?  " << isalpha('?') << "\n";
10     cout << "Is 'X' an alphabetic character?  " << isalpha('X') << "\n";
11     cout << "Is 'x' an alphabetic character?  " << isalpha('x') << "\n\n";
12
13     cout << "Ask if(isalpha('x') == true) \n";
14     if (isalpha('x') == true)
15        cout << "The letter x IS an alphabetic character. \n\n";
16     else
17        cout << "The letter x is NOT an alphabetic character. \n\n";
18
19     cout << "Ask if(isalpha('x')) \n";
20     if (isalpha('x'))
21        cout << "The letter x IS an alphabetic character. \n";
22     else
23        cout << "The letter x is NOT an alphabetic character. \n";
24
25     return 0;
26 }
```

Program Output

```
Is '?' an alphabetic character?  0
Is 'X' an alphabetic character?  1
Is 'x' an alphabetic character?  2

Ask if(isalpha('x') == true
The letter x is NOT an alphabetic character

Ask if(isalpha('x'))
The letter x IS an alphabetic character
```

In line 14 when the condition `isalpha('x') == true` was tested, the program did not produce the desired result. The value 2 returned by the `isalpha` function was compared to the value 1, so the condition evaluated to false even though, in fact, both values being tested represent true. The code in line 20 worked correctly because the value 2, returned by the `isalpha` function, was correctly interpreted as true.

Precedence and Associativity of Logical Operators

Table 4-10 shows the precedence of C++'s logical operators, from highest to lowest.

Table 4-10 Precedence of Logical Operators

!
&&
||

The ! operator has a higher precedence than many of the C++ operators. Therefore, to avoid an error, it is a good idea always to enclose its operand in parentheses, unless you intend to apply it to a variable or a simple expression with no other operators. For example, consider the following expressions:

```
!(x > 2)
!x > 2
```

The first expression applies the ! operator to the expression x > 2. It is asking "is x not greater than 2?" The second expression, however, applies the ! operator to x only. It is asking "is the logical negation of x greater than 2?" Suppose x is set to 5. Since 5 is nonzero, it would be considered true, so the ! operator would reverse it to false, which is 0. The > operator would then determine if 0 is greater than 2. To avoid such an error, it is wise to always use parentheses.

The && and || operators rank lower in precedence than relational operators, which means that relational expressions are evaluated before their results are logically ANDed or ORed.

```
a > b && x < y    is the same as   (a > b) && (x < y)
a > b || x < y    is the same as   (a > b) || (x < y)
```

Thus you don't normally need parentheses when mixing relational operators with && and ||. However it is a good idea to use them anyway to make your intent clearer for someone reading the program.

Parentheses are even more strongly recommended anytime && and || operators are both used in the same expression. This is because && has a higher precedence than ||. Without parentheses to indicate which you want done first, && will always be done before ||, which might not be what you intended. Assume recentGrad, employed, and goodCredit are three Boolean variables. Then the expression

```
recentGrad || employed && goodCredit
```

is the same as

```
recentGrad ||(employed && goodCredit)
```

and *not* the same as

```
(recentGrad || employed)&& goodCredit
```

Checking Numeric Ranges with Logical Operators

Logical operators are effective for determining if a number is in or out of a range. When determining if a number is inside a numeric range, it's best to use the && operator. For example, the following if statement checks the value in x to determine if it is in the range of 20 through 40.

```
if ((x >= 20) && (x <= 40))
   cout << x << " is in the acceptable range.\n";
```

The expression in the `if` statement will be true only when `x` is both greater than or equal to 20 AND less than or equal to 40. The value of `x` must be within the range of 20 through 40 for this expression to be true.

When determining if a number is outside a range, the `||` operator is best to use. The following statement determines if the value of `x` is outside the range of 20 to 40:

```
if ((x < 20) || (x > 40))
   cout << x << " is outside the acceptable range.\n";
```

It's important not to get the logic of these logical operators confused. For example, the following `if` statement would never test true:

```
if ((x < 20) && (x > 40))
   cout << x << " is outside the acceptable range.\n";
```

Obviously, `x` can never be less than 20 and at the same time greater than 40.



NOTE: C++ does not allow you to check numeric ranges with expressions such as `5 < x < 20`. Instead you must use a logical operator to connect two relational expressions, as previously discussed.



Checkpoint

4.21 The following truth table shows various combinations of the values `true` and `false` connected by a logical operator. Complete the table by indicating if the result of such a combination is true or false.

Logical Expression	Result (true or false)
`true && false`	
`true && true`	
`false && false`	
`true || false`	
`true || true`	
`false || false`	
`!true`	
`!false`	

4.22 If `a = 2`, `b = 4`, and `c = 6`, indicate whether each of the following conditions is true or false:

A) `(a == 4) || (b > 2)`
B) `(6 <= c) && (a > 3)`
C) `(1 != b) && (c != 3)`
D) `(a >= -1) || (a <= b)`
E) `!(a > 2)`

4.23 If `a = 2`, `b = 4`, and `c = 6`, is the following expression true or false?

```
(b > a) || (b > c) && (c == 5)
```

4.24 Rewrite the following using the `!` operator so that the logic remains the same.

```
if (activeEmployee == false)
```

4.8 Validating User Input

CONCEPT: As long as the user of a program enters bad input, the program will produce bad output. Programs should be written to filter out bad input.

Perhaps the most famous saying of the computer world is "garbage in, garbage out." The integrity of a program's output is only as good as its input, so you should try to make sure garbage does not go into your programs. *Input validation* is the process of inspecting information given to a program by the user and determining if it is valid. A good program should give clear instructions about the kind of input that is acceptable, and not assume the user has followed those instructions. Here are just a few examples of input validations performed by programs:

- Numbers are checked to ensure they are within a range of possible values. For example, there are 168 hours in a week. It is not possible for a person to be at work longer than 168 hours in one week.
- Values are checked for their "reasonableness." Although it might be possible for a person to be at work for 168 hours per week, it is not probable.
- Items selected from a menu or some other set of choices are checked to ensure they are available options.
- Variables are checked for values that might cause problems, such as division by zero.

Program 4-16 is a modification of Program 4-7, the test score program. It rejects any test score less than 0 or greater than 100.

Program 4-16

```
1  // This program uses an if/else if statement to assign a
2  // letter grade (A, B, C, D, or F) to a numeric test score.
3  // It validates the user's input.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     int testScore;        // Holds a numeric test score
10        char grade;        // Holds a letter grade
11
12    // Get the numeric score
13    cout << "Enter your numeric test score and I will\n";
14    cout << "tell you the letter grade you earned: ";
15    cin  >> testScore;
16
17    if ((testScore < 0) || (testScore > 100)) // Input validation
18    {  // testScore is invalid
19       cout << testScore << " is an invalid score.\n";
20       cout << "Run the program again and enter a value\n";
21       cout << "in the range of 0 to 100.\n";
22    }
```

(program continues)

Program 4-16 *(continued)*

```
23    else
24    {  // testScore is valid so determine the letter grade
25       if (testScore < 60)
26          grade = 'F';
27       else if (testScore < 70)
28          grade = 'D';
29       else if (testScore < 80)
30          grade = 'C';
31       else if (testScore < 90)
32          grade = 'B';
33       else     // If we got this far, testScore must be >= 90
34          grade = 'A';
35
36       // Display the letter grade
37       cout << "Your grade is " << grade << endl;
38    }
39    return 0;
40 }
```

Program Output with Example Input Shown in Bold

```
Enter your numeric test score and I will
tell you the letter grade you earned: -12[Enter]
-12 is an invalid score.
Run the program again and enter a value
in the range of 0 to 100.
```

Program Output with Other Example Input Shown in Bold

```
Enter your numeric test score and I will
tell you the letter grade you earned: 81[Enter]
Your grade is B
```

In Chapter 5 you will learn an even better way to validate input data.

4.9 More About Variable Definitions and Scope

CONCEPT: The scope of a variable is limited to the block in which it is defined.

C++ allows you to create variables almost anywhere in a program. It is a common practice to define all of a function's variables at the top of the function. However, especially in longer programs, variables are sometimes defined near the part of the program where they are used. This makes the purpose of the variable more evident. Program 4-17 is a modification of Program 4-13, which determines if the user qualifies for a loan. The definitions of the variables `income` and `years` have been moved to later points in the program.

Program 4-17

```
1  // This program determines whether or not an applicant qualifies
2  // for a loan. It demonstrates late variable declaration.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // Get annual income and years on the job
9     cout << "What is your annual income? ";
10    double income;                 // Variable definition
11    cin  >> income;
12
13    cout << "How many years have you worked at "
14         << "your current job? ";
15    int years;                     // Variable definition
16    cin  >> years;
17
18    // Determine if the applicant qualifies for a loan
19    if ((income >= 35000) || (years > 5))
20       cout << "You qualify.\n";
21    else
22    {  cout << "You must earn at least $35,000 or have\n";
23       cout << "been employed for more than 5 years.\n";
24    }
25    return 0;
26 }
```

Recall from Chapter 2 that the scope of a variable is defined as the part of the program where the variable may be used. In Program 4-17, the scope of the `income` variable is the part of the program in lines 10 through 25. The scope of the `years` variable is the part of the program in lines 15 through 25.

The variables `income` and `years` are defined inside function `main`'s braces. Variables defined inside a set of braces are said to have *local scope* or *block scope*. They may only be used in the part of the program between their definition and the block's closing brace.

You may define variables inside any block. For example, look at Program 4-18. This version of the loan program defines the variable `years` inside the block of the `if` statement.

Program 4-18

```
1  // This program determines whether or not an applicant qualifies
2  // for a loan. It demonstrates a variable defined in an inner block.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // Get annual income
9     cout << "What is your annual income? ";
```

(program continues)

Program 4-18 *(continued)*

```
10    double income;
11    cin  >> income;
12
13    // If income is high enough, get years worked
14    if (income >= 35000)
15    {  cout << "How many years have you worked at "
16            << "your current job? ";
17       int years;       // Variable defined inside the if block
18       cin >> years;
19
20       if (years > 5)
21          cout << "You qualify.\n";
22       else
23          cout << "You must have been employed for "
24               << "more than 5 years to qualify.\n";
25    }
26    else
27       cout << "You must earn at least $35,000 to qualify.\n";
28
29    return 0;
30 }
```

The scope of `years` is the part of the program from lines 17 through 24, from the point of its definition to the end of the block in which it is defined. The variable is not visible before its definition or after the closing brace of the block. This is true of any variable defined inside a set of braces.



NOTE: When a program is running and it enters the section of code that constitutes a variable's scope, it is said that the variable *comes into scope*. This simply means the variable is now visible and the program may reference it. Likewise, when a variable *leaves scope*, it may no longer be used.

Variables with the Same Name

When a block is nested inside another block, a variable defined in the inner block may have the same name as a variable defined in the outer block. As long as the variable in the inner block is visible, however, the variable in the outer block will be hidden. This is illustrated by Program 4-19.

Program 4-19

```
1  // This program uses two variables with the same name.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int number;       // Define a variable named number
```

(program continues)

Program 4-19 *(continued)*

```
 8
 9    cout << "Enter a number greater than 0: ";
10    cin  >> number;
11
12    if (number > 0)
13    {  int number;    // Define another variable named number
14
15       cout << "Now enter another number: ";
16       cin  >> number;
17       cout << "The second number you entered was ";
18       cout << number << endl;
19    }
20    cout << "Your first number was " << number << endl;
21    return 0;
22 }
```

Program Output with Example Input Shown in Bold

```
Enter a number greater than 0: 2[Enter]
Now enter another number: 7[Enter]
The second number you entered was 7
Your first number was 2
```

Program 4-19 has two separate variables named `number`. The `cin` and `cout` statements in the inner block (belonging to the `if` statement) can only work with the `number` variable defined in that block. As soon as the program leaves that block, the inner `number` goes out of scope, revealing the outer `number` variable.



WARNING! Although it's perfectly acceptable to define variables inside nested blocks, you should avoid giving them the same names as variables in the outer blocks. It's too easy to confuse one variable with another.



Checkpoint

4.25 Write an `if` statement that prints the message "The number is not valid" if the variable `speed` is outside the range 0 through 200.

4.26 The following program skeleton asks the user for two numbers and then multiplies them. The first should be negative and the second should be positive. Write the input validation code for both numbers.

```
#include <iostream>
using namespace std;
int main()
{
   int first, second, result;
   cout << "Enter a negative integer: ";
   cin  >> first;
   cout << "Now enter a positive integer: ";
   cin  >> second;
```

```
   //
   // Write input validation code
   //
   result = first * second;
   cout << first << " times " << second << " is "
        << result << endl;
   return 0;
}
```

4.27 Find and fix the errors in the following program.

```
#include <iostream>
using namespace std;

int main()
{
   cout << "This program calculates the area of a "
        << "rectangle. Enter the length: ";
   cin  >> length;
   cout << "enter the width: ";
   cin  >> width;
   int length, width, area;
   area = length * width;
   cout << "The area is " << area << endl;
   return 0;
}
```

4.28 What will the following program print if 40 is entered for `test1` and 30 for `test2`?

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Enter your first test score: ";
   int test1;
   cin  >> test1;
   cout << "Enter your second test score: ";
   int test2;
   cin  >> test2;
   int sum = test1 + test2;
   if (sum > 50)
   {  test1 += 10;
      test2 += 10;
      int sum = test1 + test2;
   }
   cout << "test 1: " << test1 << endl;
   cout << "test 2: " << test2 << endl;
   cout << "sum   : " << sum << endl;
   return 0;
}
```

4.10 Comparing Characters and Strings

CONCEPT: **Relational operators can also be used to compare characters and string objects.**

Earlier in this chapter you learned to use relational operators to compare numeric values. They can also be used to compare characters and string objects.

Comparing Characters

As you learned in Chapter 3, characters are actually stored in memory as integers. On most systems, this integer is the ASCII value of the character. For example, the letter 'A' is represented by the number 65, the letter 'B' is represented by the number 66, and so on. Table 4-11 shows the ASCII numbers that correspond to some of the commonly used characters.

Table 4-11 ASCII Values of Commonly Used Characters

Character	ASCII Value
'0'–'9'	48–57
'A'–'Z'	65–90
'a'–'z'	97–122
blank	32
period	46

Notice that every character, even the blank, has an ASCII code associated with it. Notice also that the ASCII code of a character representing a digit, such as `'1'` or `'2'`, is not the same as the value of the digit itself. A complete table showing the ASCII values for all characters can be found in Appendix A.

When two characters are compared, it is actually their ASCII values that are being compared. `'A' < 'B'` because the ASCII value of `'A'` (65) is less than the ASCII value of `'B'` (66). Likewise `'1' < '2'` because the ASCII value of `'1'` (49) is less than the ASCII value of `'2'` (50). However, as shown in Table 4-11, lowercase letters have higher numbers than uppercase letters, so `'a' > 'Z'`. Program 4-20, which is similar to Program 4-10, shows how characters can be compared with relational operators. Notice that the menu choices are now letters.

Program 4-20

```
1 // This menu-driven program shows how relational operators can be
2 // used to compare characters. Character inputs are tested to make
3 // sure they are within the set of legal menu choices.
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;
7
```

(program continues)

Program 4-20 *(continued)*

```
 8 int main()
 9 {
10    // Constants for membership rates
11    const double ADULT_RATE  = 40.0;
12    const double CHILD_RATE  = 20.0;
13    const double SENIOR_RATE = 30.0;
14
15    char choice;          // Menu choice
16    int months;           // Number of months
17    double charges;       // Monthly charges
18
19    // Display the menu and get the user's choice
20    cout << "   Health Club Membership Menu\n";
21    cout << "A. Standard Adult Membership\n";
22    cout << "B. Child Membership\n";
23    cout << "C. Senior Citizen Membership\n";
24    cout << "D. Quit the Program\n\n";
25    cout << "Enter your choice: ";
26    cin.get(choice);
27
28    // Set the numeric output formatting
29    cout << fixed << showpoint << setprecision(2);
30
31    if ((choice < 'A') || (choice > 'D'))
32    {  cout << "The valid choices are A through D.\n";
33       cout << "Run the program again and select one of those.\n";
34    }
35    else if (choice == 'A')
36    {  cout << "For how many months? ";
37       cin  >> months;
38       charges = months * ADULT_RATE;
39       cout << "The total charges are $" << charges << endl;
40    }
41    else if (choice == 'B')
42    {  cout << "For how many months? ";
43       cin  >> months;
44       charges = months * CHILD_RATE;
45       cout << "The total charges are $" << charges << endl;
46    }
47    else if (choice == 'C')
48    {  cout << "For how many months? ";
49       cin  >> months;
50       charges = months * SENIOR_RATE;
51       cout << "The total charges are $" << charges << endl;
52    }
53    return 0;
54 }
```

(program continues)

Program 4-20 *(continued)*

Program Output with Example Input Shown in Bold

```
   Health Club Membership Menu
A. Standard Adult Membership
B. Child Membership
C. Senior Citizen Membership
D. Quit the Program

Enter your choice: C[Enter]
For how many months? 6[Enter]
The total charges are $180.00
```

Comparing String Objects

String objects can also be compared with relational operators. As with individual characters, when two strings are compared, it is actually the ASCII value of the characters making up the strings that are being compared. For example, assume the following definitions exist in a program:

```
string set1 = "ABC";
string set2 = "XYZ";
```

The object `set1` is considered less than the object `set2` because the characters `"ABC"` alphabetically precede (have lower ASCII values than) the characters `"XYZ"`. So, the following `if` statement will cause the message "`set1 is less than set2.`" to be displayed on the screen.

```
if (set1 < set2)
   cout << "set1 is less than set2.";
```

One by one, each character in the first operand is compared with the character in the corresponding position in the second operand. If all the characters in both strings match, the two strings are equal. Other relationships can be determined if two characters in corresponding positions do not match. The first operand is less than the second operand if the first mismatched character in the first operand is less than its counterpart in the second operand. Likewise, the first operand is greater than the second operand if the first mismatched character in the first operand is greater than its counterpart in the second operand.

For example, assume a program has the following definitions:

```
string name1 = "Mary";
string name2 = "Mark";
```

The value in `name1`, `"Mary"`, is greater than the value in `name2`, `"Mark"`. This is because the first three characters in `name1` have the same ASCII values as the first three characters in `name2`, but the `'y'` in the fourth position of `"Mary"` has a greater ASCII value than the `'k'` in the corresponding position of `"Mark"`.

Any of the relational operators can be used to compare two `string` objects. Here are some of the valid comparisons of `name1` and `name2`.

```
name1 >  name2          // true
name1 <= name2          // false
name1 != name2          // true
```

String objects can also, of course, be compared to string constants:

```
name1 < "Mary Jane"     // true
```

Program 4-21 further demonstrates how relational operators can be used with string objects.

Program 4-21

```
1  // This program uses relational operators to compare a string
2  // entered by the user with valid stereo part numbers.
3  #include <iostream>
4  #include <iomanip>
5  #include <string>
6  using namespace std;
7
8  int main()
9  {
10    const double PRICE_A = 249.0,
11                 PRICE_B = 299.0;
12
13    string partNum;              // Holds a stereo part number
14
15    // Display available parts and get the user's selection
16    cout << "The stereo part numbers are:\n";
17    cout << "Boom Box    : part number S-29A \n";
18    cout << "Shelf Model: part number S-29B \n";
19    cout << "Enter the part number of the stereo you\n";
20    cout << "wish to purchase: ";
21    cin  >> partNum;
22
23    // Set the numeric output formatting
24    cout << fixed << showpoint << setprecision(2);
25
26    // Determine and display the correct price
27    if (partNum == "S-29A")
28       cout << "The price is $" << PRICE_A << endl;
29    else if (partNum == "S-29B")
30       cout << "The price is $" << PRICE_B << endl;
31    else
32       cout << partNum << " is not a valid part number.\n";
33    return 0;
34 }
```

Program Output with Example Input Shown in Bold

```
The stereo part numbers are:
Boom Box    : part number S-29A
Shelf Model: part number S-29B
Enter the part number of the stereo you
wish to purchase: S-29A[Enter]
The price is $249.00
```



NOTE: C-strings, unlike string objects, cannot be compared with relational operators. To compare C-strings (i.e., strings defined as arrays of characters) you must use the `strcmp` function, which is introduced in Chapter 12.

Checkpoint

4.29 Indicate whether each of the following relational expressions is true or false. Refer to the ASCII table in Appendix A if necessary.

A) 'a' < 'z'
B) 'a' == 'A'
C) '5' < '7'
D) 'a' < 'A'
E) '1' == 1
F) '1' == 49

4.30 Indicate whether each of the following relational expressions is true or false. Refer to the ASCII table in Appendix A if necessary.

A) "Bill" == "BILL"
B) "Bill" < "BILL"
C) "Bill" < "Bob"
D) "189" > "23"
E) "189" > "Bill"
F) "Mary" < "MaryEllen"
G) "MaryEllen" < "Mary Ellen"

4.11 The Conditional Operator

CONCEPT: You can use the conditional operator to create short expressions that work like `if/else` statements.

The conditional operator is powerful and unique. It provides a shorthand method of expressing a simple `if/else` statement. The operator consists of the question-mark (?) and the colon(:). Its format is

```
expression ? expression : expression;
```

Here is an example of a statement using the conditional operator:

```
x < 0 ? y = 10 : z = 20;
```

This statement is called a *conditional expression* and consists of three sub-expressions separated by the `?` and `:` symbols. The expressions are `x < 0`, `y = 10`, and `z = 20`.

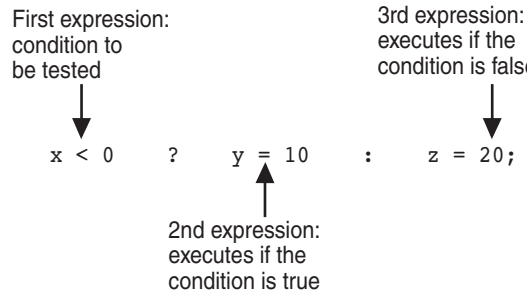
```
x < 0     ?     y = 10     :     z = 20;
```

The conditional expression above performs the same operation as this `if/else` statement:

```
if (x < 0)
   y = 10;
else
   z = 20;
```

The part of the conditional expression that comes before the question mark is the condition to be tested. It's like the expression in the parentheses of an `if` statement. If the condition is true, the part of the statement between the `?` and the `:` is executed. Otherwise, the part after the `:` is executed. Figure 4-9 illustrates the roles played by the three sub-expressions.

Figure 4-9



If it helps, you can put parentheses around the sub-expressions, as shown here:

```
(x < 0) ? (y = 10) : (z = 20);
```



NOTE: Because it takes three operands, the conditional operator is a *ternary* operator.

Using the Value of a Conditional Expression

Remember, in C++ all expressions have a value, and this includes the conditional expression. If the first sub-expression is true, the value of the conditional expression is the value of the second sub-expression. Otherwise it is the value of the third sub-expression. Here is an example of an assignment statement that uses the value of a conditional expression:

```
a = (x > 100) ? 0 : 1;
```

The value assigned to variable `a` will be either 0 or 1, depending upon whether `x` is greater than 100. This statement has the same logic as the following `if/else` statement:

```
if  (x > 100)
    a = 0;
else
    a = 1;
```

Program 4-22 can be used to help a consultant calculate her charges. Her rate is $50.00 per hour, but her minimum charge is for five hours. The conditional operator is used in a statement that ensures the number of hours does not go below five.

Program 4-22

```
1  // This program illustrates the conditional operator.
2  // It adjusts hours to 5 if fewer than 5 hours were worked.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     const double PAY_RATE = 50.0;
10    double hours, charges;
11
12    cout << "How many hours were worked? ";
13    cin  >> hours;
14
15    hours = (hours < 5) ? 5 : hours;           // Conditional operator
16    charges = PAY_RATE * hours;
17
18    cout << fixed << showpoint << setprecision(2);
19    cout << "The charges are $" << charges << endl;
20    return 0;
21 }
```

Program Output with Example Input Shown in Bold

```
How many hours were worked? 10[Enter]
The charges are $500.00
```

Program Output with Other Example Input Shown in Bold

```
How many hours were worked? 2[Enter]
The charges are $250.00
```

Let's look more closely at the statement in line 15 with the conditional expression:

```
hours = hours < 5 ? 5 : hours;
```

If the value in `hours` is less than 5, then 5 is stored in `hours`. Otherwise `hours` is assigned the value it already has `hours` will not have a value less than 5 when it is used in the next statement, which calculates the consultant's charges.

As you can see, the conditional operator gives you the ability to pack decision-making power into a concise line of code. With a little imagination it can be applied to many other programming problems. For instance, consider the following statement:

```
cout << "Your grade is: " << (score < 60 ? "Fail." : "Pass.");
```

If you were to use an `if/else` statement, this statement would be written as follows:

```
if (score < 60)
   cout << "Your grade is: Fail.";
else
   cout << "Your grade is: Pass.";
```



NOTE: The parentheses are placed around the conditional expression because the << operator has higher precedence than the `?:` operator. Without the parentheses, just the value of the expression `score < 60` would be sent to `cout`.



Checkpoint

4.31 Rewrite the following `if/else` statements as conditional expressions.

A)
```
if (x > y)
    z = 1;
else
    z = 20;
```

B)
```
if (temp > 45)
    population = base * 10;
else
    population = base * 2;
```

C)
```
if (hours > 40)
    wages *= 1.5;
else
    wages *= 1;
```

D)
```
if (result >= 0)
    cout << "The result is positive\n";
else
    cout << "The result is negative.\n";
```

4.32 Rewrite the following conditional expressions as `if/else` statements.

A) `j = k > 90 ? 57 : 12;`

B) `factor = x >= 10 ? y * 22 : y * 35;`

C) `total += count == 1 ? sales : count * sales;`

D) `cout << ((num % 2) == 0) ? "Even\n" : "Odd\n");`

4.33 What will the following program display?

```
#include <iostream>
using namespace std;

int main()
{
   const int UPPER = 8, LOWER = 2;
   int num1, num2, num3 = 12, num4 = 3;

   num1 = num3 < num4 ? UPPER : LOWER;
   num2 = num4 > UPPER ? num3 : LOWER;
   cout << num1 << " " <<  num2 << endl;
   return 0;
}
```

4.12 The `switch` Statement

CONCEPT: **The `switch` statement lets the value of a variable or expression determine where the program will branch to.**

A branch occurs when one part of a program causes another part to execute. The `if/else if` statement allows your program to branch into one of several possible paths. It performs a series of tests (usually relational) and branches when one of these tests is true. The `switch` statement is a similar mechanism. It, however, tests the value of an integer expression and then uses that value to determine which set of statements to branch to. Here is the format of the `switch` statement:

```
switch (IntegerExpression)
{
   case ConstantExpression:  // Place one or more
                             // statements here

   case ConstantExpression:  // Place one or more
                             // statements here

   // case statements may be repeated
   // as many times as necessary

   default:                  // Place one or more
                             // statements here
}
```

The first line of the statement starts with the word `switch`, followed by an integer expression inside parentheses. This can be either of the following:

- A variable of any of the integer data types (including `char`).
- An expression whose value is of any of the integer data types.

On the next line is the beginning of a block containing several `case` statements. Each `case` statement is formatted in the following manner:

```
case ConstantExpression:   // Place one or more
                           // statements here
```

After the word `case` is a constant expression (which must be of an integer type such as an `int` or `char`), followed by a colon. The constant expression can be either an integer literal or an integer named constant. The expression cannot be a variable and it cannot be a Boolean expression such as `x < 22` or `n == 25`. The `case` statement marks the beginning of a section of statements. These statements are branched to if the value of the `switch` expression matches that of the `case` expression.



WARNING! The expressions of each `case` statement in the block must be unique.

An optional `default` section comes after all the `case` statements. This section is branched to if none of the `case` expressions match the switch expression. Thus it functions like a trailing `else` in an `if/else if` statement.

Program 4-23 shows how a simple `switch` statement works.

Program 4-23

```
1  // This program demonstrates the use of a switch statement.
2  // The program simply tells the user what character they entered.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     char choice;
9
10    cout << "Enter A, B, or C: ";
11    cin  >> choice;
12
13    switch (choice)
14    {
15       case 'A':cout << "You entered A.\n";
16                break;
17       case 'B':cout << "You entered B.\n";
18                break;
19       case 'C':cout << "You entered C.\n";
20                break;
21       default: cout << "You did not enter A, B, or C!\n";
22    }
23    return 0;
24 }
```

Program Output with Example Input Shown in Bold

```
Enter A, B, or C: B[Enter]
You entered B.
```

Program Output with Different Example Input Shown in Bold

```
Enter A, B, or C: F[Enter]
You did not enter A, B, or C!
```

The first `case` statement is `case 'A':`, the second is `case 'B':`, and the third is `case 'C':`. These statements mark where the program is to branch to if the variable `choice` contains the values `'A'`, `'B'`, or `'C'`. (Remember, character variables and constants are considered integers.) The `default` section is branched to if the user enters anything other than A, B, or C.

Notice the `break` statements at the end of the `case 'A'`, `case 'B'`, and `case 'C'` sections.

```
switch (choice)
{
   case 'A':cout << "You entered A.\n";
            break; <-------
   case 'B':cout << "You entered B.\n";
            break; <-------
   case 'C':cout << "You entered C.\n";
            break; <-------
   default:cout << "You did not enter A, B, or C!\n";
}
```

The `break` statement causes the program to exit the `switch` statement. The next statement executed after encountering a `break` statement will be whatever statement follows the closing brace that terminates the `switch` statement. A `break` statement is needed whenever you want to "break out of" a `switch` statement because it is not automatically exited after carrying out a set of statements the way an `if/else if` statement is.

The `case` statements show the program where to start executing in the block and the `break` statements show the program where to stop. Without the `break` statements, the program would execute all of the lines from the matching `case` statement to the end of the block.



NOTE: The `default` section (or the last `case` section if there is no `default`) does not need a `break` statement. Some programmers prefer to put one there anyway for consistency.

Program 4-24 is a modification of Program 4-23 that demonstrates what happens if the `break` statements are omitted.

Program 4-24

```
1  // This program demonstrates how a switch statement
2  // works if there are no break statements.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     char choice;
9
10    cout << "Enter A, B, or C: ";
11    cin  >> choice;
12
13    // The following switch statement is missing its break statements!
14    switch (choice)
15    {
16       case 'A':cout << "You entered A.\n";
17       case 'B':cout << "You entered B.\n";
18       case 'C':cout << "You entered C.\n";
19       default :cout << "You did not enter A, B, or C!\n";
20    }
21    return 0;
22 }
```

(program continues)

Program 4-24 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter A, B, or C: A[Enter]
You entered A.
You entered B.
You entered C.
You did not enter A, B, or C!
```

Program Output with Different Example Input Shown in Bold

```
Enter A, B, or C: C[Enter]
You entered C.
You did not enter A, B, or C!
```

Without the `break` statement, the program "falls through" all of the statements below the one with the matching `case` expression. Sometimes this is what you want. Program 4-25 lists the features of three TV models a customer may choose from. The model 100 has remote control. The model 200 has remote control and stereo sound. The model 300 has remote control, stereo sound, and picture-in-a-picture capability. The program uses a `switch` statement with carefully omitted `breaks` to print the features of the selected model.

Program 4-25

```
1  // This program is carefully constructed to use the "fall through"
2  // feature of the switch statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int modelNum;
9
10    // Display available models and get the user's choice
11    cout << "Our TVs come in three models:\n";
12    cout << "The 100, 200, and 300. Which do you want? ";
13    cin  >> modelNum;
14
15    // Display the features of the selected model
16    cout << "That model has the following features:\n";
17    switch (modelNum)
18    {
19       case 300:  cout << "    Picture-in-a-picture\n";
20       case 200:  cout << "    Stereo sound\n";
21       case 100:  cout << "    Remote control\n";
22                  break;
23       default :  cout << "You can only choose the 100, 200, or 300.\n ";
24    }
25    return 0;
26 }
```

(program continues)

Program 4-25 *(continued)*

Program Output with Example Input Shown in Bold

```
Our TVs come in three models:
The 100, 200, and 300. Which do you want? 100[Enter]
That model has the following features:
   Remote control
```

Program Output with Different Example Input Shown in Bold

```
Our TVs come in three models:
The 100, 200, and 300. Which do you want? 200[Enter]
That model has the following features:
   Stereo sound
   Remote control
```

Program Output with Different Example Input Shown in Bold

```
Our TVs come in three models:
The 100, 200, and 300. Which do you want? 300[Enter]
That model has the following features:
   Picture-in-a-picture
   Stereo sound
   Remote control
```

Program Output with Different Example Input Shown in Bold

```
Our TVs come in three models:
The 100, 200, and 300. Which do you want? 500[Enter]
That model has the following features:
You can only choose the 100, 200, or 300.
```

Another example of how useful this "fall through" capability can be is when you want the program to branch to the same set of statements for multiple `case` expressions. For instance, Program 4-26 asks the user to select a grade of dog food. The available choices are A, B, and C. The `switch` statement will recognize either upper or lowercase letters.

Program 4-26

```
1  // The switch statement in this program uses the "fall through" feature
2  // to catch both uppercase and lowercase letters entered by the user.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     char feedGrade;
9
10    // Get the desired grade of feed
11    cout << "Our dog food is available in three grades:\n";
12    cout << "A, B, and C. Which do you want pricing for? ";
13    cin  >> feedGrade;
14
```

(program continues)

Program 4-26 *(continued)*

```
15    // Find and display the price
16    switch(feedGrade)
17    {
18       case 'a':
19       case 'A': cout << "30 cents per pound.\n";
20                 break;
21       case 'b':
22       case 'B': cout << "20 cents per pound.\n";
23                 break;
24       case 'c':
25       case 'C': cout << "15 cents per pound.\n";
26                 break;
27       default : cout << "That is an invalid choice.\n";
28    }
29    return 0;
30 }
```

Program Output with Example Input Shown in Bold

```
Our dog food is available in three grades:
A, B, and C. Which do you want pricing for? b[Enter]
20 cents per pound.
```

Program Output with Different Example Input Shown in Bold

```
Our dog food is available in three grades:
A, B, and C. Which do you want pricing for? B[Enter]
20 cents per pound.
```

When the user enters `'a'` the corresponding `case` has no statements associated with it, so the program falls through to the next `case`, which corresponds with `'A'`.

```
case 'a':
case 'A':cout << "30 cents per pound.\n";
         break;
```

The same is technique is used for `'b'` and `'c'`.

Using `switch` in Menu-Driven Systems

The `switch` statement is a natural mechanism for building menu-driven systems like the one we built in Program 4-10. However in that program, once the user selects which health club package to purchase, the program uses an `if/else if` statement to calculate the charges. Program 4-27 is a modification of that program that uses a `switch` statement instead. Notice that this program also uses a logical AND operator so that the prompt and input for the number of months only has to appear once.

Program 4-27

```
// This menu-driven program uses a switch statement to carry out
// the appropriate set of actions based on the user's menu choice.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   // Constants for membership rates
   const double ADULT_RATE  = 40.0;
   const double CHILD_RATE  = 20.0;
   const double SENIOR_RATE = 30.0;

   int choice;           // Menu choice
   int months;           // Number of months
   double charges;       // Monthly charges

   // Display the menu and get the user's choice
   cout << "   Health Club Membership Menu\n\n";
   cout << "1. Standard Adult Membership\n";
   cout << "2. Child Membership\n";
   cout << "3. Senior Citizen Membership\n";
   cout << "4. Quit the Program\n\n";
   cout << "Enter your choice: ";
   cin  >> choice;

   // Validate and process the menu choice
   if ((choice >= 1) && (choice <= 3))
   {  cout << "For how many months? ";
      cin  >> months;

      // Set charges based on user input
      switch (choice)
      {
         case 1: charges = months * ADULT_RATE;
                 break;
         case 2: charges = months * CHILD_RATE;
                 break;
         case 3: charges = months * SENIOR_RATE;
      }
      // Display the monthly charges
      cout << fixed << showpoint << setprecision(2);
      cout << "The total charges are $" << charges << endl;
   }
   else if (choice != 4)
   {  cout << "The valid choices are 1 through 4.\n";
      cout << "Run the program again and select one of these.\n";
   }
   return 0;
}
```

(program continues)

Program 4-27 *(continued)*

Program Output with Example Input Shown in Bold

```
   Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: 2[Enter]
For how many months? 6[Enter]
The total charges are $120.00
```



Checkpoint

4.34 Explain why you cannot convert the following `if/else if` statement into a `switch` statement.

```
if (temp == 100)
   x = 0;
else if (population > 1000)
   x = 1;
else if  (rate < .1)
   x = -1;
```

4.35 What is wrong with the following `switch` statement?

```
switch (temp)
{
   case temp < 0 :  cout << "Temp is negative.\n";
                    break;
   case temp == 0:  cout << "Temp is zero.\n";
                    break;
   case temp > 0 :  cout << "Temp is positive.\n";
                    break;
}
```

4.36 What will the following program segment display?

```
int funny = 7, serious = 15;

funny = serious * 2;
switch (funny)
{  case 0 :  cout << "That is funny.\n";
             break;
   case 30:  cout << "That is serious.\n";
             break;
   case 32:  cout << "That is seriously funny.\n";
             break;
   default:  cout << funny << endl;
}
```

4.37 Complete the following program skeleton by writing a `switch` statement that displays `"one"` if the user has entered 1, `"two"` if the user has entered 2, and `"three"` if the user has entered 3. If a number other than 1, 2, or 3 is entered, the program should display an error message.

```
#include <iostream>
using namespace std;

int main()
{
   int userNum;

   cout << "Enter one of the numbers 1, 2, or 3: ";
   cin  >> userNum;

   // Write the switch statement here.

   return 0;
}
```

4.38 Rewrite the following program. Use a `switch` statement instead of the `if/else if` statement.

```
#include <iostream>
using namespace std;

int main()
{
   int selection;

   cout << "Which formula do you want to see?\n\n";
   cout << "1. Area of a circle\n";
   cout << "2. Area of a rectangle\n";
   cout << "3. Area of a cylinder\n"
   cout << "4. None of them!\n";
   cin  >> selection;
   if (selection == 1)
      cout << "Pi times radius squared\n";
   else if (selection == 2)
      cout << "Length times width\n";
   else if (selection == 3)
      cout << "Pi times radius squared times height\n";
   else if (selection == 4)
      cout << "Well okay then, good-bye!\n";
   else
      cout << "Not good with numbers, eh?\n";
   return 0;
}
```

4.13 Enumerated Data Types

CONCEPT: An enumerated data type in C++ is a data type whose legal values are a set of named constant integers.

So far we have used data types that are built into the C++ language, such as `int` and `double`, and object types, like string, which are provided by C++ classes. However, C++ also allows programmers to create their own data types. An *enumerated data type* is a programmer-defined data type that contains a set of named integer constants. Here is an example of an enumerated type declaration.

```
enum Roster { Tom, Sharon, Bill, Teresa, John };
```

This creates a data type named `Roster`. It is called an enumerated type because the legal set of values that variables of this data type can have are enumerated, or listed, as part of the declaration. A variable of the `Roster` data type may only have values that are in the list inside the braces.

It is important to realize that the example `enum` statement does not actually create any variables—it just defines the data type. It says that when we later create variables of this data type, this is what they will look like—integers whose values are limited to the integers associated with the symbolic names in the enumerated set. The following statement shows how a variable of the `Roster` data type would be defined.

```
Roster student;
```

The form of this statement is like any other variable definition: first the data type name, then the variable name. Notice that the data type name is `Roster`, not `enum Roster`.

Because `student` is a variable of the `Roster` data type, we may store any of the values `Tom`, `Sharon`, `Bill`, `Teresa`, or `John` in it. An assignment operation would look like this:

```
student = Sharon;
```

The value of the variable could then be tested like this:

```
if (student == Sharon)
```

Notice in the two examples that there are no quotation marks around `Sharon`. It is a named constant, not a string literal.

In Chapter 3 you learned that named constants are constant values that are accessed through their symbolic name. So what is the value of `Sharon`? The symbol `Tom` is stored as the integer 0. `Sharon` is stored as the integer 1. `Bill` is stored as the integer 2, and so forth.

Even though the values in an enumerated data type are actually stored as integers, you cannot always substitute the integer value for the symbolic name. For example, assuming that `student` is a variable of the `Roster` data type, the following assignment statement is illegal.

```
student = 2;    // Error!
```

You can, however, test an enumerated variable by using an integer value instead of a symbolic name. For example, the following two `if` statements are equivalent.

```
if (student == Bill)
if (student == 2)
```

You can also use relational operators to compare two enumerated variables. For example, the following `if` statement determines if the value stored in `student1` is less than the value stored in `student2`:

```
if (student1 < student2)
```

If `student1` equals `Bill` and `student2` equals `John`, this statement would be true. However, if `student1` equals `Bill` and `student2` equals `Sharon`, the statement would be false.

By default, the symbols in the enumeration list are assigned the integer values 0, 1, 2, and so forth. If this is not appropriate, you can specify the values to be assigned, as in the following example.

```
enum Department { factory = 1, sales = 2, warehouse = 4 };
```

Remember that if you do assign values to the enumerated symbols, they must be integers. The following value assignments would produce an error.

```
enum Department { factory = 1.1, sales = 2.2, warehouse = 4.4 };
                                                    // Error!
```

While there is no requirement that assigned integer values be placed in ascending order, it is generally considered a good idea to do this.

If you leave out the value assignment for one or more of the symbols, it will be assigned a default value, as illustrated here:

```
enum Colors { red, orange, yellow = 9, green, blue };
```

`red` will be assigned the value 0, `orange` will be 1, `yellow` will be 9, `green` will be 10, and `blue` will be 11.

One of the purposes of an enumerated data type is that the symbolic names help to make a program self-documenting. However, because these names are not strings, they are for use inside the program only. Using the `Roster` data type in our example, the following two statements would output a `2`, not the name `Sharon`.

```
Roster student1 = Sharon;
cout << student1;
```

Because the symbolic names of an enumerated data type are associated with integer values, they may be used in a switch statement, as shown in Program 4-28. This program also demonstrates that it is possible to use an enumerated data type without actually creating any variables of that type.

Program 4-28

```
1 // This program demonstrates an enumerated data type.
2 #include <iostream>
3 using namespace std;
4
```

(program continues)

Program 4-28 *(continued)*

```
5  // Declare the enumerated type
6  enum Roster { Tom = 1, Sharon, Bill, Teresa, John };
7                    // Sharon – John will be assigned default values 2-5.
8  int main()
9  {
10    int who;
11
12    cout << "This program will give you a student's birthday.\n";
13    cout << "Whose birthday do you want to know?\n";
14    cout << "1 = Tom\n";
15    cout << "2 = Sharon\n";
16    cout << "3 = Bill\n";
17    cout << "4 = Teresa\n";
18    cout << "5 = John\n";
19    cin  >> who;
20
21    switch (who)
22    {
23       case Tom   :  cout << "\nTom's birthday is January 3.\n";
24                     break;
25       case Sharon:  cout << "\nSharon's birthday is April 22.\n";
26                     break;
27       case Bill  :  cout << "\nBill's birthday is December 19.\n";
28                     break;
29       case Teresa:  cout << "\nTeresa's birthday is February 2.\n";
30                     break;
31       case John  :  cout << "\nJohn's birthday is June 17.\n";
32                     break;
33       default    :  cout << "\nInvalid selection\n";
34    }
35    return 0;
36 }
```

Program Output with Example Input Shown in Bold

```
This program will give you a student's birthday.
Whose birthday do you want to know?
1 = Tom
2 = Sharon
3 = Bill
4 = Teresa
5 = John
2[Enter]

Sharon's birthday is April 22.
```



Checkpoint

4.39 Find all the things that are wrong with the following declaration.

```
Enum Pet = { "dog", "cat", "bird", "fish" }
```

4.40 Follow the instructions to complete the following program segment.

```
enum Paint { red, blue, yellow, green, orange, purple };
Paint color = green;

// Write an if/else statement that will print out "primary color"
// if color is red, blue, or yellow, and will print out
// "mixed color" otherwise. The if test should use a relational
// expression.
```

4.14 Testing for File Open Errors

CONCEPT: **When opening a file you can test the file stream object to determine if an error occurred.**

In Chapter 3 you were introduced to file operations and saw that the file stream member function `open` is used to open a file. Sometimes the `open` member function will not work. For example, the following code will fail if the file `info.txt` does not exist:

```
ifstream inputFile;
inputFile.open("info.txt");
```

You can determine when a file has failed to open by testing the value of the file stream object with the `!` operator. The following program segment attempts to open the file `customers.txt`. If the file cannot be opened, an error message is displayed.

```
ifstream inputFile;
inputFile.open("customers.txt");
if (!inputFile)
{
   cout << "Error opening file.\n";
}
```

Another way to detect a failed attempt to open a file is with the `fail` member function, as shown in the following code.

```
ifstream inputFile;
inputFile.open("customers.txt");
if (inputFile.fail())
{
   cout << "The customer.txt file could not be opened.\n"
        << "Make sure it is located in the default directory\n"
        << "where your program expects to find it.\n";
}
```

The `fail` member function returns `true` whenever an attempted file operation is unsuccessful. When using file I/O, you should always test the file stream object to make sure the file was opened successfully. If the file could not be opened, the user should be informed and appropriate action taken by the program.

4.15 Focus on Testing and Debugging: *Validating Output Results*

CONCEPT: **When testing a newly created or modified program, the output it produces must be carefully examined to ensure it is correct.**

Once a program being developed has been designed, written in a programming language, and found to compile and link without errors, it is easy to jump to the conclusion that it works correctly. This is especially true if it runs without aborting and produces "reasonable" output. However, just because a program runs and produces output does not mean that it is correct. It may still contain logic errors that cause the output to be incorrect. To determine if a program actually works correctly it must be tested with data whose output can be predicted and the output examined to ensure it is accurate.

Program 4-29 runs and produces output that may initially appear reasonable. However, it contains a bug that causes it to produce incorrect output.

Program 4-29

```
1  // This program determines total buffet luncheon cost when
2  // the number of guests and the per person cost are known.
3  // It contains a logic error.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  const int ADULT_MEAL_COST = 6.25; // Child meal cost = 75% of this
9
10 int main()
11 {
12    int      numAdults,         // Number of guests ages 12 and older
13             numChildren;       // Number of guests ages 2-11
14    double   adultMealTotal,    // Cost for all adult meals
15             childMealTotal,    // Cost for all child meals
16             totalMealCost;
17
18    // Get number of adults and children attending
19    cout << "This program calculates total cost "
20         << "for a buffet luncheon.\n";
21    cout << "Enter the number of adult guests (age 12 and over): ";
22    cin  >> numAdults;
23    cout << "Enter the number of child guests (age 2-11): ";
24    cin  >> numChildren;
25
26    // Calculate meal costs
27    adultMealTotal = numAdults * ADULT_MEAL_COST;
28    childMealTotal = numChildren * ADULT_MEAL_COST * .75;
29    totalMealCost  = adultMealTotal + childMealTotal;
```

(program continues)

Program 4-29 *(continued)*

```
30
31    // Display total meal cost
32    cout << fixed << showpoint << setprecision(2);
33    cout << "\nTotal buffet cost is $" << totalMealCost << endl;
34    return 0;
35 }
```

Program Output with Example Input Shown in Bold

```
This program calculates total cost for a buffet luncheon.
Enter the number of adult guests (age 12 and over): 92[Enter]
Enter the number of child guests (age 2-11): 4[Enter]

Total buffet cost is $570.00
```

At first glance the program may appear to run correctly. The per person charge for adults is $6.25, so if there were 100 adult guests the price would be $625. But there are only 96 guests and four of them are children, so it should cost less. $570 sounds "about right".

However, "about right" is not an a sufficient test of accuracy. If the program had been run with data whose output could have been more easily checked, the programmer would have quickly seen that there is an error. Here is the output from two more runs of the same program using more carefully selected sample data.

Program Output with Different Example Input Shown in Bold

```
This program calculates total cost for a buffet luncheon.
Enter the number of adult guests (age 12 and over): 1[Enter]
Enter the number of child guests (age 2-11): 0[Enter]

Total buffet cost is $6.00
```

Program Output with Still Different Example Input Shown in Bold

```
This program calculates total cost for a buffet luncheon.
Enter the number of adult guests (age 12 and over): 0[Enter]
Enter the number of child guests (age 2-11): 1[Enter]

Total buffet cost is $4.50
```

From this output we can see that the cost of a child meal is correctly being calculated as 75% of the cost of an adult meal, but the adult meal cost is wrong. For one adult, it is coming out as $6.00, when it should have been $6.25.

To find the problem, the programmer should determine which lines of code are most apt to have caused the problem. Most likely something is wrong either in the initialization or storage of `ADULT_MEAL_COST` on line 8, in the calculation or storage of `adultMealTotal` or `totalMealCost` on lines 14, 16, 27, and 29 or in the printing of `totalMealCost` on line 33. Because the cost for one adult meal is erroneously coming out as a whole dollar amount, even though it is formatted to appear as a floating-point number, one of the things to check is whether all the variables that need to hold floating-point values have been defined as type `float` or `double`. Sure enough, although `adultMealTotal` and `totalMealCost`

have each been defined as a `double`, the named constant `ADULT_MEAL_COST` has been defined to be an `int`. So the 6.25 with which it is initialized is truncated to 6 when it is stored. When the definition of this named constant is rewritten as

```
const double ADULT_MEAL_COST = 6.25;
```

and the program is rerun, we get the following results.

Output of Revised Program with Example Input Shown in Bold

```
This program calculates total cost for a buffet luncheon.
Enter the number of adult guests (age 12 and over): 1[Enter]
Enter the number of child guests (age 2-11): 0[Enter]

Total buffet cost is $6.25
```

Now that this error has been found and fixed, the program is correct. However, additional testing with carefully developed test cases should be used to confirm this. The topic of how to develop good test cases will be dealt with further in the next chapter.

4.16 Green Fields Landscaping Case Study—Part 2

Problem Statement

Another of the services provided by Green Fields Landscaping is the sale of evergreen trees, which are priced by height. Customers have the choice of purchasing a tree on a "cash and carry" basis, of purchasing a tree and having it delivered, or of purchasing a tree and having it both delivered and planted. You have been asked to develop a program that uses the number of trees purchased, their height, and the delivery and planting information to create a customer invoice. Assume for now that all trees purchased are the same height.

Table 4-12 Evergreen Tree Pricing Information

Under 3 feet tall	39.00 (tax included)
3 to 5 feet tall	69.00 (tax included)
6 to 8 feet tall	99.00 (tax included)
over 8 feet tall	199.00 (tax included)
delivery only (per tree)	20.00 (100.00 max. per order)
delivery + planting	50% of the cost of the tree

Program Design

Program Steps

The program must carry out the following general steps:

1. Have the user input the number of trees purchased and their height.
2. Have the user indicate if the trees will be planted by Green Fields.
3. If planting service is not desired, have the user indicate if they want delivery.
4. Calculate the total tree cost.
5. Calculate the planting and delivery charges.
6. Calculate the total of all charges.
7. Print a bill that displays the purchase information and all charges.

Named constants

```
double PRICE_1 =  39.00
double PRICE_2 =  69.00
double PRICE_3 =  99.00
double PRICE_4 = 199.00
double PER_TREE_DELIVERY = 20.00
double MAX_DELIVERY = 100.00
```

Variables whose values will be input

```
int  numTrees            // Number of evergreen trees purchased
int  height              // Tree height to the nearest foot
char planted             // Are trees to be planted?('Y'/'N')
char delivered           // Are trees to be delivered?('Y'/'N')
```

Variables whose values will be output

```
double treeCost          // Cost of each tree
double totalTreeCost     // Total price for all the trees
double deliveryCost      // Delivery cost for all the trees
double plantingCost      // Planting cost for all the trees
double totalCharges      // Total invoice amount
```

Detailed Pseudocode (including actual variable names and needed calculations)

```
Initialize deliveryCost and plantingCost to 0
Display screen heading
Input numTrees, height, planted
If planted = 'N'
   Input delivery
End If
If height < 3
   treeCost = PRICE_1
Else If height <= 5
   treeCost = PRICE_2
Else If height <= 8
   treeCost = PRICE_3
Else
   treeCost = PRICE_4
End If
totalTreeCost = numTrees * treeCost
If planted = 'Y'
   plantingCost = totalTreeCost / 2     // deliveryCost stays 0
Else If delivered = 'Y'
   If numTrees <= 5
      deliveryCost = PER_TREE_DELIVERY * numTrees
   Else
      deliveryCost = MAX_DELIVERY
   End If
End If
totalCharges = totalTreeCost + deliveryCost + plantingCost
Display invoice heading
Display numTrees, treeCost, totalTreeCost,
        deliveryCost, plantingCost, totalCharges
```

The Program

The next step, after the pseudocode has been checked for logic errors, is to expand the pseudocode into the final program. This is shown in Program 4-30.

Program 4-30

```
// This program is used by Green Fields Landscaping to create
// customer invoices for evergreen tree sales.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const double PRICE_1 =  39.00,   // Set prices for different
                PRICE_2 =  69.00,   // size trees
                PRICE_3 =  99.00,
                PRICE_4 = 199.00;

   const double PER_TREE_DELIVERY = 20.00, // Set delivery fees
                MAX_DELIVERY = 100.00;

   int    numTrees,           // Number of evergreen trees purchased
          height;             // Tree height to the nearest foot
   char   planted,            // Are trees to be planted?('Y'/'N')
          delivered;          // Are trees to be delivered?('Y'/'N')
   double treeCost,           // Cost of each tree
          totalTreeCost,      // Total price for all the trees
          deliveryCost = 0.0, // Delivery cost for all the trees
          plantingCost = 0.0, // Planting cost for all the trees
          totalCharges;       // Total invoice amount

   // Display purchase screen and get purchase information
   cout << "           Green Fields Landscaping\n"
        << "           Evergreen Tree Purchase\n\n";
   cout << "Number of trees purchased: ";
   cin  >> numTrees;
   cout << "Tree height to the nearest foot: ";
   cin  >> height;
   cout << "Will Green Fields do the planting?(Y/N): ";
   cin  >> planted;

   if (!(planted == 'Y' || planted == 'y'))
   {   cout << "Do you want the trees delivered?  (Y/N): ";
       cin  >> delivered;
   }
```

(program continues)

Program 4-30 *(continued)*

```
41
42    // Calculate costs
43    if (height < 3)
44       treeCost = PRICE_1;
45    else if(height <= 5)
46       treeCost = PRICE_2;
47    else if(height <= 8)
48       treeCost = PRICE_3;
49    else
50       treeCost = PRICE_4;
51
52    totalTreeCost = numTrees * treeCost;
53
54    if ((planted == 'Y') || (planted == 'y'))
55       plantingCost = totalTreeCost / 2;
56    else if((delivered == 'Y') || (delivered == 'y'))
57       if (numTrees <= 5)
58          deliveryCost = PER_TREE_DELIVERY * numTrees;
59       else
60          deliveryCost = MAX_DELIVERY;
61    //else planting and delivery costs both remain 0.0
62
63    totalCharges = totalTreeCost + deliveryCost + plantingCost;
64
65    // Display information on the invoice
66    cout << fixed << showpoint << setprecision(2);
67    cout << "\n\n           Green Fields Landscaping\n"
68         << "           Evergreen Tree Purchase\n\n";
69    cout << setw(2) << numTrees << " trees @ $" << setw(6) << treeCost
70         << " each =    $" << setw(8) << totalTreeCost << endl;
71    cout << "Delivery charge              $"
72         << setw(8) << deliveryCost << endl;
73    cout << "Planting charge              $"
74         << setw(8) << plantingCost << endl;
75    cout << "                                 ________" << endl;
76    cout << "Total Amount Due             $"
77         << setw(8) << totalCharges << endl << endl;
78    return 0;
79 }
```

Program Output with Example Input Shown in Bold

```
           Green Fields Landscaping
           Evergreen Tree Purchase

Number of trees purchased: 4[Enter]
Tree height to the nearest foot: 7[Enter]
Will Green Fields do the planting?(Y/N): y[Enter]
```

(program continues)

Program 4-30 *(continued)*

```
            Green Fields Landscaping
            Evergreen Tree Purchase

 4 trees @ $ 99.00 each =     $  396.00
Delivery charge               $    0.00
Planting charge               $  198.00

Total Amount Due              $  594.00
```

Crazy Al's Computer Emporium Case Study

The following additional case study, which contain applications of material introduced in Chapter 4, can be found on the student CD.

Crazy Al's is a retail seller of home computers whose sales staff work on commission. The commission rate varies depending on the amount of sales. This case study develops a program that computes monthly sales commission and then subtracts any pay already advanced to the salesperson to calculate how much remaining pay is due at the end of the month. The case study, which employs branching logic to determine the correct commission rate, includes problem definition, general and detailed pseudocode design, and a final running program with sample output.

4.17 Tying It All Together: *Fortune Teller*

With the `rand()` function you learned about in Chapter 3 and the `if/else if` statement you learned about in this chapter, you can now create a simple fortune telling game. Your program will start by asking users to enter three careers they would like to have some day. The program will then use random numbers to predict their future.

Program 4-31

```
1  // This program predicts the player's future using
2  // random numbers and an if/else if statement.
3  #include <iostream>
4  #include <string>           // Needed to use strings
5  #include <cstdlib>          // Needed for random numbers
6  using namespace std;
7
8  int main()
9  {
10    // Strings to hold user entered careers
11    string career1, career2, career3;
12
13    int randomNum;    // Will hold the randomly generated integer
14
```

(program continues)

Program 4-31 *(continued)*

```
15    // "Seed" the random generator
16    unsigned seed = time(0);
17    srand(seed);
18
19    // Explain the game and get the player's career choices
20    cout << "I am a fortune teller. Look into my crystal screen \n"
21         << "and enter 3 careers you would like to have. Example: \n\n"
22         << "      chef \n      astronaut \n      CIA agent \n\n"
23         << "Then I will predict what you will be. \n\n";
24
25    cout << "Career choice 1: ";
26    getline(cin, career1);
27    cout << "Career choice 2: ";
28    getline(cin, career2);
29    cout << "Career choice 3: ";
30    getline(cin, career3);
31
32    // Randomly generate an integer between 1 and 4.
33    randomNum = 1 + rand() % 4;
34
35    // Use branching logic to output the prediction
36    if (randomNum == 1)
37       cout << "\nYou will be a " << career1 << ". \n";
38    else if (randomNum == 2)
39       cout << "\nYou will be a " << career2 << ". \n";
40    else if (randomNum == 3)
41       cout << "\nYou will be a " << career3 << ". \n";
42    else
43       cout << "\nSorry. You will not be any of these. \n";
44    return 0;
45 }
```

Sample Run with User Input Shown in Bold

```
I am a fortune teller. Look into my crystal screen
and enter 3 careers you would like to have. For example,

      chef
      astronaut
      CIA agent

Then I will predict what you will be.

Career choice 1: radio announcer[Enter]
Career choice 2: sky diving instructor[Enter]
Career choice 3: circus clown[Enter]

You will be a radio announcer.
```

Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. An expression using the greater-than, less-than, greater-than-or-equal-to, less-than-or-equal-to, equal, or not-equal operator is called a(n) _____ expression.
2. The value of a relational expression is 0 if the expression is _____ or 1 if the expression is _____.
3. The `if` statement regards an expression with the value 0 as _____ and an expression with a nonzero value as _____.
4. For an `if` statement to conditionally execute a group of statements, the statements must be enclosed in a set of _____.
5. In an `if/else` statement, the `if` part executes its statement(s) if the expression is _____, and the `else` part executes its statement(s) if the expression is _____.
6. The trailing `else` in an `if/else if` statement has a similar purpose as the _____ section of a `switch` statement.
7. If the sub-expression on the left of the `&&` logical operator is _____, the right sub-expression is not checked.
8. If the sub-expression on the left of the `||` logical operator is _____, the right sub-expression is not checked.
9. The _____ logical operator has higher precedence than the other logical operators.
10. Logical operators have _____ precedence than relational operators.
11. The _____ logical operator works best when testing a number to determine if it is within a range.
12. The _____ logical operator works best when testing a number to determine if it is outside a range.
13. A variable with _____ scope is only visible when the program is executing in the block containing the variable's definition.
14. The expression that follows the `switch` statement must have a(n) _____ value.
15. A program will "fall through" to the following `case` section if it is missing the _____ statement.
16. What value will be stored in the variable `t` after each of the following statements executes?

    A) `t = (12 > 1);`_____
    B) `t = (2 < 0);`_____
    C) `t = (5 == (3 * 2));`_____
    D) `t = (5 == 5);`_____

17. Write an `if` statement that assigns 100 to `x` when `y` is equal to 0.
18. Write an `if/else` statement that assigns 0 to `x` when `y` is equal to 10. Otherwise it should assign 1 to `x`.

19. Write an `if/else` statement that prints "Excellent" when `score` is 90 or higher, "Good" when `score` is between 80 and 89, and "Try Harder" when `score` is less than 80.
20. Write an `if` statement that sets the variable `hours` to 10 when the flag variable `minimum` is set.
21. Convert the following conditional expression into an `if/else` statement.

```
q = (x < y) ? (a + b) : (x * 2);
```

22. Convert the following `if/else if` statement into a `switch` statement:

```
if (choice == 1)
{
   cout << fixed << showpoint << setprecision(2);
}
else if ((choice == 2) || (choice == 3))
{
   cout << fixed << showpoint << setprecision(4);
}
else if (choice == 4)
{
   cout << fixed << showpoint << setprecision(6);
}
else
{
   cout << fixed << showpoint << setprecision(8);
}
```

23. Assume the variables `x = 5`, `y = 6`, and `z = 8`. Indicate if each of the following conditions is true or false:

    A) `(x == 5) || (y > 3)`
    B) `(7 <= x) && (z > 4)`
    C) `(2 != y) && (z != 4)`

24. Assume the variables `x = 5`, `y = 6`, and `z = 8`. Indicate if each of the following conditions is true or false:

    A) `(x >= 0) || (x <= y)`
    B) `(z - y) > y`
    C) `!((z - y) > x)`

Algorithm Workbench

25. Write a C++ statement that prints the message "The number is valid." if the variable `grade` is within the range 0 through 100.
26. Write a C++ statement that prints the message "The number is valid." if the variable `temperature` is within the range –50 through 150.
27. Write a C++ statement that prints the message "The number is not valid." if the variable `hours` is outside the range 0 through 80.
28. Write a C++ statement that displays the strings `title1` and `title2` in alphabetical order.

29. Using the following chart , write a C++ statement that assigns .10, .15, or .20 to `commission`, depending on the value in `sales`.

Sales	Commission Rate
Up to $10,000	10%
$10,000 to $15,000	15%
Over $15,000	20%

30. Write one or more C++ statements that assign the correct value to `discount`, using the logic described here:

    Assign .20 to `discount` if `dept` equals 5 and `price` is $100 or more.
    Assign .15 to `discount` if `dept` is anything else and `price` is $100 or more.
    Assign .10 to `discount` if `dept` equals 5 and `price` is less than $100.
    Assign .05 to `discount` if `dept` is anything else and `price` is less than $100.

31. The following statement should determine if `x` is not greater than 20. What is wrong with it?

```
if (!x > 20)
```

32. The following statement should determine if `count` is within the range of 0 through 100. What is wrong with it?

```
if (count >= 0 || count <= 100)
```

33. The following statement should determine if `count` is outside the range of 0 through 100. What is wrong with it?

```
if (count < 0 && count > 100)
```

34. The following statement should determine if `x` has a value other than 1 or 2. What is wrong with it?

```
if (x! = 1 || x! = 2)
```

Find the Errors

35. Each of the following program segments has errors. Find as many as you can.

A)
```
cout << "Enter your 3 test scores and I will ";
     << "average them:";
int score1, score2, score3,
cin  >> score1 >> score2 >> score3;

double average;
average = (score1 + score2 + score3) / 3.0;
if (average = 100);
   perfectScore = true;// Set the flag variable
cout << "Your average is " << average << endl;
bool perfectScore;
if (perfectScore);
{
   cout << "Congratulations!\n";
   cout << "That's a perfect score.\n";
   cout << "You deserve a pat on the back!\n";
```

B)
```
double num1, num2, quotient;

cout << "Enter a number: ";
cin  >> num1;
cout << "Enter another number: ";
cin  >> num2;

if (num2 == 0)
   cout << "Division by zero is not possible.\n";
   cout << "Please run the program again ";
   cout << "and enter a number besides zero.\n";
else
   quotient = num1 / num2;
   cout << "The quotient of " << num1 <<
   cout << " divided by " << num2 << " is ";
   cout << quotient << endl;
```

C)
```
int testScore;

cout << "Enter your test score and I will tell you\n";
cout << "the letter grade you earned: ";
cin  >> testScore;

if (testScore < 60)
   cout << "Your grade is F.\n";
else if (testScore < 70)
   cout << "Your grade is D.\n";
else if (testScore < 80)
   cout << "Your grade is C.\n";
else if (testScore < 90)
   cout << "Your grade is B.\n";
else
   cout << "That is not a valid score.\n";
else if (testScore <= 100)
   cout << "Your grade is A.\n";
```

D)
```
double testScore;

cout << "Enter your test score and I will tell you\n";
cout << "the letter grade you earned: ";
cin  >> testScore;

switch (testScore)
{  case (testScore < 60.0):
              cout << "Your grade is F.\n";
              break;
   case (testScore < 70.0):
              cout << "Your grade is D.\n";
              break;
   case (testScore < 80.0):
              cout << "Your grade is C.\n";
              break;
   case (testScore < 90.0):
              cout << "Your grade is B.\n";
              break;
   case (testScore <= 100.0):
              cout << "Your grade is A.\n";
              break;
   default:  cout << "That score isn't valid\n"; }
```

Soft Skills

Programmers need to be able to look at alternative approaches to solving a problem and at different ways of implementing a solution, weighing the pros and cons of each. Further, they need to be able to clearly articulate to others why they recommend, or have chosen, a particular solution. Come to class prepared to discuss the following:

36. Sometimes either a `switch` statement or an `if/else if` statement can be used to implement logic that requires branching to different blocks of program code. But the two are not interchangeable.
    A) Under what circumstances would an `if/else if` statement be a more appropriate choice than a switch statement?
    B) Under what circumstances would a `switch` statement be a more appropriate choice than an `if/else if` statement?
    C) Under what circumstances would a set of nested `if/else` statements be more appropriate than either of the other two structures?

Try to come up with at least one example case for each of the three, where it is the best way to implement the desired branching logic.

Programming Challenges

1. Minimum/Maximum

Write a program that asks the user to enter two numbers. The program should use the conditional operator to determine which number is the smaller and which is the larger.

2. Roman Numeral Converter

Write a program that asks the user to enter a number within the range of 1 through 10. Use a switch statement to display the Roman numeral version of that number.

Input Validation: Do not accept a number less than 1 or greater than 10.

3. Magic Dates

The date June 10, 1960, is special because when we write it in the following format, the month times the day equals the year.

6/10/60

Write a program that asks the user to enter a month (in numeric form), a day, and a two-digit year. The program should then determine whether the month times the day is equal to the year. If so, it should display a message saying the date is magic. Otherwise, it should display a message saying the date is not magic.

4. Areas of Rectangles

The area of a rectangle is the rectangle's length times its width. Write a program that asks for the length and width of two rectangles. The program should tell the user which rectangle has the greater area, or if the areas are the same.

5. Book Club Points

An online book club awards points to its customers based on the number of books purchased each month. Points are awarded as follows:

Books Purchased	Points Earned
0	0
1	5
2	15
3	30
4 or more	60

Write a program that asks the user to enter the number of books purchased this month and then displays the number of points awarded.

6. Mass and Weight

Scientists measure an object's mass in kilograms and its weight in newtons. If you know an object's mass, you can calculate its weight in newtons with the following formula:

$$\text{weight} = \text{mass} \times 9.8$$

Write a program that asks the user to enter an object's mass, and then calculates and displays its weight. If the object weighs more than 1000 newtons, display a message indicating that it is too heavy. If the object weighs less than 10 newtons, display a message indicating that the object is too light.

7. Time Calculator



VideoNote
Solving the Time Calculator Problem

Write a program that asks the user to enter a number of seconds.

- There are 86400 seconds in a day. If the number of seconds entered by the user is greater than or equal to 86400, the program should display the number of days in that many seconds.
- There are 3600 seconds in an hour. If the number of seconds entered by the user is less than 86400, but is greater than or equal to 3600, the program should display the number of hours in that many seconds.
- There are 60 seconds in a minute. If the number of seconds entered by the user is less than 3600, but is greater than or equal to 60, the program should display the number of minutes in that many seconds.

8. Math Tutor

This is a modification of the math tutor problem in Chapter 3. Write a program that can be used as a math tutor for a young student. The program should display two random numbers between 10 and 50 that are to be added, such as:

```
  24
+ 12
  ——
```

The program should then wait for the student to enter the answer. If the answer is correct, a message of congratulations should be printed. If the answer is incorrect, a message should be printed showing the correct answer.



9. Software Sales

A software company sells a package that retails for $99. Quantity discounts are given according to the following table.

Quantity	Discount
10–19	20%
20–49	30%
50–99	40%
100 or more	50%

Write a program that asks for the number of units purchased and computes the total cost of the purchase.

Input Validation: Make sure the number of units is greater than 0.

10. Bank Charges

A bank charges $10 per month plus the following check fees for a commercial checking account:

$.10 each for fewer than 20 checks
$.08 each for 20–39 checks
$.06 each for 40–59 checks
$.04 each for 60 or more checks

Write a program that asks for the number of checks written during the past month, then computes and displays the bank's fees for the month.

Input Validation: Do not accept a negative value for the number of checks written.

11. Geometry Calculator

Write a program that displays the following menu:

```
Geometry Calculator

   1. Calculate the Area of a Circle
   2. Calculate the Area of a Rectangle
   3. Calculate the Area of a Triangle
   4. Quit

Enter your choice (1-4):
```

If the user enters 1, the program should ask for the radius of the circle and then display its area. Use 3.14159 for π. If the user enters 2, the program should ask for the length and width of the rectangle, and then display the rectangle's area. If the user enters 3, the program should ask for the length of the triangle's base and its height, and then display its area. If the user enters 4, the program should end.

Input Validation: Display an error message if the user enters a number outside the range of 1 through 4 when selecting an item from the menu. Do not accept negative values for the circle's radius, the rectangle's length or width, or the triangle's base or height.



12. Running the Race

Write a program that asks for the names of three runners and the time it took each of them to finish a race. The program should display who came in first, second, and third place. Think about how many test cases are needed to verify that your problem works correctly. (That is, how many different finish orders are possible?)

Input Validation: Only accept positive numbers for the times.

13. Personal Best

Write a program that asks for the name of a pole vaulter and the dates and vault heights (in meters) of the athlete's three best vaults. It should then report in height order (best first), the date on which each vault was made, and its height.

Input Validation: Only accept values between 2.0 and 5.0 for the heights.

14. Body Mass Index

Write a program that calculates and displays a person's body mass index (BMI). The BMI is often used to determine whether a person with a sedentary lifestyle is overweight or underweight for his or her height. A person's BMI is calculated with the following formula:

$$\text{BMI} = \text{weight} \times 703/\text{height}^2$$

where weight is measured in pounds and height is measured in inches. The program should display a message indicating whether the person has optimal weight, is underweight, or is overweight. A sedentary person's weight is considered to be optimal if his or her BMI is between 18.5 and 25. If the BMI is less than 18.5, the person is considered to be underweight. If the BMI value is greater than 25, the person is considered to be overweight.

15. Fat Gram Calculator

Write a program that asks for the number of calories and fat grams in a food. The program should display the percentage of calories that come from fat. If the calories from fat are less than 30 percent of the total calories of the food, it should also display a message indicating the food is low in fat.

One gram of fat has 9 calories, so

```
Calories from fat = fat grams * 9
```

The percentage of calories from fat can be calculated as

```
Calories from fat ÷ total calories
```

Input Validation: Make sure the number of calories is greater than 0 and the number of fat grams is 0 or more. Also, the number of calories from fat cannot be greater than the total number of calories. If that happens, display an error message indicating that either the calories or fat grams were incorrectly entered.



16. The Speed of Sound

The speed of sound varies depending on the medium through which it travels. In general, sound travels fastest in rigid media, such as steel, slower in liquid media, such as water, and slowest of all in gases, such as air. The following table shows the approximate speed of sound, measured in feet per second, in air, water, and steel.

Medium	Speed (feet per sec.)
Air	1,100
Water	4,900
Steel	16,400

Write a program that displays a menu allowing the user to select air water, or steel. After the user has made a selection, the number of feet a sound wave will travel in the selected medium should be entered. The program will then display the amount of time it will take. (Round the answer to four decimal places.)

Input Validation: Check that the user has selected one of the available menu choices. Do not accept distances less than 0.

17. The Speed of Sound in Gases

When traveling through a gas, the speed of sound depends primarily on the density of the medium. The less dense the medium, the faster the speed will be. The following table shows the approximate speed of sound at 0 degree celsius, measured in meters per second, when traveling through carbon dioxide, air, helium, and hydrogen.

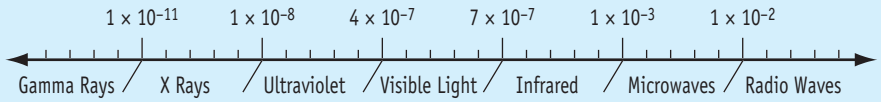
Medium	Speed (meters per sec.)
Carbon dioxide	258.0
Air	331.5
Helium	972.0
Hydrogen	1270.0

Write a program that displays a menu allowing the user to select one of these 4 gases. After a selection has been made, the user should enter the number of seconds it took for the sound to travel in this medium from its source to the location at which it was detected. The program should then report how far away (in meters) the source of the sound was from the detection location.

Input Validation: Check that the user has selected one of the available menu choices. Do not accept times less than 0 seconds or more than 30 seconds.

18. Spectral Analysis

If a scientist knows the wavelength of an electromagnetic wave she can determine what type of radiation it is. Write a program that asks for the wavelength in meters of an electromagnetic wave and then displays what that wave is according to the following chart. (For example, a wave with a wavelength of 1E-10 meters would be an X-ray.)



19. Long-Distance Calls

A long-distance carrier charges the following rates for telephone calls between the United States and Mexico:

Starting Time of Call	Rate per Minute
00:00–06:59	$0.12
07:00–19:00	0.55
19:01–23:59	0.35

Write a program that asks for the starting time and the number of minutes of the call, and displays the charges. The program should ask for the time to be entered as a floating-point number in the form HH.MM. For example, 07:00 hours will be entered as 07.00, and 16:28 hours will be entered as 16.28.

> *Input Validation: The program should not accept times that are greater than 23:59. Also, no number whose last two digits are greater than 59 should be accepted. Hint: Assuming* `num` *is a floating-point variable, the following expression will give you its fractional part:*
>
> ```
> num - static_cast<int>(num)
> ```

20. Freezing and Boiling Points

The following table lists the freezing and boiling points of several substances. Write a program that asks the user to enter a temperature, and then shows all the substances that will freeze at that temperature and all that will boil at that temperature. For example, if the user enters –20 the program should report that water will freeze and oxygen will boil at that temperature.

Substance	Freezing Point (°F)	Boiling Point (°F)
Ethyl alcohol	–173	172
Mercury	–38	676
Oxygen	–362	–306
Water	32	212

21. Using Files—Freezing and Boiling Points Modification

Modify the freezing and boiling points program described in Programming Challenge 20 so it reads its input from a file instead of from the keyboard. Perform the necessary test to determine if an error occurs when the file is opened. If an error occurs, display a message informing the user. The following data to test your program can be found in the `FrzBoil.dat` file.

```
-173   -38   -362    32
 172   676   -306   212
```

It is important that your program read the data from the file in the same order it is written in the file. Notice that the four freezing point values come first, followed by the four boiling point values.

22. Using Files—Internet Service Provider Part 1

An Internet service provider has three different subscription packages for its customers:

Package A: For $9.95 per month 10 hours of access are provided. Additional hours are $2.00 per hour.

Package B: For $14.95 per month 20 hours of access are provided. Additional hours are $1.00 per hour.

Package C: For $19.95 per month unlimited access is provided.

Write a program that calculates a customer's monthly bill. It should input customer name, which package the customer has purchased, and how many hours were used. It should then create a bill that includes the input information and the total amount due. The bill should be written to a file.

Input Validation: Be sure the user only selects package A, B, or C. Also, the number of hours used in a month cannot exceed 744.

23. Using Files—Internet Service Provider Part 2

Modify the program in problem 22 so it also displays how much money Package A customers would save if they purchased packages B or C, and how much money package B customers would save if they purchased package C. If there would be no savings, no message should be printed.

24. Using Files—Internet Service Provider Part 3

Months with 30 days have 720 hours, and months with 31 days have 744 hours. February, with 28 days, has 672 hours. Enhance the input validation of the Internet Service Provider program by asking the user for the month (by name), and validating that the number of hours entered is not more than the maximum for the entire month. Here is a table of the months, their days, and number of hours in each.

Month	Days	Hours
January, March, May, July, August, October, December	31	744
April, June, September, November	30	720
February	28	672

This page intentionally left blank

CHAPTER 5

Looping

TOPICS

5.1 The Increment and Decrement Operators
5.2 Introduction to Loops: The `while` Loop
5.3 Using the `while` Loop for Input Validation
5.4 Counters
5.5 The `do-while` Loop
5.6 The `for` Loop
5.7 Keeping a Running Total
5.8 Sentinels
5.9 Using a Loop to Read Data from a File
5.10 Focus on Software Engineering: *Deciding Which Loop to Use*
5.11 Nested Loops
5.12 Breaking Out of a Loop
5.13 The `continue` Statement
5.14 Focus on Testing and Debugging: *Creating Good Test Data*
5.15 Central Mountain Credit Union Case Study
5.16 Tying It All Together: *What a Colorful World*

5.1 The Increment and Decrement Operators

CONCEPT: **C++ provides a pair of unary operators for incrementing and decrementing variables.**

To *increment* a value means to increase it, and to *decrement* a value means to decrease it. In the example below, `qtyOrdered` is incremented by 10 and `numSold` is decremented by 3.

```
qtyOrdered = qtyOrdered + 10;
numSold  =  numSold - 3;
```

Although the values stored in variables can be increased or decreased by any amount, it is particularly common to increment them or decrement them by 1. In fact, if we say that

a value is being incremented or decremented without specifying by how much, it is understood that it is being incremented or decremented by 1. C++ provides a pair of unary operators to do this. The ++ operator increases its operand's value by 1. The –– operator decreases its operand's value by 1.

Here are three different ways to increment the value of the variable num by 1.

```
num = num + 1;
num += 1;
num++;
```

And here are three different ways to decrement it by 1:

```
num = num - 1;
num -= 1;
num--;
```



NOTE: The expression num++ is pronounced "num plus plus," and num-- is pronounced "num minus minus."

Our examples so far show the increment and decrement operators used in *postfix mode*, which means the operator is placed after the variable. The operators also work in *prefix mode*, where the operator is placed before the variable name:

```
++num;
--num;
```

In both prefix and postfix mode, these operators add 1 to, or subtract 1 from, their operand. The following example illustrates the use of these operators in both prefix and postfix mode. Notice that there is no space between the name of the variable and the ++ or –– preceding it or following it.

```
num = 4;
num++;      // now num has the value 5
++num;      // now num has the value 6
num--;      // now num has the value 5 again
--num;      // now num has the value 4 again
```

Program 5-1 includes these 5 lines of code along with cout statements to further illustrate how they work.

Program 5-1

```
1  // This program demonstrates the ++ and -- operators.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int num = 4;   // num starts out with 4
8
9     // Display the value in num
10    cout << "The variable num is " << num << endl;
11    cout << "I will now increment num.\n\n";
12
```

(program continues)

Program 5-1 *(continued)*

```
13    // Use postfix ++ to increment num
14    num++;
15    cout << "Now the variable num is " << num << endl;
16    cout << "I will increment num again.\n\n";
17
18    // Use prefix ++ to increment num
19    ++num;
20    cout << "Now the variable num is " << num << endl;
21    cout << "I will now decrement num.\n\n";
22
23    // Use postfix -- to decrement num
24    num--;
25    cout << "Now the variable num is " << num << endl;
26    cout << "I will decrement num again.\n\n";
27
28    // Use prefix -- to increment num
29    --num;
30    cout << "Now the variable num is " << num << endl;
31    return 0;
32 }
```

Program Output

```
The variable num is 4
I will now increment num.

Now the variable num is 5
I will increment num again.

Now the variable num is 6
I will now decrement num.

Now the variable num is 5
I will decrement num again.

Now the variable num is 4
```

The Difference Between Postfix and Prefix Modes

In the simple statements used in Program 5-1, it doesn't matter if the increment or decrement operator is used in postfix or prefix mode. The difference is important, however, when these operators are used in statements that do more than just increment or decrement. For example, look at the following lines:

```
num = 4;
cout << num++;
```

This `cout` statement is doing two things: 1) displaying the value of `num`, and 2) incrementing `num`. But which happens first? `cout` will display a different value if `num` is incremented first than if `num` is incremented last. The answer depends on the mode of the increment operator.

Postfix mode causes the increment to happen after the value of the variable is used in the expression. In the example, `cout` will display 4, then `num` will be incremented to 5. Prefix

mode, however, causes the increment to happen first. In the following statements, num will first be incremented to 5, then cout will display 5:

```
num = 4;
cout << ++num;
```

Program 5-2 illustrates these dynamics further by placing increment and decrement operators in cout statements. This makes it easy to see the difference between using them in prefix and postfix mode. However, this should not normally be done. That is, in actual programming applications it is not recommended to place increment or decrement operators in cout statements. So you should not write code like this.

Program 5-2

```
1  // This program demonstrates the postfix and prefix
2  // modes of the increment and decrement operators.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int num = 4;
9
10    // Illustrate postfix and prefix ++ operator
11    cout << num   << "  ";       // Displays 4
12    cout << num++ << "  ";       // Displays 4, then adds 1 to num
13    cout << num   << "  ";       // Displays 5
14    cout << ++num << "\n\n";     // Adds 1 to num, then displays 6
15
16    // Illustrate postfix and prefix -- operator
17    cout << num   << "  ";       // Displays 6
18    cout << num-- << "  ";       // Displays 6, then subtracts 1 from num
19    cout << num   << "  ";       // Displays 5
20    cout << --num << "\n\n";     // Subtracts 1 from num, then displays 4
21
22    return 0;
23 }
```

Program Output

```
4  4  5  6

6  6  5  4
```

Let's analyze the statements in this program. In line 8, num is initialized with the value 4, so the cout statement in line 11 displays 4. Then, line 12 sends the expression num++ to cout. Because the ++ operator is used in postfix mode, the value 4 is first sent to cout, and then 1 is added to num, making its value 5.

When line 13 executes, num will hold the value 5, so 5 is displayed. Then, line 14 sends the expression ++num to cout. Because the ++ operator is used in prefix mode, 1 is first added to num (making it 6), and then the value 6 is sent to cout. This same sequence of events happens in lines 17 through 20, except the -- operator is used.

For another example, look at the following code:

```
int x = 1;
int y
y = x++;    // Postfix increment
```

The first statement defines the variable x (initialized with the value 1) and the second statement defines the variable y. The third statement does two things:

- It assigns the value of x to the variable y.
- The variable x is incremented.

The value that will be stored in y depends on when the increment takes place. Because the ++ operator is used in postfix mode, the old value of x (which is 1) is assigned to y before x is incremented. After the statement executes, y will contain 1, and x will contain 2. Let's look at the same code, but with the ++ operator used in prefix mode:

```
int x = 1;
int y;
y = ++x;    // Prefix increment
```

In the third statement, the ++ operator is used in prefix mode, causing variable x to be incremented before the assignment takes place. So, this code will store 2 in y. After the code has executed, x and y will both contain 2.

Using ++ and -- in Mathematical Expressions

The increment and decrement operators can also be used on variables in mathematical expressions. Consider the following program segment:

```
a = 2;
b = 5;
c = a * b++;
cout << a << " " << b << " " << c;
```

In the statement c = a * b++, c is assigned the value of a times b, which is 10. The variable b is then incremented. The cout statement will display

```
2 6 10
```

If the statement were changed to read

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

the variable b would be incremented before it was multiplied by a. In this case c would be assigned the value of 2 times 6, so the cout statement would display

```
2 6 12
```

You can pack a lot of action into a single statement using the increment and decrement operators, but don't get too tricky with them. You might be tempted to try something like the following, thinking that c will be assigned 11:

```
a = 2;
b = 5;
c = ++(a * b);    // Error!
```

But this assignment statement simply will not work because the operand of the increment and decrement operators must be an lvalue. Recall from Chapter 2 that an lvalue identifies a place in memory whose contents may be changed. The increment and decrement operators usually have variables for their operands, but generally speaking, anything that can go on the left side of an = operator is legal.

Using ++ and -- in Relational Expressions

The ++ and -- operators may also be used in relational expressions. Just as in mathematical expressions, the difference between postfix and prefix mode is critical. Consider the following program segment:

```
x = 10;
if (x++ > 10)
   cout << "x is greater than 10.\n";
```

Two operations are happening in this `if` statement: 1) The value in `x` is tested to determine if it is greater than 10, and 2) `x` is incremented. Because the increment operator is used in postfix mode, the comparison happens first. Since 10 is not greater than 10, the `cout` statement won't execute. If the mode of the increment operator is changed, however, the `if` statement will compare 11 to 10 and the `cout` statement will execute:

```
x = 10;
if (++x > 10)
   cout << "x is greater than 10.\n";
```



NOTE: Some instructors prefer that you only use the ++ and -- operators in statements whose sole purpose is to increment or decrement a variable. They may ask you not to use them in assignment statements, mathematical expressions, or relational expressions.



Checkpoint

5.1 What will the following program segments display?

A)
```
x = 2;
y = x++;
cout << x << y;
```

B)
```
x = 2;
y = ++x;
cout << x << y;
```

C)
```
x = 2;
y = 4;
cout << x++ << --y;
```

D)
```
x = 2;
y = 2 * x++;
cout << x << y;
```

E)
```
x = 99;
if (x++ < 100)
   cout "It is true!\n";
else
   cout << "It is false!\n";
```

F)
```
x = 0;
if (++x)
   cout << "It is true!\n";
else
   cout << "It is false!\n";
```

5.2 Introduction to Loops: The `while` Loop

CONCEPT: A loop is part of a program that repeats.



VideoNote
The `while` Loop

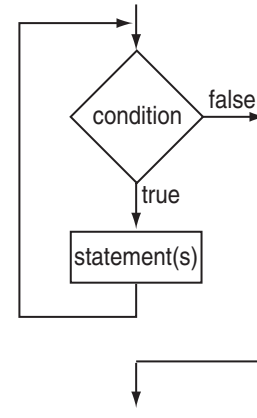
Chapter 4 introduced the concept of control structures, which direct the flow of a program. A *loop* is a control structure that causes a statement or group of statements to repeat. C++ has three looping control structures: the `while` loop, the `do-while` loop, and the `for` loop. The difference between each of these is how they control the repetition.

The `while` Loop

The `while` loop has two important parts: (1) an expression that is tested for a true or false value, and (2) a statement or block that is repeated as long as the expression is true. Figure 5-1 shows the general format of the `while` loop and a flowchart visually depicting how it works.

Figure 5-1

```
while (condition)
{
     statement;
     statement;
     // Place as many statements
     // here as necessary
}
```



Let's look at each part of the `while` loop. The first line, sometimes called the *loop header*, consists of the key word `while` followed by a *condition* to be tested enclosed in parentheses.

The condition is expressed by any expression that can be evaluated as true or false. Next comes the *body* of the loop. This contains one or more C++ *statements*.

Here's how the loop works. The condition expression is tested, and if it is true, each statement in the body of the loop is executed. Then, the condition is tested again. If it is still true, each statement is executed again. This cycle repeats until the condition is false.

Notice that, as with an `if` statement, each statement in the body to be conditionally executed ends with a semicolon, but there is no semicolon after the condition expression in parentheses. This is because the `while` loop is not complete without the statements that follow it. Also, as with an `if` statement, when the body of the loop contains two or more statements, these statements must be surrounded by braces. When the body of the loop contains only one statement, the braces may be omitted. Essentially, the `while` loop works like an `if` statement that can execute over and over. As long as the expression in the parentheses is true, the conditionally executed statements will repeat.

Program 5-3 uses a `while` loop to print "Hello" five times.

Program 5-3

```
1  // This program demonstrates a simple while loop.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int number = 1;
8
9     while (number <= 5)
10    {
11       cout << "Hello   ";
12       number++;
13    }
14    cout << "\nThat's all!\n";
15    return 0;
16 }
```

Program Output

```
Hello   Hello   Hello   Hello   Hello
That's all!
```

Let's take a closer look at this program. In line 7 an integer variable `number` is defined and initialized with the value 1. In line 9 the `while` loop begins with this statement:

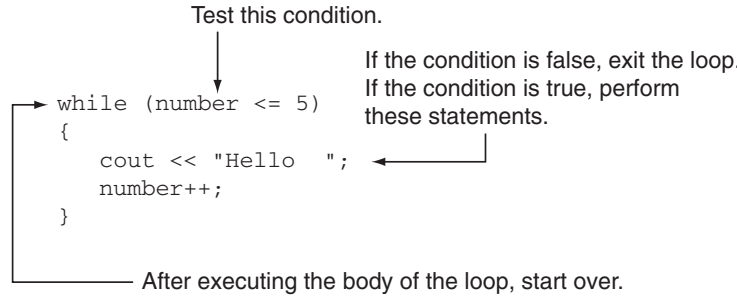
```
while (number <= 5)
```

This statement tests the variable `number` to determine whether its value is less than or equal to 5. Because it is, the statements in the body of the loop (lines 11 and 12) are executed:

```
cout << "Hello   ";
number++;
```

The statement in line 11 prints the word "Hello". The statement in line 12 uses the increment operator to add one to number, giving it the value 2. This is the last statement in the body of the loop, so after it executes the loop starts over. It tests the expression number <= 5 again, and because it is still true, the statements in the body of the loop are executed again. This cycle repeats until the value of number equals 6, making the expression number <= 5 false. Then the loop is exited. This is illustrated in Figure 5-2.

Figure 5-2



Each execution of a loop is known as an *iteration*. This loop will perform five iterations before the expression number <= 5 is tested and found to be false, causing the loop to terminate. The program then resumes execution at the statement immediately following the loop. A variable that controls the number of time a loop iterates is referred to as a *loop control variable*. In the example we have just seen, number is the loop control variable.

while is a Pretest Loop

The while loop is known as a *pretest* loop, which means it tests its expression before each iteration. Notice the variable definition of number in line 7 of Program 5-3:

```
int number = 1;
```

The number variable is initialized with the value 1. If number had been initialized with a value greater than 5, as shown in the following program segment, the loop would never execute:

```
int number = 6;
while (number <= 5)
{
   cout << "Hello   ";
   number++;
}
```

An important characteristic of the while loop is that the loop will never iterate if the test expression is false to start with. If you want to be sure a while loop executes the first time, you must initialize the relevant data in such a way that the test expression starts out as true.

Infinite Loops

In all but rare cases, loops must contain within themselves a way to terminate. This means that something inside the loop must eventually make the test expression false. The loop in Program 5-3 stops when the expressions `number <= 5` is false.

If a loop does not have a way of stopping, it is called an *infinite loop*. Infinite loops keep repeating until the program is interrupted. Here is an example:

```
int number = 1;
while (number <= 5)
{
   cout << "Hello   ";
}
```

This is an infinite loop because it does not contain a statement that changes the value of the `number` variable. Each time the expression `number <= 5` is tested, `number` will contain the value 1.

It's also possible to create an infinite loop by accidentally placing a semicolon after the first line of the `while` loop. Here is an example:

```
int number = 1;
while (number <= 5);  // This semicolon is an ERROR!
{
   cout << "Hello   ";
   number++;
}
```

The semicolon at the end of the first line is assumed to be a null statement and disconnects the `while` statement from the block that comes after it. To the compiler, this loop looks like this:

```
while (number <= 5);
```

This `while` loop will forever execute the null statement, which does nothing. The program will appear to have "gone into space" because there is nothing to display screen output or show any activity.

Don't Forget the Braces with a Block of Statements

If you write a loop that conditionally executes a block of statements, don't forget to enclose all of the statements in a set of braces. If the braces are accidentally left out, the `while` statement conditionally executes only the very next statement. For example, look at the following code.

```
int number = 1;
// This loop is missing its braces!
while (number <= 5)
   cout << "Hello   ";
   number++;
```

In this code the `number++` statement is not in the body of the loop. Because the braces are missing, the `while` statement only executes the statement that immediately follows it. This loop will execute infinitely because there is no code in its body that changes the `number` variable.

Another common pitfall with loops is accidentally using the = operator when you intend to use the == operator. The following is an infinite loop because the test expression assigns 1 to `remainder` each time it is evaluated rather than testing if remainder is equal to 1:

```
while (remainder = 1)   // Error: Notice the assignment.
{
   cout << "Enter a number: ";
   cin  >> num;
   remainder = num % 2;
}
```

Remember, any nonzero value is evaluated as true.

Programming Style and the `while` Loop

It's possible to create loops that look like this:

```
while (number <= 5) { cout << "Hello   "; number++; }
```

Avoid this style of programming, however. The programming layout style you should use with the `while` loop is similar to that of the `if` statement:

- If there is only one statement repeated by the loop, it should appear on the line after the `while` statement and be indented one level.
- If the loop repeats a block of statements, the block should begin on the line after the `while` statement and each line inside the braces should be indented.

In general, you'll find a similar layout style being used with the other types of loops presented in this chapter.

5.3 Using the `while` Loop for Input Validation

CONCEPT: **The `while` loop can be used to create input routines that repeat until acceptable data is entered.**

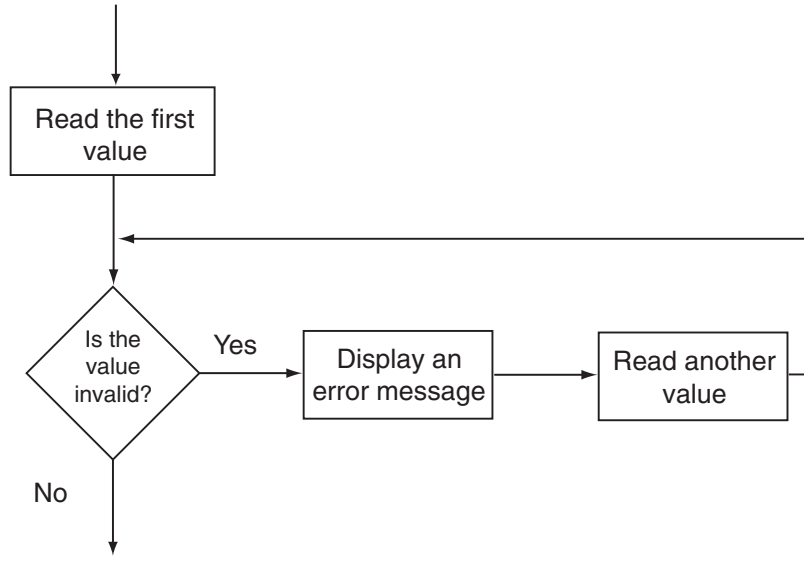
Perhaps the most famous saying of the computer industry is "garbage in, garbage out." The integrity of a program's output is only as good as its input, so you should try to make sure garbage does not go into your programs. *Input validation* is the process of inspecting data given to a program by the user and determining if it is valid. A good program should give clear instructions about the kind of input that is acceptable, and not assume the user has followed those instructions.

The `while` loop is especially useful for validating input. If an invalid value is entered, a loop can require that the user re-enter it as many times as necessary. For example, the following loop asks for a number in the range of 1 through 100:

```
cout << "Enter a number in the range 1 - 100: ";
cin  >> number;
while ((number < 1) || (number > 100))
{
   cout << "ERROR: Enter a value in the range 1 - 100: ";
   cin  >> number;
}
```

This code first allows the user to enter a number. This takes place just before the loop. If the input is valid, the loop will not execute. If the input is invalid, however, the statements in the body of the loop will be executed. They will display an error message and require the user to enter another number. The loop will continue to execute until the user enters a valid number. The general logic of performing input validation is shown in Figure 5-3.

Figure 5-3



The read operation that takes place just before the loop is called a *priming read*. It provides the first value for the loop to test. Subsequent values are obtained by the loop.

Program 5-4 calculates the number of soccer teams a youth league may create, based on a given number of players and a maximum number of players per team. The program uses `while` loops (in lines 20 through 25 and lines 31 through 35) to validate the user's input.

Program 5-4

```
1 // This program calculates the number of soccer teams a youth
2 // league may create from the number of available players.
3 // Input validation is done with while loops.
4 #include <iostream>
5 using namespace std;
```

(program continues)

Program 5-4 *(continued)*

```
 6
 7 int main()
 8 {
 9    int players,      // Number of available players
10        teamPlayers,  // Number of desired players per team
11        numTeams,     // Number of teams
12        leftOver;     // Number of players left over
13
14    // Get the number of players per team
15    cout << "How many players do you wish per team?\n";
16    cout << "(Enter a value in the range 9 - 15): ";
17    cin  >> teamPlayers;
18
19    // Validate the input
20    while ((teamPlayers) < 9) || (teamPlayers > 15))
21    {
22       cout << "Team size should be 9 to 15 players.\n";
23       cout << "How many players do you wish per team? ";
24       cin  >> teamPlayers;
25    }
26    // Get the number of players available
27    cout << "How many players are available? ";
28    cin  >> players;
29
30    // Validate the input
31    while (players <= 0)
32    {
33       cout << "Please enter a positive number: ";
34       cin  >> players;
35    }
36    // Calculate the number of teams
37    numTeams = players / teamPlayers;
38
39    // Calculate the number of leftover players
40    leftOver = players % teamPlayers;
41
42    // Display the results
43    cout << "\nThere will be " << numTeams << " teams with ";
44    cout << leftOver << " players left over.\n";
45    return 0;
46 }
```

Program Output with Example Input Shown in Bold

```
How many players do you wish per team?
(Enter a value in the range 9 - 15): 4[Enter]
Team size should be 9 to 15 players.
How many players do you wish per team? 12[Enter]
How many players are available? –142[Enter]
Please enter a positive number: 142[Enter]

There will be 11 teams with 10 players left over.
```

Checkpoint

5.2 How many times will "Sunday\n" be printed in each of the following program segments?

A)
```
int count = 1;
while (count < 5)
{  cout << "My favorite day is "
   cout << "Sunday\n;
   count++;
}
```

B)
```
int count = 10;
while (count < 5)
{  cout << "My favorite day is "
   cout << "Sunday\n;
   count++;
}
```

C)
```
int count = 1;
while (count < 5);
{  cout << "My favorite day is "
   cout << "Sunday\n;
   count++;
}
```

D)
```
int count = 1;
while (count < 5)
   cout << "My favorite day is "
   cout << "Sunday\n;
   count++;
```

5.3 In Checkpoint 5.2A, what is the loop control variable?

5.4 Write input validation loops that ask the user to do each of the following:

A) Enter a menu choice between 1 and 4.

B) Enter 'Y', 'y', 'N', or 'n'.

5.4 Counters

CONCEPT: A counter is a variable that is regularly incremented or decremented each time a loop iterates.

Sometimes it's important for a program to keep track of the number of iterations a loop performs. For example, Program 5-5 displays a table consisting of the numbers 1 through 5 and their squares, so its loop must iterate 5 times.

Program 5-5

```
1 // This program uses a while loop to display the numbers 1-5
2 // and their squares.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
```

(program continues)

Program 5-5 *(continued)*

```
7  int main()
8  {  int num = 1;
9
10    cout << "Number  Square\n";
11    cout << "--------------\n";
12    while (num <= 5)
13    {
14       cout << setw(4) << num  << setw(7) << (num * num) << endl;
15       num++;       // Increment counter
16    }
17    return 0;
18 }
```

Program Output

```
Number  Square
--------------
   1       1
   2       4
   3       9
   4      16
   5      25
```

In Program 5-5 the loop control variable `num` starts at `1` and is incremented each time through the loop. When `num` reaches `6`, the condition `num <= 5` becomes false, and the loop is exited. Variable `num` also acts as a *counter*, keeping count of how many times the loop has iterated so far.



NOTE: It's important that `num` be properly initialized. Remember, variables defined inside a function have no guaranteed starting value.

Letting the User Control the Loop

Sometimes we want to let the user decide how many times a loop should iterate. Program 5-6, which is a revision of Program 5-5, does this. It prompts the user to enter the maximum integer value to be displayed and squared. Then it has `num`, the loop counter, count up to that value.

Program 5-6

```
1  // This program displays integer numbers and their squares, beginning
2  // with one and ending with whatever number the user requests.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
```

(program continues)

Program 5-6 *(continued)*

```
 8 {
 9    int num,          // Counter telling what number to square
10        lastNum;      // The final integer value to be squared
11
12    // Get and validate the last number in the table
13    cout << "This program will display a table of integer\n"
14         << "numbers and their squares, starting with 1.\n"
15         << "What should the last number be?\n"
16         << "Enter an integer between 2 and 10: ";
17    cin  >> lastNum;
18
19    while ((lastNum < 2) || (lastNum > 10))
20    {  cout << "Please enter an integer between 2 and 10: ";
21       cin  >> lastNum;
22    }
23    // Display the table
24    cout << "\nNumber  Square\n";
25    cout << "--------------\n";
26
27    num = 1;          // Set the counter to the starting value
28    while (num <= lastNum)
29    {
30       cout << setw(4) << num  << setw(7) << (num * num) << endl;
31       num++;         // Increment the counter
32    }
33    return 0;
34 }
```

Program Output with Example Input Shown in Bold

```
This program will display a table of integer
numbers and their squares, starting with 1.
What should the last number be?
Enter an integer between 2 and 10: 3[Enter]

Number  Square
--------------
   1       1
   2       4
   3       9
```

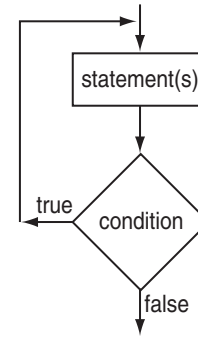
5.5 The `do-while` Loop

CONCEPT: The `do-while` loop is a post test loop, which means its expression is tested after each iteration.

In addition to the `while` loop, C++ also offers the `do-while` loop. The `do-while` loop looks similar to a `while` loop turned upside down. Figure 5-4 shows its format and a flowchart visually depicting how it works.

Figure 5-4

```
do
{    statement;
     statement;
     // Place as many statements
     // here as necessary.
} while (condition);
```



As with the `while` loop, if there is only one conditionally executed statement in the loop body, the braces may be omitted.



NOTE: The `do-while` loop must be terminated with a semicolon after the closing parenthesis of the test expression.

Besides the way it looks, the difference between the `do-while` loop and the `while` loop is that `do-while` is a *post test* loop. It tests its expression after each iteration is complete. This means `do-while` always performs at least one iteration, even if the test expression is false at the start. For example, in the following `while` loop the `cout` statement will not execute at all:

```
int x = 1;
while (x < 0)
    cout << x << endl;
```

But the `cout` statement in the following `do-while` loop will execute once because the `do-while` loop does not evaluate the expression `x < 0` until the end of the iteration.

```
int x = 1;
do
    cout << x << endl;
while (x < 0);
```

You should use the `do-while` loop when you want to make sure the loop executes at least once. For example, Program 5-7 computes and displays the average of a set of test scores before asking if the user wants to repeat the process with another set of scores. As with the `while` loop, a `do-while` loop can be written to iterate a set number of times or to allow the user to control how many times to loop. Program 5-7 illustrates another method for letting the user control the loop. It will repeat as long as the user enters a Y or y for yes.

Program 5-7

```
1 // This program averages 3 test scores. It uses a do-while loop
2 // that allows the code to repeat as many times as the user wishes.
3 #include <iostream>
4 using namespace std;
```

(program continues)

Program 5-7 *(continued)*

```
5
6  int main()
7  {
8     int score1, score2, score3;   // Three test scores
9     double average;               // Average test score
10    char again;                   // Loop again? Y or N
11
12    do
13    {  // Get three test scores
14       cout << "\nEnter 3 scores and I will average them: ";
15       cin  >> score1 >> score2 >> score3;
16
17       // Calculate and display the average
18       average = (score1 + score2 + score3) / 3.0;
19       cout << "The average is " << average << ".\n";
20
21       // Does the user want to average another set?
22       cout << "Do you want to average another set? (Y/N) ";
23       cin  >> again;
24    } while ((again == 'Y') || (again == 'y'));
25    return 0;
26 }
```

Program Output with Example Input Shown in Bold

```
Enter 3 scores and I will average them: 80 90 70[Enter]
The average is 80.
Do you want to average another set? (Y/N) y[Enter]

Enter 3 scores and I will average them: 60 75 88[Enter]
The average is 74.3333.
Do you want to average another set? (Y/N) n[Enter]
```

Using `do-while` with Menus

The `do-while` loop is a good choice for repeating a menu. Recall Program 4-27, which displays a menu of health club packages. Program 5-8 is a modification of that program that uses a `do-while` loop to repeat the program until the user selects item 4 from the menu.

Program 5-8

```
1  // This menu-driven Health Club membership program carries out the
2  // appropriate actions based on the menu choice entered. A do-while loop
3  // allows the program to repeat until the user selects menu choice 4.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
```

(program continues)

Program 5-8 *(continued)*

```
 7
 8 int main()
 9 {
10    // Constants for membership rates
11    const double ADULT_RATE  = 40.0;
12    const double CHILD_RATE  = 20.0;
13    const double SENIOR_RATE = 30.0;
14
15    int choice;       // Menu choice
16    int months;       // Number of months
17    double charges;   // Monthly charges
18
19    do
20    {  // Display the menu and get the user's choice
21       cout << "\n   Health Club Membership Menu\n\n";
22       cout << "1. Standard Adult Membership\n";
23       cout << "2. Child Membership\n";
24       cout << "3. Senior Citizen Membership\n";
25       cout << "4. Quit the Program\n\n";
26       cout << "Enter your choice: ";
27       cin  >> choice;
28
29       // Validate the menu selection
30       while ((choice < 1) || (choice > 4))
31       {
32          cout << "Please enter 1, 2, 3, or 4: ";
33          cin  >> choice;
34       }
35       // Process the user's choice
36       if (choice != 4)
37       {  cout << "For how many months? ";
38          cin  >> months;
39
40          // Compute charges based on user input
41          switch (choice)
42          {
43             case 1: charges = months * ADULT_RATE;
44                     break;
45             case 2: charges = months * CHILD_RATE;
46                     break;
47             case 3: charges = months * SENIOR_RATE;
48          }
49          // Display the monthly charges
50          cout << fixed << showpoint << setprecision(2);
51          cout << "The total charges are $" << charges << endl;
52       }
53    } while (choice != 4); // Loop again if the user did not
54                           // select choice 4 to quit
55    return 0;
56 }
```

(program continues)

Program 5-8 *(continued)*

Program Output with Example Input Shown in Bold

```
   Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: 1[Enter]
For how many months? 12[Enter]
The total charges are $480.00

   Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: 4[Enter]
```



Checkpoint

5.5 What will each of the following program segments display?

A)
```
int count = 10;
do
   cout << "Hello World\n";
while (count++ < 1);
```

B)
```
int v = 0;
do
    cout << v++;
while (v < 5);
```

C)
```
int count = 0, funny = 1, serious = 0, limit = 4;
do
{   funny++;
    serious += 2;
} while (count++ < limit);
cout << funny << " " << serious << " ";
cout << count << endl;
```

5.6 Write a program segment with a `do-while` loop that displays whether a user-entered integer is even or odd. The code should then ask the user if he or she wants to test another number. The loop should repeat so long as the user enters Y or y.

5.6 The for Loop

CONCEPT: **The `for` loop is a pretest loop that combines the initialization, testing, and updating of a loop control variable in a single loop header.**



VideoNote
The for Loop

In general, there are two categories of loops: conditional loops and count-controlled loops. A *conditional loop* executes as long as a particular condition exists. For example, an input validation loop executes as long as the input value is invalid. When you write a conditional loop, you have no way of knowing the number of times it will iterate.

Sometimes you know the exact number of iterations that a loop must perform. A loop that repeats a specific number of times is known as a *count-controlled loop*. For example, if a loop asks the user to enter the sales amounts for each month in the year, it will iterate twelve times. In essence, the loop counts to twelve and asks the user to enter a sales amount each time it makes a count. A count-controlled loop must possess three elements:

1. It must initialize a counter variable to a starting value.
2. It must test the counter variable by comparing it to a final value. When the counter variable reaches its final value, the loop terminates.
3. It must update the counter variable during each iteration. This is usually done by incrementing the variable.

Count-controlled loops are so common that C++ provides a type of loop specifically for them. It is known as the `for` loop. The `for` loop is specifically designed to initialize, test, and update a counter variable. Here is the format of the `for` loop.

```
for (initialization; test; update)
{
   statement;
   statement;
   // Place as many statements
   // here as necessary.
}
```

As with the other loops you have used, if there is only one statement in the loop body, the braces may be omitted.

The first line of the `for` loop is the *loop header*. After the key word `for`, there are three expressions inside the parentheses, separated by semicolons. (Notice there is no semicolon after the third expression.) The first expression is the *initialization expression*. It is typically used to initialize a counter to its starting value. This is the first action performed by the loop and it is only done once.

The second expression is the *test expression*. It tests a condition in the same way the test expression in the `while` and `do-while` loop does, and controls the execution of the loop. As long as this condition is true, the body of the `for` loop will repeat. The `for` loop is a pretest loop, so it evaluates the test expression before each iteration.

The third expression is the *update expression*. It executes at the end of each iteration. Typically, this is a statement that increments the loop's counter variable.

Here is an example of a simple `for` loop that prints "Hello" five times:

```
for (count = 1; count <= 5; count++)
   cout << "Hello" << endl;
```

In this loop, the initialization expression is `count = 1`, the test expression is `count <= 5`, and the update expression is `count++`. The body of the loop has one statement, which is the `cout` statement. Figure 5-5 illustrates the sequence of events that take place during the loop's execution. Notice that Steps 2 through 4 are repeated as long as the test expression is `true`.

Figure 5-5

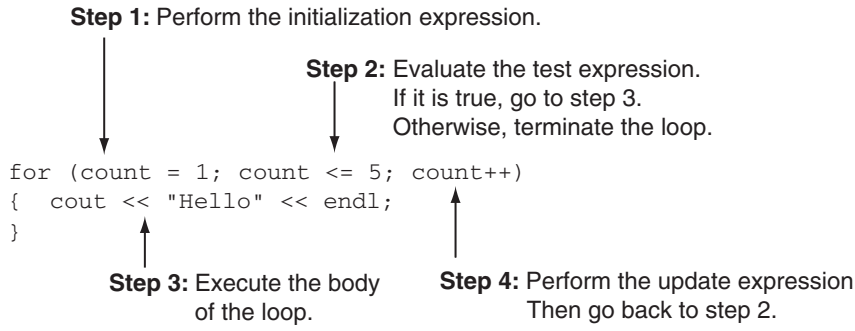
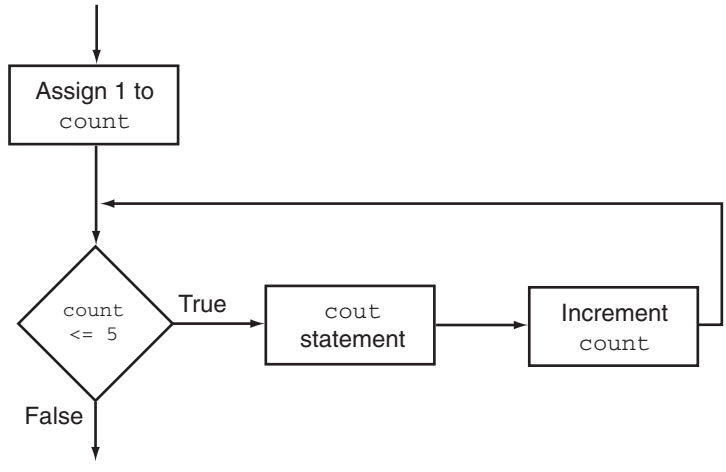


Figure 5-6 shows the loop's logic in the form of a flowchart.

Figure 5-6



Notice how the counter variable `count` is used to control the number of times the loop iterates. During the execution of the loop, this variable takes on the values 1 through 5, and when the test expression `count <= 5` becomes `false`, the loop terminates. Also notice that in this example the `count` variable is used only in the loop header, to control the number of loop iterations. It is not used for any other purpose. However, it is also possible to use the counter variable within the body of a loop. For example, look at the following code:

```
for (number = 1; number <= 5; number++)
   cout << number << " ";
```

The counter variable in this loop is `number`. In addition to controlling the number of iterations, it is also used in the body of the loop. This loop will produce the following output:

```
1 2 3 4 5
```

As you can see, the loop displays the contents of the `number` variable during each iteration.

Program 5-9 is a new version of Program 5-5 that displays the numbers 1–5 and their squares by using a `for` loop instead of a `while` loop.

Program 5-9

```
1  // This program uses a for loop to display the numbers 1-5
2  // and their squares.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {  int num;
9
10     cout << "Number  Square\n";
11     cout << "--------------\n";
12
13     for (num = 1; num <= 5; num++)
14        cout << setw(4) << num  << setw(7) << (num * num) << endl;
15     return 0;
16 }
```

Program Output

```
Number      Squared
------------------
  1           1
  2           4
  3           9
  4          16
  5          25
```

The `for` Loop is a Pretest Loop

Because the `for` loop tests its test expression before it performs an iteration, it is possible to write a `for` loop in such a way that it will never iterate. Here is an example:

```
for (count = 11; count <= 10; count++)
   cout << "Hello" << endl;
```

Because the variable `count` is initialized to a value that makes the test expression false from the beginning, this loop terminates as soon as it begins.

Avoid Modifying the Counter Variable in the Body of the `for` Loop

Be careful not to place a statement that modifies the counter variable in the body of the `for` loop. All modifications of the counter variable should take place in the update expression, which is automatically executed at the end of each iteration. If a statement in the body of the loop also modifies the counter variable, the loop will probably not terminate when you expect it to. The following loop, for example, increments `x` twice for each iteration:

```
for (x = 1; x <= 10; x++)
{
   cout << x << endl;
   x++;       // Wrong!
}
```

Other Forms of the Update Expression

You are not limited to incrementing the loop control variable by just 1 in the update expression. Here is a loop that displays all the even numbers from 2 through 100 by adding 2 to its counter:

```
for (num = 2; num <= 100; num += 2)
   cout << num << endl;
```

And here is a loop that counts backward from 10 down to 0:

```
for (num = 10; num >= 0; num--)
   cout << num << endl;
```

Defining a Variable in the `for` Loop's Initialization Expression

Not only may the counter variable be initialized in the initialization expression, it may be defined there as well. The following code shows an example. This is a modified version of the loop in Program 5-9.

```
for (int num = 1; num <= 5; num++)
   cout << setw(4) << num  << setw(7) << (num * num) << endl;
```

In this loop, the `num` variable is both defined and initialized in the initialization expression. If the counter variable is used only in the loop, it is considered good programming practice to define it in the loop header. This makes the variable's purpose clearer.

When a variable is defined in the initialization expression of a `for` loop, the scope of the variable is limited to the loop. This means you cannot access the variable in statements outside the loop. For example, the following program segment will not compile because the last `cout` statement cannot access the variable `count`.

```
for (int count = 1; count <= 10; count++)
   cout << count << endl;
cout << "count is now " << count << endl;    // ERROR!
```

Creating a User-Controlled for Loop

In Program 5-6 we allowed the user to control how many times a while loop should iterate. This can also be done with a for loop by having the user enter the final value for the counter variable. The following program segment illustrates this.

```
// Get the final counter value
cout << "How many times should the loop execute? ";
cin  >> finalValue;

for (int num = 1; num <= finalValue; num++)
{
   // Statements in the loop body go here.
}
```

Using Multiple Statements in the Initialization and Update Expressions

It is possible to execute more than one statement in the initialization expression and the update expression. When using multiple statements in either of these expressions, simply separate the statements with commas. For example, look at the loop in the following code, which has two statements in the initialization expression.

```
for (int x = 1, y = 1; x <= 5; x++)
{
   cout << x << " plus " << y << " equals " << (x + y) << endl;
}
```

The loop's initialization expression is

```
int x = 1, y = 1
```

This defines and initializes two int variables, x and y. The output produced by this loop is:

```
1 plus 1 equals 2
2 plus 1 equals 3
3 plus 1 equals 4
4 plus 1 equals 5
5 plus 1 equals 6
```

We can further modify the loop to execute two statements in the update expression. Here is an example:

```
for (int x = 1, y = 1; x <= 5; x++, y++)
{
   cout << x << " plus " << y << " equals " << (x + y) << endl;
}
```

The loop's update expression increments both the x and y variables.

```
x++, y++
```

The output produced by this loop is:

```
1 plus 1 equals 2
2 plus 2 equals 4
3 plus 3 equals 6
4 plus 4 equals 8
5 plus 5 equals 10
```

Connecting multiple statements with commas works well in the initialization and update expressions, but do *not* try to connect multiple expressions this way in the test expression. If you wish to combine multiple expressions in the test expression, you must use the `&&` or `||` operators.

Omitting the `for` Loop's Expressions or Loop Body

Although it is generally considered bad programming style to do so, one or more of the `for` loop's expressions, or even its loop body, can be omitted.

The initialization expression may be omitted from inside the `for` loop's parentheses if it has already been performed or if no initialization is needed. Here is an example a loop with the initialization being performed prior to the loop:

```
int num = 1;
for ( ; num <= maxValue; num++)
   cout << num << "       " << (num * num) << endl;
```

The update expression may be omitted if it is being performed elsewhere in the loop or if none is needed. Although this type of code is not recommended, the following `for` loop works just like a `while` loop:

```
int num = 1;
for ( ; num <= maxValue; )
{  cout << num << "       " << (num * num) << endl;
   num++;
}
```

It is also possible, though not recommended, to write a `for` loop that has no formal body. In this case, all the work of the loop is done by statements in the loop header. Here is an example that displays the numbers from 1 to 10. The combined increment operation and `cout` statement in the update expression perform the work of each iteration.

```
for (number = 1; number <= 10; cout << number++);
```



Checkpoint

5.7 Name the three expressions that appear inside the parentheses in the `for` loop's header.

5.8 You want to write a `for` loop that displays "I love to program" 50 times. Assume that you will use a counter variable named `count`.

A) What initialization expression will you use?
B) What test expression will you use?
C) What update expression will you use?
D) Write the loop.

5.9 What will each of the following program segments display?

A)
```
for (int count = 0; count < 6; count++)
      cout << (count + count) << " ";
```

B)
```
for (int value = -5; value < 5; value++)
      cout << value << " ";
```

C)
```
int x
for (x = 3; x <= 10; x += 3)
   cout << x << "   ";
   cout << x << "   ";
```

5.10 Write a `for` loop that displays your name 10 times.

5.11 Write a `for` loop that displays all of the odd numbers, 1 through 49.

5.12 Write a `for` loop that displays every fifth number, 0 through 100.

5.7 Keeping a Running Total

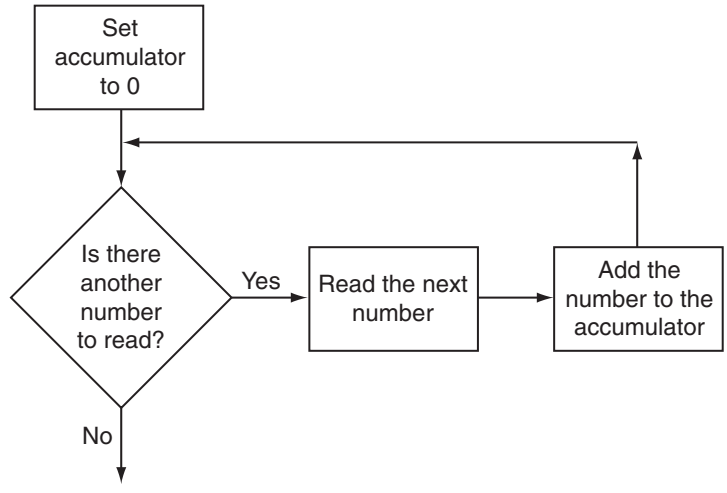
CONCEPT: **A *running total* is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an *accumulator*.**

Many programming tasks require you to add up a series of numbers. For example, if you want to find the average of a set of number, you must first add them up. Programs that add a series of numbers typically use two elements:

- A loop that reads each number in the series.
- A variable that accumulates the total of the numbers as they are read.

The variable that is used to accumulate the total of the numbers is called an *accumulator*. It is often said that the loop keeps a *running total* because it accumulates the total as it reads each number in the series. Figure 5-7 shows the general logic of a loop that calculates a running total.

Figure 5-7



When the loop finishes, the accumulator will contain the total of the numbers read by the loop. Notice that the first step in the flowchart is to set the accumulator variable to 0. This is a critical step. Each time the loop reads a number, it adds it to the accumulator. If the accumulator starts with any value other than 0, it will not contain the correct total when the loop finishes.

Let's look at a program that keeps a running total. Program 5-10 calculates a company's total sales over a period of time by reading daily sales figures and adding them to an accumulator. It then uses this total to find the average sales per day.

Program 5-10

```
1  // This program takes daily sales figures over a period of time
2  // and calculates their total. It then uses this total to compute
3  // the average daily sales.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  int main()
9  {
10     int numDays;               // Number of days
11     double dailySales,         // The sales amount for a single day
12            totalSales = 0.0,   // Accumulator, initialized with 0
13            averageSales;       // The average daily sales amount
14
15     // Get the number of days
16     cout << "For how many days do you have sales figures? ";
17     cin  >> numDays;
18
19     // Get the sales for each day and accumulate a total
20     for (int day = 1; day <= numDays; day++)  // day is the counter
21     {
22        cout << "Enter the sales for day " << day << ": ";
23        cin  >> dailySales;
24        totalSales += dailySales;    // Accumulate the running total
25     }
26     // Compute the average daily sales
27     averageSales = totalSales / numDays;
28
29     // Display the total sales and average daily sales
30     cout << fixed << showpoint << setprecision(2);
31     cout << "\nTotal sales:         $" << setw(8) << totalSales;
32     cout << "\nAverage daily sales: $" << setw(8) << averageSales
33          << endl;
34     return 0;
35 }
```

Program Output with Example Input Shown in Bold

```
For how many days do you have sales figures? 5[Enter]
Enter the sales for day 1: 425.16[Enter]
Enter the sales for day 2: 397.20[Enter]
Enter the sales for day 3: 404.11[Enter]
Enter the sales for day 4: 468.43[Enter]
Enter the sales for day 5: 502.19[Enter]

Total sales:         $ 2197.09
Average daily sales: $  439.42
```

Let's take a closer look at a few of the key lines in this program. In line 12 the `totalSales` variable is defined. This is the accumulator. Notice that it is initialized with 0. In line 17 the user enters how many days of sales figures there are. This number is stored in the `numDays` variable and determines how many times the loop beginning in line 20 iterates. The variable `day`, which is defined in the loop's initialization expression, is initialized with 1. This variable is the counter that controls the loop and keeps track of which day's sales amount is currently being read in and processed. The test expression specifies the loop will repeat as long as `day` is less than or equal to `numDays`. The update expression increments `day` by one at the end of each loop iteration.

During each loop iteration, in line 23, the user enters the amount of sales for one specific day. This amount is stored in the `dailySales` variable. Then, in line 24, this amount is added to the existing value stored in the `totalSales` variable. Note that line 24 does not *assign* `dailySales` to `totalSales`, but rather *increases* the value stored in `totalSales` by the amount in `dailySales`. After the loop has finished, `totalSales` will contain the total of all the daily sales figures entered. In line 27, this total is used to calculate the average daily sales amount. To do this the value stored in `totalSales` is divided by the value stored in `numDays` and the result is placed in the variable `averageSales`. The program now has all the information needed to display the `totalSales` and `averageSales` in lines 31–33.

5.8 Sentinels

CONCEPT: A *sentinel* is a special value that marks the end of a list of values.

Program 5-10, in the previous section, requires the user to know in advance the number of days there are sales figures for. Sometimes the user has a list that is very long and doesn't know how many items there are. In other cases, the user might be entering several lists and it is impractical to require that every item in every list be counted.

A technique that can be used in these situations is to ask the user to enter a sentinel at the end of the list. A *sentinel* is a special value that cannot be mistaken as a member of the list and signals that there are no more values to be entered. When the user enters the sentinel, the loop terminates.

Program 5-11 provides an example of using an end sentinel. This program calculates the total points earned by a soccer team over a series of games. It allows the user to enter the series of game points, then enter –1 to signal the end of the list.

Program 5-11

```
1  // This program illustrates the use of an end sentinel. It calculates
2  // the total number of points a soccer team has earned over a series
3  // of games. The user enters the point values, then -1 when finished.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     int game = 1,          // Game counter
10        points,             // Holds number of points for a specific game
11        total = 0;          // Accumulates total points for all games
```

(program continues)

Program 5-11 *(continued)*

```
12
13    // Read in the points for game 1
14    cout << "Enter the number of points your team has earned\n";
15    cout << "so far this season. Then enter -1 when finished.\n\n";
16    cout << "Enter the points for game " << game << ": ";
17    cin  >> points;
18
19    // Loop as long as the end sentinel has not yet been entered
20    while (points != -1)
21    {  // Add point just read in to the accumulator
22       total += points;
23
24       // Enter the points for the next game
25       cout << "Enter the points for game " << ++game << ": ";
26       cin  >> points;
27    }
28    // Display the total points
29    cout << "\nThe total points are " << total << endl;
30    return 0;
31 }
```

Program Output with Example Input Shown in Bold

```
Enter the number of points your team has earned
so far this season. Then enter -1 when finished.

Enter the points for game 1: 7
Enter the points for game 2: 9
Enter the points for game 3: 4
Enter the points for game 4: 6
Enter the points for game 5: -1

The total points are 26
```

Program Output with Different Example Input Shown in Bold

```
Enter the number of points your team has earned
so far this season. Then enter -1 when finished.

Enter the points for game 1: -1

The total points are 0
```

The value `-1` was chosen for the sentinel in this program because it is not possible for a team to score negative points. Notice that this program performs a priming read in line 17 to get the first value. This is done so the `while` loop will not try to test the value of `points` until a first value has been read in. It also makes it possible for the loop to immediately terminate if the user enters `-1` for the first value, as shown in the second sample run. Also note that the sentinel value is not included in the running total.



Checkpoint

5.13 In the following program segment, which variable is the counter and which is the accumulator?

```
int x = 0, y = 0, z;
cout << "How many numbers do you wish to enter? ";
cin  >> z;
while (x < z)
{
   int a;
   cout << "Enter a number: ";
   cin  >> a;
   y += a;
   x++;
}
cout << "The sum of those numbers is " << y << endl;
```

5.14 Write a `for` loop that sums up the squares of the integers from 1 through 10.

5.15 Write a `for` loop that sums up the squares of the odd integers from 1 through 9.

5.16 Write a `for` loop that repeats seven times, asking the user to enter a number each time and summing the numbers entered.

5.17 Write a `for` loop that calculates the total of the following series of numbers:

$$\frac{1}{30} + \frac{2}{29} + \frac{3}{28} + \frac{4}{27} + \ldots \frac{30}{1}$$

5.18 Write a `for` loop that calculates the total of the following series of numbers:

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots \frac{1}{1024}$$

5.19 Write a sentinel controlled `while` loop that accumulates a series of test scores input by the user, until `-99` is entered. The code should then report how many scores were entered and the average of these scores. Do not count the end sentinel `-99` as a score.

5.9 Using a Loop to Read Data from a File

CONCEPT: It is possible to read data from an input file without knowing how many values there are. This can be done by placing a read statement in a loop and reading data until all the values have been read.

You already know how easy it is for a program to read data from a file. By placing the statement that reads the data in a loop, the same statement can be executed multiple times and can therefore be used to read many different pieces of data. For example, it would be easy to modify Program 5-10 to read the sales figures or Program 5-11 to read the game points from a data file, rather than having these input from the keyboard by the user. However, both of these programs require the user to know how much data there is. Program 5-10 requires the user to enter the number of data values to be read and Program 5-11 requires the user to input a sentinel value to signal there is no more data when all the values have been entered.

When reading data from a file, it is not necessary for the user to know how many data values there are or where the data ends. This is because files have an *end of file* mark at their end. You cannot see it, but it is there, and a program can test to see whether or not it has

been reached. One way to do this is with the `eof` member function, which can be called for any `ifstream` object. It returns `true` when the previous read encountered the end of file.

Otherwise it returns `false`. The format of the function call is

```
inputFile.eof()
```

where `inputFile` is the name of an `ifstream` object. The following code segment shows a call to this function being used to control a loop.

```
inputFile >> item;         // Read a value from the file
while (!inputFile.eof())   // While the end of file was not reached yet
{  cout << item;           // Display the value just read
   inputFile >> item;      // Read the next value
}
```

This loop will read and display items from the file until the end of file is reached. Notice that the first read from the data file is done before the `while` loop, and thus before the `eof` function is called the first time. This is necessary because the `eof` function does not return `true`, signaling you are at the end of the file, even if a file is empty. It only returns `true` when the end of file is detected while attempting to do a read. If the first read attempt fails because the file is empty, the loop body will never be executed and no data values will be processed. If a data value is successfully read, the loop will be entered, the value will be processed (in this case displayed), and then the next read operation will be performed. Program 5-12 demonstrates how to use a loop controlled by the return value of the `eof` function to read all the numbers from a file, regardless of how many there are.

Program 5-12

```
1  // This program uses a loop to read and display all the numbers in a
2  // file. The ifstream eof member function is used to control the loop.
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  int main()
8  {
9     int number;
10    ifstream inputFile;
11
12    inputFile.open("numbers.dat");  // Open the file
13    if  (!inputFile)                // Test for errors
14        cout << "Error opening file.\n";
15    else
16    {  inputFile >> number;         // Read the first number
17       while (!inputFile.eof())     // While read was good; no eof yet
18       {
19          cout << number << "   ";  // Display the number
20          inputFile >> number;      // Read the next number
21       }
22       cout << endl;
23       inputFile.close();           // Close the file
24    }
25    return 0;
26 }
```

(program continues)

Program 5-12 *(continued)*

Program Output

```
2   4   6   8   10   12   14
```

Although the `eof` function is commonly used to test if the end of a file has been reached, there is a potential problem with using it. For it to work correctly, using the method we have shown, there must be whitespace (a space, tab, or [Enter]) in the file after the last data value. Otherwise the `eof` function will return a `true` when the last value is read, rather than when no more values exist to be read, and this last value will never be processed. For example, the data file used for input in Program 5-12 contained the 7 numbers shown in the output, followed by a space. If the final space is removed from the data file and the program is rerun, the following output will be produced.

Program Output with Final Whitespace Removed from the File

```
2   4   6   8   10   12
```

Luckily a safer method exists in C++ to determine when there is no more data in a file. The stream extraction operator (>>) not only can read data from a file, but it also returns a value indicating whether the data was successfully read or not. If the operator returns `true`, then a value was successfully read. If the operator returns `false`, it means that no value was read from the file. For example, look at the following code.

```
if (inputFile >> number)
{
   // Data was successfully read from the file
   cout << "The data read from the file is " << number << endl;
}
else
{
   // No data was read from the file
   cout << "Could not read an item from the file.\n";
}
```

Notice that the read operation and the test to see if it was successful are both done in the same statement. Combining a read and its test for success like this can also be used to control a loop. By using the return value of the >> operator to control a `while` loop that reads from a file, data will be read in until a read fails. This only occurs when there is no more data to read (or some catastophic event, such as a disk crash). All the data in the file will be read, regardless of whether the final data value is followed by whitespace or not. Program 5-13 modifies Program 5-12 to use this method to control the loop.

Program 5-13

```
1 // This program uses a loop to read and display all of the numbers in
2 // a file. The >> operator return value is used to control the loop.
3 #include <iostream>
4 #include <fstream>
5 using namespace std;
6
```

(program continues)

Program 5-13 *(continued)*

```
7  int main()
8  {
9     int number;
10    ifstream inputFile;
11
12    inputFile.open("numbers.dat");    // Open the file
13    if (!inputFile)                   // Test for errors
14        cout << "Error opening file.\n";
15    else
16    {  while(inputFile >> number)     // Read a number and execute the
17       {                              // loop while read was successful
18          cout << number << "    ";   // Display the number
19       }
20       cout << endl;
21       inputFile.close();             // Close the file
22    }
23    return 0;
24 }
```

Program Output

```
2    4    6    8    10    12    14
```

5.10 Focus on Software Engineering: *Deciding Which Loop to Use*

CONCEPT: **Although most repetitive algorithms can be written with any of the three types of loops, each works best in different situations.**

Each of C++'s three loops are ideal to use in different situations. Here's a short summary of when each loop should be used.

The `while` Loop

The `while` loop is a pretest loop. It is ideal in situations where you do not want the loop to iterate if the test condition is false from the beginning. For example, validating input that has been read and reading lists of data terminated by a sentinel value are good applications of the `while` loop.

```
cout << "This program finds the square of any integer.\n";
cout << "\nEnter an integer, or -99 to quit: ";
cin  >> num;

while (num != -99)
{  cout << num << " squared is " << pow(num, 2.0) << endl;
   cout << "\nEnter an integer, or -99 to quit ";
   cin  >> num;
}
```

The `do-while` Loop

The `do-while` loop is a post test loop. It is ideal in situations where you always want the loop to iterate at least once. The `do-while` loop is a good choice for repeating a menu or for asking the user if they want to repeat a set of actions.

```
cout << "This program finds the square of any integer.\n";
do
{  cout << "\nEnter an integer: ";
   cin  >> num;
   cout << num << " squared is " << pow(num, 2.0) << endl;
   cout << "Do you want to square another number? (Y/N) ";
   cin  >> doAgain;
} while ((doAgain == "Y") || (doAgain == "y"));
```

The `for` Loop

The `for` loop is a pretest loop with built-in expressions for initializing, testing, and updating a counter variable. The `for` loop is ideal in situations where the exact number of iterations is known.

```
cout << "This program finds the squares of the integers "
     << "from 1 to 8.\n\n";
for (num = 1; num <= 8; num++)
{
   cout << num << " squared is " << pow(num, 2.0) << endl;
}
```

5.11 Nested Loops

CONCEPT: A loop that is inside another loop is called a *nested loop*.



VideoNote
Nested Loops

In Chapter 4 you saw how one `if` statement could be nested inside another one. It is also possible to nest one loop inside another loop. The first loop is called the *outer loop*. The one nested inside it is called the *inner loop*. This is illustrated by the following two `while` loops. Notice how the inner loop must be completely contained within the outer one.

```
while (condition1)      // Beginning of the outer loop
{  ---
   while (condition2)   // Beginning of the inner loop
   {  ---
      ---
   }                    // End of the inner loop
}                       // End of the outer loop
```

Nested loops are used when, for each iteration of the outer loop, something must be repeated a number of times. Here are some examples from everyday life:

- For *each* batch of cookies to be baked we must put *each* cookie on the cookie sheet.
- For *each* salesperson, we must add up *each* sale to determine total commission.
- For *each* teacher we must produce a class list for *each* of their classes.
- For *each* student we must add up *each* test score to find the student's test average.

Whatever the task, the inner loop will go through all its iterations each time the outer loop is executed. This is illustrated by Program 5-14, which handles this last task, finding student test score averages. Any kind of loop can be nested within any other kind of loop. This program uses two `for` loops.

Program 5-14

```
1  // This program averages test scores. It asks the user for the
2  // number of students and the number of test scores per student.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int numStudents,    // Number of students
9         numTests;       // Number of tests per student
10    double average;     // Average test score for a student
11
12    // Get the number of students
13    cout << "This program averages test scores.\n";
14    cout << "How many students are there? ";
15    cin  >> numStudents;
16
17    // Get the number of test scores per student
18    cout << "How many test scores does each student have? ";
19    cin  >> numTests;
20    cout << endl;
21
22    // Read each student's scores and compute their average
23    for (int snum = 1; snum <= numStudents; snum++)    // Outer loop
24    {  double total = 0.0;   // Initialize accumulator
25
26       for (int test = 1; test <= numTests; test++)    // Inner loop
27       {  int score;
28
29          // Read a score and add it to the accumulator
30          cout << "Enter score " << test << " for ";
31          cout << "student " << snum << ": ";
32          cin  >> score;
33          total += score;  //
34       }                                               // End inner loop
35       // Compute and display the student's average
36       average = total / numTests;
37       cout << "The average score for student " << snum;
38       cout << " is " << average << "\n\n";
39    }                                                  // End outer loop
40    return 0;
41 }
```

(program continues)

Program 5-14 *(continued)*

Program Output with Example Input Shown in Bold

```
This program averages test scores.
How many students are there? 2[Enter]
How many test scores does each student have? 3[Enter]

Enter score 1 for student 1: 84[Enter]
Enter score 2 for student 1: 79[Enter]
Enter score 3 for student 1: 97[Enter]
The average for student 1 is 86.6667
Enter score 1 for student 2: 92[Enter]
Enter score 2 for student 2: 88[Enter]
Enter score 3 for student 2: 94[Enter]
The average for student 2 is 91.3333
```

Let's trace what happened in Program 5-14, using the sample data shown. In this case, for each of two students, each of three scores were input and summed. First in line 23 the outer loop was entered and `snum` was set to 1. Then, once the `total` accumulator was initialized to zero for that student, the inner loop, which begins on line 26, was entered. While the outer loop was still on its first iteration and `snum` was still 1, the inner loop went through all of its iterations, handling tests 1, 2, and 3 for that student. It then exited the inner loop and in lines 36 through 38 calculated and output the average for student 1. Only then did the program reach the bottom of the outer loop and go back up to do its second iteration. The second iteration of the outer loop processed student 2. For *each* iteration of the outer loop, the inner loop did *all* its iterations.

It might help to think of each loop as a rotating wheel. The outer loop is a big wheel that is moving slowly. The inner loop is a smaller wheel that is spinning quickly. For every rotation the big wheel makes, the little wheel makes many rotations. Since, in our example, the outer loop was done twice, and the inner loop was done three times for each iteration of the outer loop, the inner loop was done a total of six times in all. This corresponds to the six scores input by the user. The following points summarize this.

- An inner loop goes through all of its iterations for each iteration of an outer loop.
- Inner loops complete their iterations faster than outer loops.
- To get the total number of iterations of an inner loop, multiply the number of iterations of the outer loop by the number of iterations done by the inner loop each time the outer loop is done.

5.12 Breaking Out of a Loop

CONCEPT: The `break` statement causes a loop to terminate early.

Sometimes it's necessary to stop a loop before it goes through all its iterations. The `break` statement, which was used with `switch` in Chapter 4, can also be placed inside a loop. When it is encountered, the loop stops and the program jumps to the statement immediately following the loop.

The `while` loop in the following program segment appears to execute 10 times, but the `break` statement causes it to stop after the fifth iteration.

```
int count = 1;
while (count <= 10)
{
   cout << count << endl;
   count++;
   if (count == 6)
      break;
}
```

This example is just to illustrate what a `break` statement inside a loop will do. You would not normally want to use one in this way because it violates the rules of structured programming and makes code more difficult to understand, debug, and maintain. The exit from a loop should be controlled by its condition test at the top of the loop, as in a `while` loop or `for` loop, or at the bottom, as in a `do-while` loop. Normally the only time a `break` statement is used inside a loop is to exit the loop early if an error condition occurs. Program 5-15 illustrates an example of this.

Program 5-15

```
1  // This program is supposed to find the square root of 5 numbers
2  // entered by the user. However, if a negative number is entered
3  // an error message displays and a break statement is used to
4  // stop the loop early.
5  #include <iostream>
6  #include <cmath>
7  using namespace std;
8
9  int main()
10 {
11    double number;
12
13    cout << "Enter 5 positive numbers separated by spaces and \n"
14         << "I will find their square roots: ";
15
16    for (int count = 1; count <= 5; count++)
17    {
18       cin >> number;
19       if (number >= 0.0)
20       {  cout << "\nThe square root of " << number << " is "
21               << sqrt(number);
22       }
23       else
24       {  cout << "\n\n" << number << " is negative. "
25               << "I cannot find the square root \n"
26               << "of a negative number. The program is terminating.\n";
27          break;
28       }
29    }
30    return 0;
31 }
```

(program continues)

Program 5-15 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter 5 positive numbers separated by spaces and
I will find their square roots: 12 15  17 19 31[Enter]

The square root of 12 is 3.4641
The square root of 15 is 3.87298

-17 is negative. I cannot find the square root
of a negative number. The program is terminating.
```

Using `break` in a Nested Loop

In a nested loop, the `break` statement only interrupts the loop it is placed in. The following program segment displays five rows of asterisks on the screen. The outer loop controls the number of rows and the inner loop controls the number of asterisks in each row. The inner loop is designed to display 20 asterisks, but the `break` statement stops it during the 11th iteration.

```
for (row = 0; row < 3; row++)
{
   for (star = 0; star < 20; star++)
   {
      cout << '*';
      if (star == 10)
          break;
   }
   cout << endl;
}
```

The output of this program segment is

```
***********
***********
***********
```



WARNING! Use the `break` statement with great caution. Because it bypasses the loop condition to terminate a loop, it violates the rules of structured programming and makes code more difficult to understand, debug, and maintain. For this reason, we do ***not*** recommend using it to exit a loop. Because it is part of the C++ language, however, we have discussed it briefly in this section.

5.13 The `continue` Statement

CONCEPT: **The `continue` statement causes a loop to stop its current iteration and begin the next one.**

The `continue` statement causes the current iteration of a loop to end immediately. When `continue` is encountered, all the statements in the body of the loop that appear after it are

ignored, and the loop prepares for the next iteration. In a `while` loop, this means the program jumps to the test expression at the top of the loop. As usual, if the expression is still true, the next iteration begins. In a `do-while` loop, the program jumps to the test expression at the bottom of the loop, which determines if the next iteration will begin. In a `for` loop, `continue` causes the update expression to be executed, and then the test expression to be evaluated.

The following program segment demonstrates the use of `continue` in a `while` loop:

```
int testVal = 0;
while (testVal < 10)
{
   testVal++;
   if (testVal) == 4
      continue;        // Terminate this iteration of the loop
   cout << testVal << " ";
}
```

This loop looks like it displays the integers 1–10. However, here is the output:

```
1 2 3 5 6 7 8 9 10
```

Notice that the number 4 does not print. This is because when `testVal` is equal to 4, the `continue` statement causes the loop to skip the `cout` statement and begin the next iteration.



WARNING! As with the `break` statement, the `continue` statement violates the rules of structured programming and makes code more difficult to understand, debug, and maintain. For this reason, you should use `continue` with great caution.

There are some practical uses of the `continue` statement, however, and Program 5-16 illustrates one of these. The program calculates the charges for DVD rentals where current releases cost $3.50 and all others cost $2.50. If a customer rents several DVDs, every third one is free. The `continue` statement is used to skip the part of the loop that calculates the charges for every third DVD.

Program 5-16

```
1  // This program calculates DVD rental charges where every third DVD
2  // is free. It illustrates the use of the continue statement.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     int numDVDs;          // Number of DVDs being rented
10    double total = 0.0;   // Accumulates total charges for all DVDs
11    char current;         // Current release? (Y/N)
12
13    // Get number of DVDs rented
14    cout << "How many DVDs are being rented? ";
15    cin  >> numDVDs;
16
```

(program continues)

Program 5-16 *(continued)*

```
17    // Determine the charges
18    for (int dvdCount = 1; dvdCount <= numDVDs; dvdCount++)
19    {   if (dvdCount % 3 == 0)   // If it's a 3rd DVD it's free
20        {
21           cout << "DVD #" << dvdCount << " is free!\n";
22           continue;
23        }
24        cout << "Is DVD #" << dvdCount << " a current release (Y/N)? ";
25        cin  >> current;
26        if ((current == 'Y') || (current == 'y'))
27            total += 3.50;
28        else
29            total += 2.50;
30    }
31    // Display the total charges
32    cout << fixed << showpoint << setprecision(2);
33    cout << "The total is $" << total << endl;
34    return 0;
35 }
```

Program Output with Example Input Shown in Bold

```
How many DVDs are being rented? 6[Enter]
Is DVD #1 a current release (Y/N)? y[Enter]
Is DVD #2 a current release (Y/N)? n[Enter]
DVD #3 is free!
Is DVD #4 a current release (Y/N)? n[Enter]
Is DVD #5 a current release (Y/N)? y[Enter]
DVD #6 is free!
The total is $12.00
```



Checkpoint

5.20 Which loop (`while`, `do-while`, or `for`) is best to use in the following situations?

A) The user must enter a set of exactly 14 numbers.
B) A menu must be displayed for the user to make a selection.
C) A calculation must be made an unknown number of times. (Maybe even no times.)
D) A series of numbers must be entered by the user, terminated by a sentinel value.
E) A series of values must be entered. The user specifies exactly how many.

5.21 How many stars be displayed in each of the following program segments?

A)
```
for (x = 0; x < 20; x++)
{   for (y = 0; y < 30; y++)
         cout << '*' << endl;
}
```

B)
```
for (x = 0; x < 20; x++)
{   for (y = 0; y < 30; y++)
    { if (y > 10)
         break;
      cout << '*' << endl;
    }
}
```

5.22 What will the following program segment display?

```
int x = 0, y = 0;

while (x++ < 5)
{
   if (x == 3)
      continue;
   y += x;
   cout << y << endl;
}
```

5.14 Focus on Testing and Debugging: *Creating Good Test Data*

CONCEPT: Thorough testing of a program requires good test data.

Once a program has been designed, written in a programming language, and found to compile and link without errors, it must be thoroughly tested to find any logic errors and to ensure that it works correctly according to the original problem specification. When it comes to creating test data, quality is more important than quantity. That is, a small set of good test cases can provide more information about how a program works than twice as many cases that are not carefully thought out. Each test case should be designed to test a different aspect of the program and you should always know what each test set you use is checking for. To illustrate this, look at Program 5-17. It uses a sentinel-controlled loop to average two test scores for each student in the class, where all test scores are between 0 and 100. The program compiles, links, and runs. But it contains several logic errors.

Program 5-17

```
1  // This program averages 2 test scores for each student in a class.
2  // It contains logic errors.
3  #include <iostream>
4  #include <string>
5  #include <iomanip>
6  using namespace std;
7
8  int main()
9  {
10    string name;                  // Student first name
11
12    int count = 1,                // Student counter
13        score,                    // An individual score read in
14        totalScore = 0;           // Total of a student's 2 scores
15    double average;               // Average of a student's 2 scores
16
17    cout << fixed << showpoint << setprecision(1);
18    cout << "Enter the first name of student " << count
19         << " (or Q to quit): ";
20    cin  >> name;
```

(program continues)

Program 5-17 *(continued)*

```
21
22    while ((name != "Q") && (name != "q"))
23    {
24       // Get and validate the first score
25       cout << "Enter score 1: ";
26       cin  >> score;
27       if ((score <= 0) || (score >= 100))
28       {  cout << "Score must be between 0 and 100. Please reenter: ";
29          cin  >> score;
30       }
31       totalScore += score;      // Add the first score onto the total
32
33       // Get and validate the second score
34       cout << "Enter score 2: ";
35       cin  >> score;
36       if ((score <= 0) || (score >= 100))
37       {  cout << "Score must be between 0 and 100. Please reenter: ";
38          cin  >> score;
39       }
40       totalScore += score;     // Add the second score onto the total
41
42       // Calculate and print average
43       average = totalScore / 2;
44       cout << name << setw(6) << average << endl;
45
46       // Get the next student name
47       cout << "Enter the first name of student " << count++
48            << " (or Q to quit): ";
49       cin  >> name;
50    }
51    return 0;
52 }
```

Table 5-1 Preliminary Test Plans for Program 5-17

	Name	Score 1	Score 2	Expected Outcome
Test 1:	Mary Q	80	80	80.0 program quits
Test 2:	Bill Q	70	80	75.0 program quits
Test 3:	Tom q	80	90	85.0 program quits
Test 4:	Sam q	–1 then 1	999 then 99	50.0 program quits

Try running the program using the four test cases shown in Table 5-1. The program contains five logic errors. However, if it is run with just these four test cases, none of the errors will be revealed. The test data is not designed carefully enough to catch them. Tests 1, 2, and 3 are really just three versions of the same test. They all simply check that the program can compute a correct average for a single student where the result has no decimal digits. The final test checks that the program can catch a single invalid value that is too small or too big, but does not check what will happen if a second invalid value is entered for the same input. Table 5-2 contains a better set of tests and illustrates some of the kinds of things you should check for when you test a program. These tests will reveal all five of the program's errors.

Table 5-2 Modified Test Plans for Program 5-17

Test	Name	Score 1	Score 2	Purpose	Expected Outcome
1	Mary Bill Tom Q	80 70 80	80 80 91	Program correctly handles both even results and ones with decimal values. Program can loop to handle multiple students. Program ends when Q is entered for the name.	80.0 75.0 85.5 program ends
2	Sam Ted q	–1 then 1 –1 then –2 then 1	101 then 99 200 then 500 then 99	Program correctly handles invalid scores, even when more than one bad score is entered in a row. Program catches bad inputs immediately outside the valid range (e.g., –1 & 101). Program ends when q is entered for the name.	50.0 50.0 program ends
3	Bob q	0	100	Program allows values at extreme ends of the valid range.	50.0 program ends

Rerun Program 5-17 using the test cases from Table 5-2 and examine the incorrect output to identify the errors. Then see if you can fix these errors. Do not rewrite the program. Just make the smallest changes necessary to correct the errors. Now test the program again using the test cases in Table 5-2. Continue making corrections and retesting until the program successfully passes all three of these test cases. A correct solution can be found on the student CD in the `pr5-17B.cpp` file of the Chapter 5 programs folder.

5.15 Central Mountain Credit Union Case Study

The Central Mountain Credit Union uses a central data processing system where each branch accesses a minicomputer at the Credit Union's main office. The minicomputer has a tendency to slow down when many users are on the system. Because of this, the loan officer at one of the branch offices has asked you to write a loan amortization program to run locally, on a desktop PC.

Problem Statement

When given the loan amount, annual interest rate, and number of years of a loan, the program must determine and display the monthly payment amount. It must then create and display an amortization table that lists the following information for each month of the loan:

- payment number
- amount of that month's payment that was applied to interest
- amount of that month's payment that was applied to principal
- balance after that payment.

The following report may be used as a model. It shows all the required information on a $2000 loan at 7.5% annual interest for .5 years (i.e., 6 months).

```
Monthly payment: $340.66

Month   Interest   Principal   Balance
-----------------------------------
   1      12.50      328.16    1671.84
   2      10.45      330.21    1341.62
   3       8.39      332.28    1009.34
   4       6.31      334.35     674.99
   5       4.22      336.44     338.55
   6       2.12      338.55       0.00
```

Calculations

The credit union uses the following formula to calculate the monthly payment of a loan:

$$\text{Payment} = \frac{\text{Loan} * \text{Rate}/12 * \text{Term}}{\text{Term} - 1}$$

where:

Loan = the amount of the loan
Rate = the annual interest rate
Term = $(1 + \text{Rate}/12)^{\text{Years}*12}$

Variables

Table 5-3 lists the variables needed in the program.

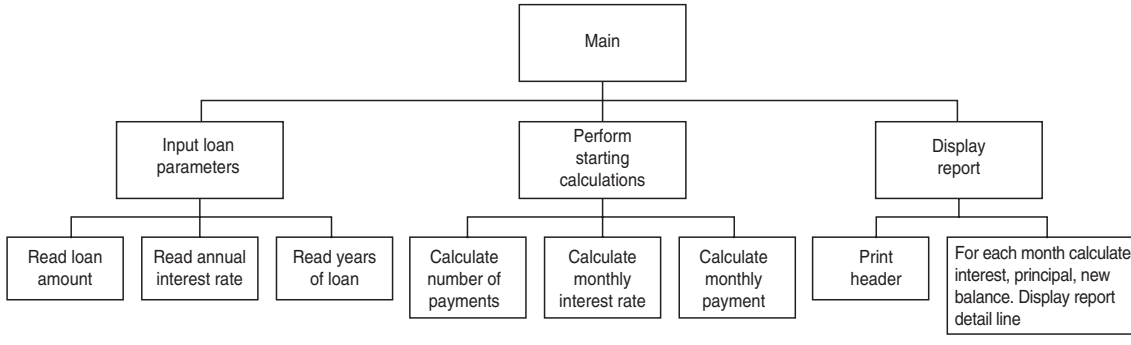
Table 5-3 Variables Used in the Central Mountain Credit Union Case Study

Variable	Description
`loan`	A `double`. Holds the loan amount.
`rate`	A `double`. Holds the annual interest rate.
`moInterestRate`	A `double`. Holds the monthly interest rate.
`years`	A `double`. Holds the number of years of the loan.
`balance`	A `double`. Holds the remaining balance to be paid.
`term`	A `double`. Used in the monthly payment calculation.
`payment`	A `double`. Holds the monthly payment amount.
`numPayments`	An `int`. Holds the total number of payments.
`month`	An `int`. Loop control variable that holds the current payment number.
`moInterest`	A `double`. Holds the monthly interest amount.
`principal`	A `double`. Holds the amount of the monthly payment that pays down the loan.

Program Design

Figure 5-8 shows a hierarchy chart for the program.

Figure 5-8



Detailed Pseudocode (including actual variable names and needed calculations)

```
Input loan, rate, years
numPayments = years * 12.0
moInterestRate = rate / 12.0
term = (1 + moInterestRate)numPayments
payment = (loan * moInterestRate * term) / (term – 1.0)
Display payment
Display a report header with column headings
balance = loan       // Remaining balance starts out as full loan amount
```

```
For each month of the loan
   moInterest = moInterestRate * balance  // Calculate interest first
   If it's not the final month
      principal = payment – moInterest    // Rest of pmt goes to principal
   Else                                   // It's the last month so
      principal = balance                 // pay off exact final balance
      payment = balance + moInterest
   End If
   balance = balance – principal          // Only principal reduces the
                                          // balance, not the whole pmt
   Display month, moInterest, principal, balance
End of loop
```

The Program

The next step, after the pseudocode has been checked for logic errors, is to expand the pseudocode into the final program. This is shown in Program 5-18.

Program 5-18

```
1  // This program produces a loan amortization table for the
2  // Central Mountain Credit Union.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath>          // Needed for the pow function
6  using namespace std;
7
8  int main()
9  {
10    double loan,            // Loan amount
11           rate,            // Annual interest rate
12           moInterestRate,  // Monthly interest rate
13           years,           // Years of loan
14           balance,         // Monthly balance
15           term,            // Used to calculate payment
16           payment;         // Monthly payment
17    int    numPayments;     // Number of payments
18
19    // Get loan information
20    cout << "Loan amount: $";
21    cin  >> loan;
22    cout << "Annual interest rate (entered as a decimal): ";
23    cin  >> rate;
24    cout << "Years of loan: ";
25    cin  >> years;
26
27    // Calculate monthly payment
28    numPayments = static_cast<int>(12 * years);
29    moInterestRate = rate / 12.0;
30    term = pow((1 + moInterestRate), numPayments);
31    payment = (loan * moInterestRate * term) / (term - 1.0);
32
33    // Display monthly payment
34    cout << fixed << showpoint << setprecision(2);
35    cout << "Monthly payment: $" << payment << endl;
36
```

(program continues)

Program 5-18 *(continued)*

```
37    // Display report header
38    cout << endl;
39    cout << setw(5)  << "Month"     << setw(10) << "Interest";
40    cout << setw(10) << "Principal" << setw(9)  << "Balance" << endl;
41    cout << "---------------------------------\n";
42
43    balance = loan;       // Remaining balance starts out as full loan amount
44
45    // Produce a listing for each month
46    for (int month = 1; month <= numPayments; month++)
47    {
48       double moInterest,         // Amount of pmt that pays interest
49              principal;          // Amount of pmt that lowers the balance
50
51       // Calculate amount paid for this month's interest and principal
52       moInterest = moInterestRate * balance; // Calculate interest first
53       if (month != numPayments)              // If not the final month
54           principal = payment - moInterest;  // rest of pmt goes
55                                              // to principal
56
57       else                                   // It's the last month so
58       {   principal = balance;               // pay exact final balance
59           payment = balance + moInterest;
60       }
61       // Calculate new loan balance          // Only principal reduces the
62       balance -= principal;                  // balance, not the whole pmt
63
64       // Display this month's payment figures
65       cout << setw(4)  << month     << setw(10) << moInterest;
66       cout << setw(10) << principal << setw(10) << balance << endl;
67    }
68    return 0;
69 }
```

Program Output with Example Input Shown in Bold

```
Loan amount: $1200[Enter]
Annual interest rate (entered as a decimal): .08[Enter]
Years of loan: 1[Enter]
Monthly payment: $104.39

Month   Interest Principal   Balance
---------------------------------
   1       8.00     96.39   1103.61
   2       7.36     97.03   1006.59
   3       6.71     97.68    908.91
   4       6.06     98.33    810.58
   5       5.40     98.98    711.60
   6       4.74     99.64    611.96
   7       4.08    100.31    511.65
   8       3.41    100.98    410.68
   9       2.74    101.65    309.03
  10       2.06    102.33    206.70
  11       1.38    103.01    103.69
  12       0.69    103.69      0.00
```



NOTE: You might have noticed in the output that for some months, such as months 5 and 6, the interest amount plus the principal amount does not add up to the monthly payment amount. Also, for some months, the previous balance minus the principal paid does not exactly equal the new balance. These problems are due to *round-off error*, which is caused by a disparity between the precision of a value the computer stores internally and the precision of the value it displays. Do not worry about this for now. You will learn later how to deal with this.

Testing the Program

Testing the program has been left as an exercise for you to do. Use what you learned in Section 5.14 about developing good test cases to develop a set of cases you can use to test Program 5-18. The program runs correctly except for one special case, where it fails. The program design failed to realize the need to handle this special case differently than it handles other data. Try to come up with input data for a test case that reveals the error. Then, once you have identified the problem, see if you can revise the program to fix it. A corrected version of Program 5-18 can be found in the `pr5-18B.cpp` file of the Chapter 5 programs folder on the student CD.

Lightening Lanes Case Study

The following additional case study, which contain applications of material introduced in Chapter 5, can be found on the student CD.

On Tuesday afternoons, Lightening Lanes Bowling Alley runs a special class to teach children to bowl. Each lane has an instructor who works with a team of four student bowlers and instructs them as they bowl three lines (i.e., games). The management of Lightening Lanes has asked you to develop a program that will report each student's three-game average score and compare it to the average score they bowled the previous week. In this way, the students can see how much they are improving. The program will use looping structures and data validation techniques learned in Chapter 5.

5.16 Tying It All Together: *What a Colorful World*

In Chapter 5's Tying It All Together section we'll take a look at how to use the looping constructs you learned about in this chapter, along with colorful output characters, to create interesting screen displays.

All the C++ programs you have seen so far produce output that is white on a black background. This is because they use the standard C++ `iostream` libraries, which can only display output in these two colors. However, C++ compilers provide other libraries you can use to call operating system functions that can display output in many colors. Because these libraries are tailored to specific operating systems, programs that use them will only run on the system they were written for.

Here is how to use Microsoft Windows functions to create programs with colorful output that can run on Windows 2000, XP, Vista, and 7.

The first thing you need to do is include the following file in your program so you will be able to use the functions you need:

```
#include <windows.h>
```

Next, because programs can actually access more than one screen device at a time, you will need to indicate which screen you want the colors you set to appear on. The `cout` object writes to the standard output screen. You can set colors on this screen by providing a `handle` to it. A handle is an object of type `HANDLE`, which is defined by Microsoft Windows. Here is how to obtain a handle to the standard output screen:

```
HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);
```

`GetStdHandle` is a Windows-specific library function and `STD_OUTPUT_HANDLE` is a Windows-specific constant.

The easiest way to set a color is to call the `SetConsoleTextAttribute` function and pass it the name of the handle to the output screen and a number that tells what color you want the output text to appear in. Table 5-4 shows the number that corresponds to each color.

Table 5-4 Windows Text Colors

Number	Text Color	Number	Text Color
0	Black	8	"Bright" Black
1	Blue	9	Bright Blue
2	Green	10	Bright Green
3	Cyan	11	Bright Cyan
4	Red	12	Bright Red
5	Purple	13	Bright Purple
6	Yellow	14	Bright Yellow
7	White	15	Bright White

Once you set a color it will remain in effect for all output text until you set a new one.

The following code segment shows how you can write the string "red" in red, "white" in white, "blue" in blue, and "bright yellow" in bright yellow.

```
SetConsoleTextAttribute(screen, 4);
cout << "Red" << endl;
SetConsoleTextAttribute(screen, 7);
cout << "White" << endl;
SetConsoleTextAttribute(screen, 1);
cout << "Blue" << endl;
SetConsoleTextAttribute(screen, 14);
cout << "Bright Yellow" << endl;
```

Here are two programs that use color. Neither one requires any input. Try running them to see their output displayed in color. Program 5-19 uses a loop to display `"Hello World"` on a black background in each of the 16 colors shown in Table 5-4.

Program 5-19

```
// This program demonstrates Windows functions to print colored
// text. It displays " Hello World!" in 16 different colors.
#include <iostream>
#include <windows.h>     // Needed to display colors and call Sleep
using namespace std;

int main()
{
   // Create a handle to the computer screen.
   HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);

   // Write 16 lines in 16 different colors.
   for (int color = 0; color < 16; color++)
   {
      SetConsoleTextAttribute (screen, color);
      cout << " Hello World!" << endl;
      Sleep(400);   // Pause between lines to watch them appear
   }
   // Restore the normal text color)
    SetConsoleTextAttribute(screen, 7);
    return 0;
}
```

Notice in Program 5-19 that each `cout` statement ended with an `endl`. This is needed to "flush" the buffer to ensure that all the output has been written to the screen before you change to another color. A `'\n'` will not work because it causes output to go to the next line, but does not flush the output buffer.

Program 5-20 provides another example of creating colorful output. It uses a loop to print the ABCs in color, alternating between bright green, red, and yellow.

Program 5-20

```
// This program writes the ABCs in green, red, and yellow.
#include <iostream>
#include <windows.h>     // Needed to display colors and call sleep
using namespace std;

int main()
{
   // Bright Green = 10   Bright Red = 12   Bright Yellow = 14

   // Get the handle to standard output device (the console)
   HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);

    // Write the ABCs using 3 colors
   int color = 10;     // Staring color = green
```

(program continues)

Program 5-20 *(continued)*

```
15    for (char letter = 'A'; letter <= 'Z'; letter++)
16 {
17       SetConsoleTextAttribute (screen, color); // Set the color
18       cout << letter << "  " << endl;          // Print the letter
19
20       color +=2;                               // Choose next color
21       if (color > 14)
22          color = 10;
23
24       Sleep(280);  // Pause between characters to watch them appear
25    }
26    // Restore normal text attribute (i.e. white)
27    SetConsoleTextAttribute(screen, 7);
28    return 0;
29 }
```

There are three important things to remember when working with colors:

- Include the `<windows.h>` header file.
- Follow each `cout` statement with an `endl`.
- Always set the text color back to normal (i.e., white) before quitting.

Review Questions and Exercises

Fill-in-the-Blank

1. To _____ a value means to increase it by one
2. To _____ a value means to decrease it by one.
3. When the increment or decrement operator is placed before the operand (or to the operand's left), the operator is being used in _____ mode.
4. When the increment or decrement operator is placed after the operand (or to the operand's right), the operator is being used in _____ mode.
5. The statement or block that is repeated is known as the _____ of the loop.
6. Each repetition of a loop is known as a(n) _____.
7. A loop that evaluates its test expression before each repetition is a(n) _____ loop.
8. A loop that evaluates its test expression after each repetition is a(n) _____ loop.
9. A loop that does not have a way of stopping is a(n) _____ loop.
10. A(n)_____ is a variable that "counts" the number of times a loop repeats.
11. A(n) _____ is a sum of numbers that accumulates with each iteration of a loop.
12. A(n) _____ is a variable that is initialized to some starting value, usually zero, and then has numbers added to it in each iteration of a loop.
13. A(n) _____ is a special value that marks the end of a series of values.
14. The _____ loop is ideal for situations that require a counter.

15. The _____ loop always iterates at least once.
16. The _____ and _____ loops will not iterate at all if their test expressions are false to start with.
17. Inside the `for` loop's parentheses, the first expression is the _____ , the second expression is the _____ , and the third expression is the _____.
18. A loop that is inside another is called a(n) _____ loop.
19. The _____ statement causes a loop to terminate immediately.
20. The _____ statement causes a loop to skip the remaining statements in the current iteration.

Algorithm Workbench

21. Write code that lets the user enter a number. The number should be multiplied by 2 and printed until the number exceeds 50. Use a `while` loop.
22. Write a `do-while` loop that asks the user to enter two numbers. The numbers should be added and the sum displayed. The user should be asked if he or she wishes to perform the operation again. If so, the loop should repeat; otherwise it should terminate.
23. Write a `for` loop that displays the following set of numbers:

```
0, 10, 20, 30, 40, 50 . . . 1000
```

24. Write a loop that asks the user to enter a number. The loop should iterate 10 times and keep a running total of the numbers entered.
25. Write a nested loop that displays the following ouput:

```
*****
*****
*****
```

26. Write a nested loop that displays 10 rows of '#' characters. There should be 15 '#' characters in each row.
27. Rewrite the following code, converting the `while` loop to a `do-while` loop:

```
char doAgain = 'y';
int sum = 0;

cout << "This code will increment sum 1 or more times.\n";
while ((doAgain == 'y') || (doAgain == 'Y'))
{  sum++;
   cout << "Sum has been incremented. Increment it again(y/n)? ";
   cin  >> doAgain;
}
cout << "Sum was incremented " << sum << " times.\n";
```

28. Rewrite the following code, replacing the `do-while` loop with a `while` loop. When you do this you will no longer need an `if` statement.

```
int number;
cout << "Enter an even number: ";
do
{  cin >> number;
   if (number % 2 != 0)
      cout << "Number must be even. Reenter number: ";
} while (number % 2 != 0);
```

29. Convert the following `while` loop to a `for` loop:

```
int count = 0;
while (count < 50)
{
   cout << "count is " << count << endl;
   count++;
}
```

30. Convert the following `for` loop to a `while` loop:

```
for (int x = 50; x > 0; x--)
{
   cout << x << " seconds to go.\n";
}
```

Predict the Output

What will each of the following program segments display?

31.
```
int x = 1;
while (x < 10);
    x++;
    cout << x;
```

32.
```
int x = 1;
while (x < 10)
    x++;
    cout << x;
```

33.
```
for (int count = 1; count <= 10; count++)
{  cout << ++count << "  ";   // This is a bad thing to do!
}
```

34.
```
for (int row = 1; row <= 3; row++)
{  cout << "\n$";
   for (int digit = 1; digit <= 4; digit++)
      cout << '9';
}
```

Find the Errors

35. Each of the programs in this section has errors. Find as many as you can.

A)
```
// Find the error in this program.
#include <iostream>
using namespace std;
int main()
{  int num1 = 0, num2 = 10, result;

   num1++;
   result = ++(num1 + num2);
   cout << num1 << " " << num2 << " " << result;
   return 0;
}
```

B)
```
// This program adds two numbers entered by the user.
#include <iostream>
using namespace std;
int main()
{  int num1, num2;
   char again;

   while ((again == 'y') || (again == 'Y'))
      cout << "Enter two numbers: ";
      cin  >> num1 >> num2;
      cout << "Their sum is << (num1 + num2) << endl;
      cout << "Do you want to do this again? ";
      cin  >> again;
      return 0;
}
```

36. A)
```
// This program uses a loop to raise a number to a power.
#include <iostream>
using namespace std;
int main()
{  int num, bigNum, power, count;

   cout << "Enter an integer: ";
   cin  >> num;
   cout << "What power do you want it raised to? ";
   cin  >> power;
   bigNum = num;
   while (count++ < power);
      bigNum *= num;
   cout << "The result is << bigNum << endl;
   return 0;
}
```

B)
```
// This program averages a set of numbers.
#include <iostream>
using namespace std;

int main()
{
   int numCount, total;
   double average;

   cout << "How many numbers do you want to average? ";
   cin  >> numCount;
   for (int count = 0; count < numCount; count++)
   {
      int num;
      cout << "Enter a number: ";
      cin  >> num;
      total += num;
      count++;
   }
   average = total / numCount;
   cout << "The average is << average << endl;
   return 0;
}
```

37. A)
```
// This program displays the sum of two numbers.
#include <iostream>
using namespace std;

int main()
{
   int choice, num1, num2;

   do
   {
      cout << "Enter a number: ";
      cin  >> num1;
      cout << "Enter another number: ";
      cin  >> num2;
      cout << "Their sum is " << (num1 + num2) << endl;
      cout << "Do you want to do this again?\n";
      cout << "1 = yes, 0 = no\n";
      cin  >> choice;
   } while (choice = 1)
   return 0;
}
```

B)
```
// This program displays the sum of the numbers 1 - 100.
#include <iostream>
using namespace std;

int main()
{
   int count = 1, total;

   while (count <= 100)
      total += count;
   cout << "The sum of the numbers 1 - 100 is ";
   cout << total << endl;
   return 0;
}
```

Soft Skills

Programmers not only need to be able to analyze what is wrong with a faulty algorithm, but also need to be able to explain the problem to others.

38. Write a clear problem description for a simple program and create a pseudocode solution for it. The pseudocode should incorporate the logic, including all the calculations, needed in the program, but should purposely contain a subtle logic error. Then pair up with another student in the class who has done the same thing and swap your work. Each of you should trace the logic to find the error in the pseudocode you are given, then clearly explain to your partner what the problem is, why the "code" will not work as written, and what should be done to correct it.

    As an alternative, your instructor may wish to provide you with a problem description and an incorrect pseudocode solution. Again, the goal is not only for you to find the error, but also to clearly explain what the problem is, why the "code" will not work as written, and what should be done to correct it.

Programming Challenges

1. Characters for the ASCII Codes

Write a program that uses a loop to display the characters for each ASCII code 32 through 127. Display 16 characters on each line with one space between characters.

2. Sum of Numbers

Write a program that asks the user for a positive integer value. The program should use a loop to get the sum of all the integers from 1 up to the number entered. For example, if the user enters 50, the loop will find the sum of 1, 2, 3, 4, … 50.

Input Validation: Do not accept an input that is less than 1.

3. Distance Traveled

The distance a vehicle travels can be calculated as follows:

```
distance = speed * time
```

For example, if a train travels 40 miles per hour for 3 hours, the distance traveled is 120 miles.

Write a program that asks the user for the speed of a vehicle (in miles per hour) and how many hours it has traveled. It should then use a loop to display the total distance traveled at the end of each hour of that time period. Here is an example of the output:

```
What is the speed of the vehicle in mph? 40
How many hours has it traveled? 3
Hour   Distance Traveled
--------------------------------
   1            40
   2            80
   3           120
```

Input Validation: Do not accept a negative number for speed and do not accept any value less than one for time traveled.

4. Celsius to Fahrenheit Table

In one of the Chapter 3 Programming Challenges you were asked to write a program that converts a Celsius temperature to Fahrenheit. Modify that program so it uses a loop to display a table of the Celsius temperatures from 0 to 20 and their Fahrenheit equivalents.

5. Speed Conversion Chart

Write a program that displays a table of speeds in kilometers per hour with their values converted to miles per hour. The table should display the speeds from 60 kilometers per hour through 130 kilometers per hour, in increments of 5 kilometers per hour. (In other words, it should display 60 kph, 65 kph, 70 kph and so forth, up through 130 kph.)

MPH = KPH * 0.6214



VideoNote
Solving the Ocean Levels Problem

6. Ocean Levels

Assuming the level of the Earth's oceans continues rising at about 3.1 millimeters per year, write a program that displays a table showing the total number of millimeters the oceans will have risen each year for the next 25 years.

7. Pennies for Pay

Write a program that calculates how much a person earns in a month if the salary is one penny the first day, two pennies the second day, four pennies the third day, and so on with the daily pay doubling each day the employee works. The program should ask the user for the number of days the employee worked during the month and should display a table showing how much the salary was for each day worked, as well as the total pay earned for the month. The output should be displayed in dollars with two decimal points, not in pennies.

Input Validation: Do not accept a number less than 1 or more than 31 for the number of days worked.

8. Calories Burned

Running on a particular treadmill you burn 3.9 calories per minute. Write a program that uses a loop to display the number of calories burned after 10, 15, 20, 25, and 30 minutes.

9. Membership Fees Increase

A country club, which currently charges $2500 per year for membership, has announced it will increase its membership fee by 4% each year for the next six years. Write a program that uses a loop to display the projected rates for the next six years.

10. Random Number Guessing Game

Write a program that generates a random number between 1 and 100 and asks the user to guess what the number is. If the user's guess is higher than the random number, the program should display `"Too high. Try again."` If the user's guess is lower than the random number, the program should display `"Too low. Try again."` The program should use a loop that repeats until the user correctly guesses the random number. Then the program should display `"Congratulations. You figured out my number."`

11. Random Number Guessing Game Enhancement

Enhance the program that you wrote for Programming Challenge 10 so it keeps a count of the number of guesses the user makes. When the user correctly guesses the random number, the program should display the number of guesses along with the message of congratulations.

12. The Greatest and Least of These

Write a program with a loop that lets the user enter a series of integers, followed by −99 to signal the end of the series. After all the numbers have been entered, the program should display the largest and smallest numbers entered.

13. Student Line Up

A teacher has asked all her students to line up single file according to their first name. For example, in one class Amy will be at the front of the line and Yolanda will be at the end. Write a program that prompts the user to enter the number of students in the class, then loops to read in that many names. Once all the names have been read in it reports which student would be at the front of the line and which one would be at the end of the line. You may assume that no two students have the same name.

Input Validation: Do not accept a number less than 1 or greater than 25 for the number of students.

14. Rate of Inflation

The annual rate of inflation is the rate at which money loses its value. For example, if the annual rate of inflation is 3.0%, then in one year it will cost $1030 to buy the goods that could have been purchased for $1000 today. Put another way, a year from now $1000 will only buy 1/1.03 * $1000, or $970.87, worth of goods. Two years from now $1000 will only buy only 1/1.03 of $970.87, or $942.59 worth of goods. Write a program that allows the user to enter an annual rate of inflation and then reports how much $1000 today will be worth each year for the next 10 years.

Input validation: Do not accept an inflation rate less than 1% or greater than 15%.

15. Population

Write a program that will predict the size of a population of organisms. The program should ask the user for the starting number of organisms, their average daily population increase (as a percentage of current population), and the number of days they will multiply. A loop should display the size of the population for each day.

Input Validation: Do not accept a number less than two for the starting size of the population. Do not accept a negative number for average daily population increase. Do not accept a number less than one for the number of days they will multiply.

16. Math Tutor

This program started in Chapter 3 and was modified in Chapter 4. Starting with the vesion described in Chapter 4, modify the program again so it displays a menu allowing the user to select an addition, subtraction, or multiplication problem. The final selection on the menu should let the user quit the program. After the user has finished the math problem, the program should display the menu again. This process is repeated until the user chooses to quit the program.

Input Validation: If the user selects an item not on the menu, display an error message and display the menu again.

17. Hotel Suites Occupancy

Write a program that calculates the occupancy rate of the 120 suites (20 per floor) located on the top 6 floors of a 15-story luxury hotel. These are floors 10–12 and 14–16 because, like many hotels, there is no 13th floor. Solve the problem by using a *single* loop that loops once for each floor between 10 and 16 and, on each iteration, asks the user to input the number of suites occupied on that floor. After all the iterations, the program should display how many suites the hotel has, how many of them are occupied, and what percentage of them are occupied.

> *Input Validation: Do not accept a value less than 0 or more than 20 for the number of suites occupied on a given floor.*

18. Rectangle Display

Write a program that asks the user for two positive integers between 2 and 10 to use for the length and width of a rectangle. If the numbers are different, the larger of the two numbers should be used for the length and the smaller for the width. The program should then display a rectangle of this size on the screen using the character 'X'. For example, if the user enters either 2 5 or 5 2, the program should display the following:

```
xxxxx
xxxxx
```

19. Diamond Display

Write a program that uses nested loops to display the diamond pattern shown below.

```
   +
  +++
 +++++
+++++++
 +++++
  +++
   +
```

20. Triangle Display

Write a program that uses nested loops to display the triangle pattern shown below.

```
+
+++
+++++
+++++++
+++++
+++
+
```

21. Arrowhead Display

Write a program that uses nested loops to display the arrowhead pattern shown below.

```
       +
       +++
       +++++
+++++++++++++
       +++++
       +++
       +
```



22. Sales Bar Chart

Write a program that asks the user to enter today's sales rounded to the nearest $100 for each of three stores. The program should then display a bar graph comparing each store's sales. Create each bar in the graph by displaying a row of asterisks. Each asterisk should represent $100 of sales.

Here is an example of the program's output. User input is shown in bold.

```
Enter today's sales for store 1: 1000[Enter]
Enter today's sales for store 2: 1200[Enter]
Enter today's sales for store 3: 900[Enter]

    DAILY SALES
  (each * = $100)
Store 1: **********
Store 2: ************
Store 3: *********
```

Input Validation: Do not accept sales figures less than $0.00.

23. Population Bar Chart

Write a program that produces a bar chart showing the population growth of Prairieville, a small town in the Midwest, at 20 year intervals during the past 100 years. The program should read in the population figures (rounded to the nearest 1000 people) for 1900, 1920, 1940, 1960, 1980, and 2000 from a file. For each year it should display the date and a bar consisting of one asterisk for each 1000 people. The data can be found in the `people.dat` file.

Here is an example of how the chart might begin:

```
PRAIRIEVILLE POPULATION GROWTH
(each * represents 1000 people)

1900  **
1920  ****
1940  *****
```

24. Payroll Report

Write a program that displays a weekly payroll report. A loop in the program should ask the user for the employee number, gross pay, state tax, federal tax, and FICA withholdings. The loop will terminate when 0 is entered for the employee number. After the data is entered, the program should display totals for gross pay, state tax, federal tax, FICA withholdings, and net pay.

Input Validation: Do not accept negative numbers for any of the items entered. If the state tax + federal tax + FICA withholdings for any employee is greater than gross pay, print an error message and ask the user to reenter the data for that employee.

25. Savings Account Balance

Write a program that calculates the balance of a savings account at the end of a three-month period. It should ask the user for the starting balance and the annual interest rate. A loop should then iterate once for every month in the period, performing the following:

A) Ask the user for the total amount deposited into the account during that month. Do not accept negative numbers. This amount should be added to the balance.
B) Ask the user for the total amount withdrawn from the account during that month. Do not accept negative numbers or numbers greater than the balance after the deposits for the month have been added in.
C) Calculate the interest for that month. The monthly interest rate is the annual interest rate divided by 12. Multiply the monthly interest rate by the average of that month's starting and ending balance to get the interest amount for the month. This amount should be added to the balance.

After the last iteration, the program should display a final report that includes the following information:

- starting balance at the beginning of the three-month period.
- total deposits made during the three months
- total withdrawals made during the three months
- total interest posted to the account during the three months
- final balance.

26. Using Files—Student Line Up

Modify the Student Line Up program described in Programming Challenge 13 so that it gets the names from a data file. Names should be read in until there is no more data to read. Data to test your program can be found in the `LineUp.dat` file.

27. Using Files—Savings Account Balance Modification

Modify the Savings Account Balance program described in Programming Challenge 25 so that it writes the final report to a file. After the program runs, print the file to hand in to your instructor.

CHAPTER 6

Functions

TOPICS

6.1 Modular Programming
6.2 Defining and Calling Functions
6.3 Function Prototypes
6.4 Sending Data into a Function
6.5 Passing Data by Value
6.6 The `return` Statement
6.7 Returning a Value from a Function
6.8 Returning a Boolean Value
6.9 Using Functions in a Menu-Driven Program
6.10 Local and Global Variables
6.11 Static Local Variables
6.12 Default Arguments
6.13 Using Reference Variables as Parameters
6.14 Overloading Functions
6.15 The `exit()` Function
6.16 Stubs and Drivers
6.17 Little Lotto Case Study
6.18 Tying It All Together: *Glowing Jack-o-lantern*

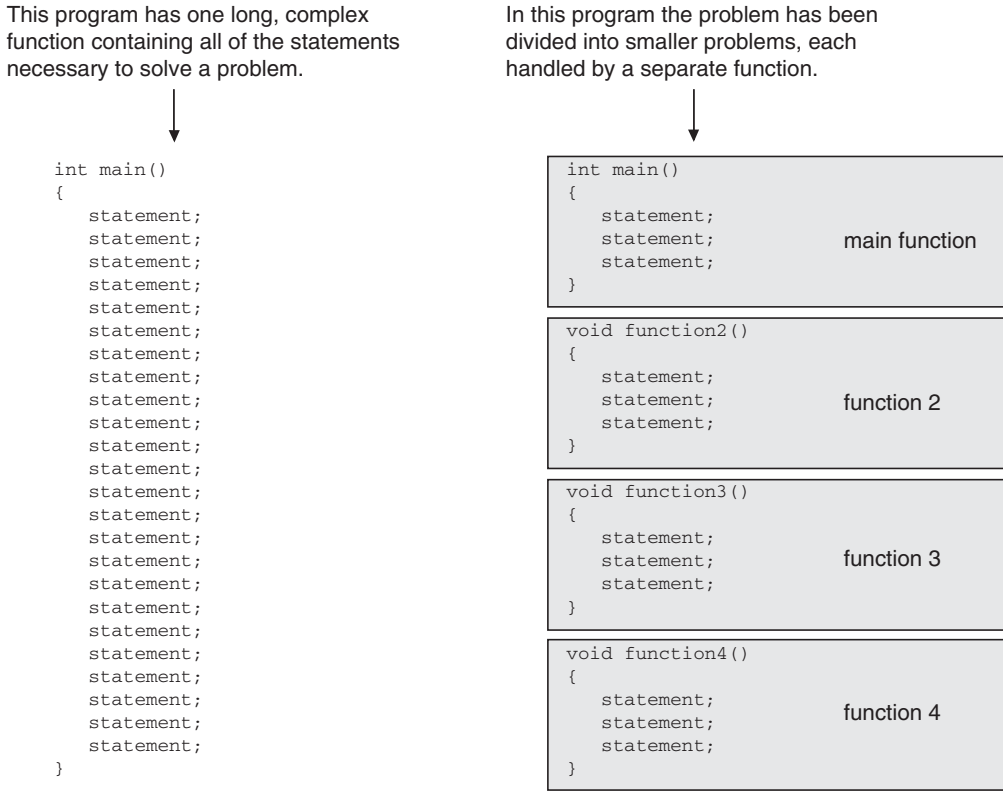
6.1 Modular Programming

CONCEPT: A program may be broken up into a set of manageable functions, or modules. This is called modular programming.

A function is a collection of statements that performs a specific task. So far you have used functions in two ways: 1) you have created a function called `main` in every program you've written, and 2) you have called library functions such as `pow` and `sqrt`. In this chapter you will learn how to create your own functions that can be used like library functions.

Functions are commonly used to break a problem down into small manageable pieces, or modules. Instead of writing one long function that contains all the statements necessary to solve a problem, several smaller functions can be written, with each one solving a specific part of the problem. These small functions can then be executed in the desired order to solve the problem. This approach is sometimes called *divide and conquer* because a large problem is divided into several smaller problems that are more easily solved. Figure 6-1 illustrates this idea by comparing two programs, one that uses a single module containing all of the statements necessary to solve a problem, and another that divides a problem into a set of smaller problems, each handled by a separate function.

Figure 6-1



Another reason to write functions is that they simplify programs. If a specific task is performed in several places in a program, a function can be written once to perform that task, and then be executed anytime it is needed. This benefit of using functions is known as *code reuse* because you are writing the code to perform a task once and then reusing it each time you need to perform the task.

6.2 Defining and Calling Functions

CONCEPT: A function call is a statement that causes a function to execute. A function definition contains the statements that make up the function.



VideoNote
Defining and Calling Functions

When creating a function, you must write its *definition*. All function definitions have the following parts:

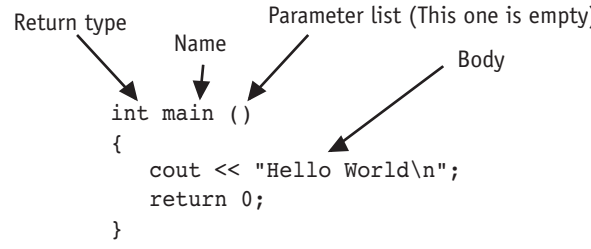
Name — Every function must have a name. In general, the same rules that apply to variable names also apply to function names.

Parameter list — The program module that calls a function can send data to it. The parameter list is the list of variables that hold the values being passed to the function. If no values are being passed to the function, its parameter list is *empty*.

Body	The body of a function is the set of statements that carry out the task the function is performing. These statements are enclosed in a set of braces.
Return type	A function can send a value back to the program module that called it. The return type is the data type of the value being sent back.

Figure 6-2 shows the definition of a simple function with the various parts labeled. Notice that the function's return type is actually listed first.

Figure 6-2



The line in the definition that reads `int main ()` is called the *function header.*

Void Functions

You already know that a function can return a value. The `main` function in all of the programs you have seen in this book is declared to return an `int` value to the operating system. The `return 0;` statement causes the value 0 to be returned when the `main` function finishes executing.

It isn't necessary for all functions to return a value, however. Some functions simply perform one or more statements and then return. In C++ these are called *void functions*. The `displayMessage` function shown here is an example:

```
void displayMessage()
{
   cout << "Hello from the function displayMessage.\n";
}
```

The function's name is `displayMessage`. This name is descriptive, as function names should be. It gives an indication of what the function does. It displays a message. Notice the function's return type is `void`. This means the function does not send back a value when it has finished executing and returns to the part of the program that invoked it. Because no value is being sent back, no `return` statement is required. When the statements in the function have finished executing and the right brace that ends the function is encountered, the program automatically returns.

Calling a Function

A function is executed when it is *called*. Function `main` is called automatically when a program starts, but all other functions must be executed by *function call* statements. When a function is called, the program branches to that function and executes the

statements in its body. Let's look at Program 6-1, which contains two functions: `main` and `displayMessage`.

Program 6-1

```
1  // This program has two functions: main and displayMessage.
2  #include <iostream>
3  using namespace std;
4
5  /**************************************
6   *           displayMessage           *
7   *  This function displays a greeting. *
8   **************************************/
9  void displayMessage()
10 {
11    cout << "Hello from the function displayMessage.\n";
12 }
13
14 /**************************************
15  *               main                 *
16  **************************************/
17 int main()
18 {
19    cout << "Hello from main.\n";
20    displayMessage();       // Call displayMessage
21    cout << "Back in function main again.\n";
22    return 0;
23 }
```

Program Output

```
Hello from main.
Hello from the function displayMessage.
Back in function main again.
```

The function `displayMessage` is called by the following statement in line 20:

```
displayMessage();
```

This statement is the function call. It is simply the name of the function followed by a set of parentheses and a semicolon. Let's compare this with the function header:

Function Header ⟶ `void displayMessage()`
Function Call ⟶ `displayMessage();`

The function header is part of the function definition. It declares the function's return type, name, and parameter list. It must *not* be terminated with a semicolon because the definition of the function's body follows it.

The function call is a statement that executes the function, so it *is* terminated with a semicolon like all other C++ statements. Notice that the function call does not list the return type and, if the program is not passing data into the function, the parentheses are left empty.



NOTE: Later in this chapter you will see how data can be passed into a function by being listed inside the parentheses.

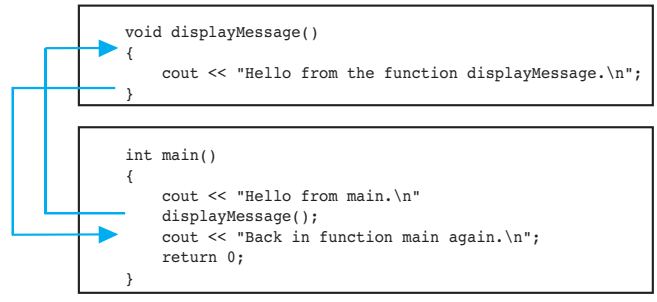
Even though the program starts executing at `main`, the function `displayMessage` is defined first. This is because before a function can be called, the compiler must know certain things about it. It must know the function's name, its return type, how many parameter variables it has, and what the data type of each of these variables is. One way to ensure the compiler will know this information is to place the function definition before all calls to that function. (Later you will see an alternative and preferred method of accomplishing this.)



NOTE: You should always document your functions by writing comments that describe what they do. These comments should appear just before the function definition.

Notice how Program 6-1 flows. It starts, of course, in function `main`. When the call to `displayMessage` is encountered, the program branches to that function and performs its statements. Once `displayMessage` has finished executing, the program branches back to function `main` and resumes with the line that follows the function call. This is illustrated in Figure 6-3.

Figure 6-3



Function call statements may be used in control structures such as loops, `if` statements, and `switch` statements. Program 6-2 places the `displayMessage` function call inside a loop.

Program 6-2

```
1  // The function displayMessage is repeatedly called from within a loop.
2  #include <iostream>
3  using namespace std;
4
5  /**************************************
6   *           displayMessage           *
7   *  This function displays a greeting. *
8   **************************************/
9  void displayMessage()
10 {
11    cout << "Hello from the function displayMessage.\n";
12 }
13
```

(program continues)

Program 6-2 *(continued)*

```
14 /***************************************
15  *               main                  *
16  ***************************************/
17 int main()
18 {
19    cout << "Hello from main.\n";
20
21    for (int count = 0; count < 3; count++)
22        displayMessage();   // Call displayMessage
23
24    cout << "Back in function main again.\n";
25    return 0;
26 }
```

Program Output

```
Hello from main.
Hello from the function displayMessage.
Hello from the function displayMessage.
Hello from the function displayMessage.
Back in function main again.
```

It is possible to have many functions and function calls in a program. Program 6-3 has three functions: `main`, `first`, and `second`.

Program 6-3

```
1  // This program has three functions: main, first, and second.
2  #include <iostream>
3  using namespace std;
4
5  /***************************************
6   *                first                *
7   * This function displays a message.   *
8   ***************************************/
9  void first()
10 {
11    cout << "I am now inside the function first.\n";
12 }
13
14 /***************************************
15  *                second               *
16  * This function displays a message.   *
17  ***************************************/
18 void second()
19 {
20    cout << "I am now inside the function second.\n";
21 }
22
23 /***************************************
24  *                main                 *
25  ***************************************/
```

(program continues)

Program 6-3 *(continued)*

```
26 int main()
27 {
28    cout << "I am starting in function main.\n";
29    first();    // Call function first
30    second();   // Call function second
31    cout << "Back in function main again.\n";
32    return 0;
33 }
```

Program Output

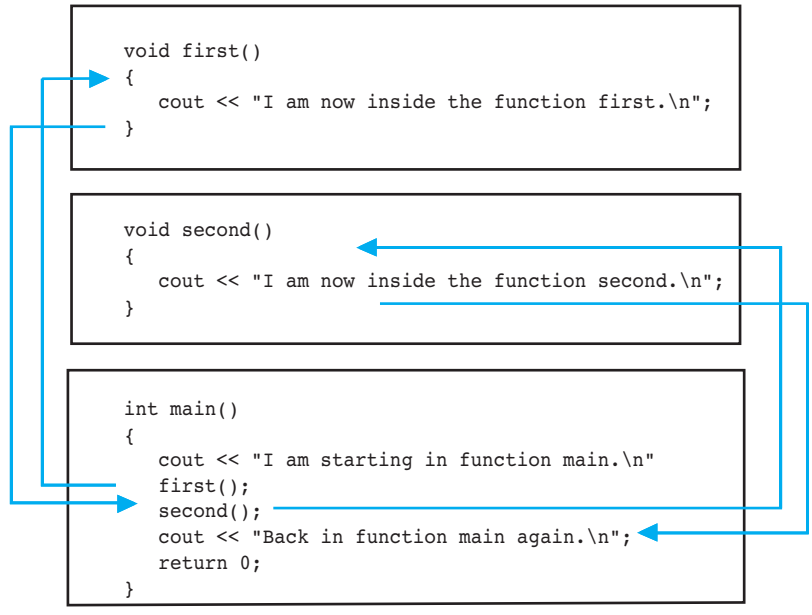
```
I am starting in function main.
I am now inside the function first.
I am now inside the function second.
Back in function main again.
```

In lines 29 and 30 of Program 6-3, function `main` contains a call to `first` and a call to `second`:

```
first();
second();
```

Each call statement causes the program to branch to a function and then back to `main` when the function is finished. Figure 6-4 illustrates the paths taken by the program.

Figure 6-4



Functions may also be called in a hierarchical, or layered, fashion. This is demonstrated by Program 6-4, which has three functions: `main`, `deep`, and `deeper`.

Program 6-4

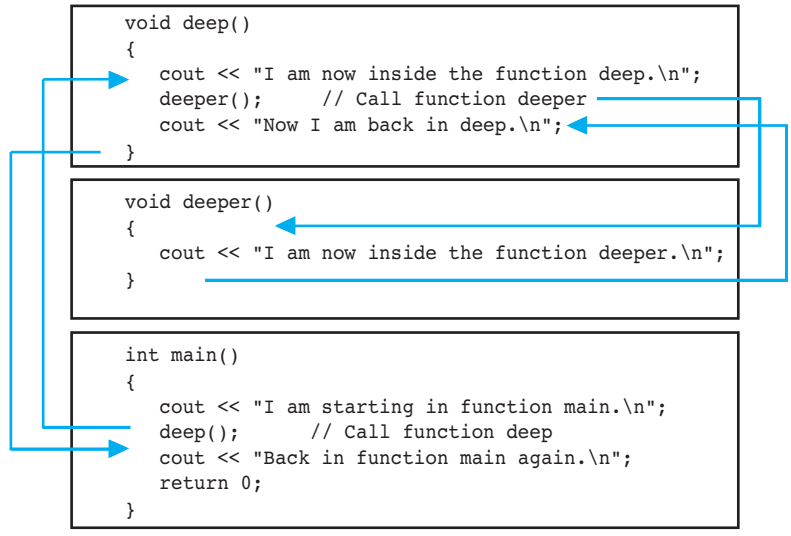
```
1  // This program has three functions: main, deep, and deeper
2  #include <iostream>
3  using namespace std;
4
5  /**************************************
6   *                deeper              *
7   * This function displays a message.  *
8   **************************************/
9  void deeper()
10 {
11    cout << "I am now inside the function deeper.\n";
12 }
13
14 /**************************************
15  *                 deep               *
16  * This function displays a message.  *
17  **************************************/
18 void deep()
19 {
20    cout << "I am now inside the function deep.\n";
21    deeper();  // Call function deeper
22    cout << "Now I am back in deep.\n";
23 }
24
25 /**************************************
26  *                 main               *
27  **************************************/
28 int main()
29 {
30    cout << "I am starting in function main.\n";
31    deep();    // Call function deep
32    cout << "Back in function main again.\n";
33    return 0;
34 }
```

Program Output

```
I am starting in function main.
I am now inside the function deep.
I am now inside the function deeper.
Now I am back in deep.
Back in function main again.
```

In Program 6-4, function `main` only calls the function `deep`. In turn, `deep` calls `deeper`. The paths taken by the program are shown in Figure 6-5.

Figure 6-5



Checkpoint

6.1 Is the following a function header or a function call?

```
calcTotal();
```

6.2 Is the following a function header or a function call?

```
void showResults()
```

6.3 What will the output of the following program be if the user enters 10?

```
#include <iostream>
using namespace std;

void func1()
{
   cout << "Able was I\n";
}

void func2()
{
   cout << "I saw Elba\n";
}

int main()
{
   int input;
   cout << "Enter a number: ";
   cin  >> input;
   if (input < 10)
   {
      func1();
      func2();
   }
```

```
   else
   {
      func2();
      func1();
   }
   return 0;
}
```

6.4 The following program skeleton determines whether a person qualifies for a credit card. To qualify, the person must have worked on his or her current job for at least two years and make at least $17,000 per year. Finish the program by writing the definitions of the functions `qualify` and `noQualify`. The function `qualify` should explain that the applicant qualifies for the card and that the annual interest rate is 12 percent. The function `noQualify` should explain that the applicant does not qualify for the card and give a general explanation why.

```
#include <iostream>
using namespace std;

// You must write definitions for the two functions qualify
// and noQualify.

int main()
{
   double salary;
   int years;

   cout << "This program will determine if you qualify\n";
   cout << "for our credit card.\n";
   cout << "What is your annual salary? ";
   cin  >> salary;
   cout << "How many years have you worked at your ";
   cout << "current job? ";
   cin  >> years;

   if (salary >= 17000.0 && years >= 2)
       qualify();
   else
       noQualify();
   return 0;
}
```

6.3 Function Prototypes

CONCEPT: A function prototype eliminates the need to place a function definition before all calls to the function.

Before the compiler encounters a call to a particular function, it must already know certain things about the function. In particular, it must know the number of parameters the function uses, the type of each parameter, and the return type of the function. Parameters allow information to be sent to a function. Certain return types allow information to be returned

from a function. You will learn more about parameters and return types in later sections of this chapter. For now, the functions we will use will have no parameters and, except for `main`, will have a return type of `void`.

One way of ensuring that the compiler has this required information is to place the function definition before all calls to that function. This was the approach taken in Programs 6-1 through 6-4. Another method is to declare the function with a *function prototype*. Here is a prototype for the `displayMessage` function in Program 6-1:

```
void displayMessage();
```

This prototype looks similar to the function header, except there is a semicolon at the end. The statement tells the compiler that the function `displayMessage` uses no parameters and has a `void` return type, meaning it doesn't return a value.



NOTE: Function prototypes are also known as *function declarations*.



WARNING! You must either place the function definition or the function prototype ahead of all calls to the function. Otherwise the program will not compile.

Function prototypes are usually placed near the top of a program so the compiler will encounter them before any function calls. Program 6-5 is a modification of Program 6-3. The definitions of the functions `first` and `second` have been placed after `main`, and their function prototypes have been placed above `main`, directly after the `using namespace std` statement.

Program 6-5

```
1  // This program has three functions: main, first, and second.
2  // It uses function prototypes.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototypes
7  void first();
8  void second();
9
10 int main()
11 {
12    cout << "I am starting in function main.\n";
13    first();    // Call function first
14    second();   // Call function second
15    cout << "Back in function main again.\n";
16    return 0;
17 }
18
19 /**************************************
20  *               first                *
21  * This function displays a message.   *
22  **************************************/
```

(program continues)

Program 6-5 *(continued)*

```
23 void first()
24 {
25    cout << "I am now inside the function first.\n";
26 }
27
28 /**************************************
29  *                second                *
30  * This function displays a message.   *
31  **************************************/
32 void second()
33 {
34    cout << "I am now inside the function second.\n";
35 }
```

Program Output is the same as the output of Program 6-3.

When the compiler is reading Program 6-5, it encounters the calls to the functions `first` and `second` in lines 13 and 14 before it has read the definition of those functions. Because of the function prototypes, however, the compiler already knows the return type and parameter information of `first` and `second`. There should be a prototype for each function in a program except `main`. A prototype is never needed for `main` because it is the starting point of the program.



NOTE: Although some programmers make `main` the last function in the program, many prefer it to be first because it is the program's starting point.

6.4 Sending Data into a Function

CONCEPT: When a function is called, the program may send values into the function.



VideoNote
Using Function Arguments

Values that are sent into a function are called *arguments*. You're already familiar with how to use arguments in a function call. In the following statement the function `pow` is being called with two arguments, 2.0 and 4.0 passed to it:

```
result = pow(2.0, 4.0);
```

A *parameter* is a special variable that holds a value being passed as an argument into a function. By using parameters, you can design your own functions that accept data this way. Here is the definition of a function that uses a parameter:

```
void displayValue(int num)
{
   cout << "The value is " << num << endl;
}
```

Notice the integer variable definition inside the parentheses `(int num)`. The variable `num` is a parameter. This enables the function `displayValue` to accept an integer value as an argument. Program 6-6 is a complete program that uses this function.



NOTE: In this text, the values that are passed into a function are called arguments, and the variables that receive those values are called parameters. There are several variations of these terms in use. Some call the arguments *actual parameters* and call the parameters *formal parameters*. Others use the terms *actual arguments* and *formal arguments*. Regardless of which set of terms you use, it is important to be consistent.

Program 6-6

```
1  // This program demonstrates a function with a parameter.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype
6  void displayValue(int num);
7
8  int main()
9  {
10    cout << "I am passing 5 to displayValue.\n";
11    displayValue(5);  // Call displayValue with argument 5
12    cout << "Now I am back in main.\n";
13    return 0;
14 }
15
16 /*****************************************
17  *               displayValue            *
18  *  This function uses an integer parameter *
19  *  whose value is displayed.            *
20  *****************************************/
21 void displayValue(int num)
22 {
23    cout << "The value is " << num << endl;
24 }
```

Program Output

```
I am passing 5 to displayValue.
The value is 5
Now I am back in main.
```

Notice the function prototype for `displayValue` in line 6:

```
void displayValue(int num);      // Function prototype
```

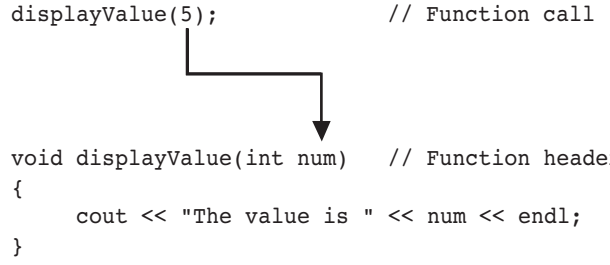
It lists both the data type and the name of the function's parameter variable. However, it is not actually necessary to list the name of the parameter variable inside the parentheses. Only the data type of the variable is required. The function prototype could have been written like this:

```
void displayValue(int);          // Function prototype
```

Because some instructors prefer that you list only the data type for each parameter in a function prototype, while others prefer that you list both the data type and name, we use both versions throughout this book. Your instructor will tell you which version to use.

In Program 6-6 the `displayValue` function is called in line 11 of `main` with the argument 5 inside the parentheses. The number 5 is passed into `num`, which is `displayValue's` parameter. This is illustrated in Figure 6-6.

Figure 6-6



Any argument listed inside the parentheses of a function call is copied into the function's parameter variable. In essence, parameter variables are initialized to the value of the corresponding arguments passed to them when the function is called. Program 6-7 shows the function `displayValue` being called several times with a different argument passed each time.

Program 6-7

```
1  // This program demonstrates a function with a parameter.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype
6  void displayValue(int num);
7
8  int main()
9  {
10    cout << "I am passing several values to displayValue.\n";
11    displayValue(5);          // Call displayValue with argument 5
12    displayValue(10);         // Call displayValue with argument 10
13    displayValue(2);          // Call displayValue with argument 2
14    displayValue(16);         // Call displayValue with argument 16
15    cout << "Now I am back in main.\n";
16    return 0;
17 }
18
19 /*****************************************
20  *               displayValue            *
21  *  This function uses an integer parameter *
22  *  whose value is displayed.            *
23  *****************************************/
24 void displayValue(int num)
25 {
26    cout << "The value is " << num << endl;
27 }
```

(program continues)

Program 6-7 *(continued)*

Program Output

```
I am passing several values to displayValue.
The value is 5
The value is 10
The value is 2
The value is 16
Now I am back in main.
```

In lines 11–14 of Program 6-7 the `displayValue` function is called four times, and each time `num` takes on a different value. Any expression whose value could normally be assigned to `num` may be used as an argument. For example, the following function call would pass the value 8 into `num`:

```
displayValue(3 + 5);
```

When a function is called, it is best if each argument passed to it has the same data type as the parameter receiving it. However, it is possible to send an argument with a different data type. In this case, the argument will be promoted or demoted to match the data type of the parameter receiving it. Be very careful if you do this, as you may introduce a hard to find bug. For example, the `displayValue` function in Program 6-7 has an integer parameter, which means it expects to receive an integer value. If the function is called as shown here,

```
displayValue(4.7);
```

the argument will be truncated and the integer 4 will be stored in the parameter `num`.

Often it is useful to pass several arguments into a function. Program 6-8 includes a function that has three parameters. Look carefully at how these three parameters are defined in the function header in line 27, as well as in the function prototype in line 6. In this case the prototype lists variable names along with the data types of the function's parameters. This makes the prototype look exactly like the function header except that the prototype *must* end with a semicolon and the function header *must not* end with a semicolon.

```
// Prototype
   void showSum(int num1, int num2, int num3);  // Ends with ;

// Function header
   void showSum(int num1, int num2, int num3)   // NO ;
```

Notice also that the call to the function, in line 18 of the program, now must send three arguments to the function.

```
showSum(value1, value2, value3);
```

Program 6-8

```
1 // This program demonstrates a function with three parameters.
2 #include <iostream>
3 using namespace std;
4
```

(program continues)

Program 6-8 *(continued)*

```
// Function prototype
void showSum(int num1, int num2, int num3);

int main()
{
   int value1, value2, value3;

   // Get 3 integers
   cout << "Enter three integers and I will display ";
   cout << "their sum: ";
   cin  >> value1 >> value2 >> value3;

   // Call showSum, passing 3 arguments.
   showSum(value1, value2, value3);
   return 0;
}

/*****************************************
 *                 showSum                *
 *  This function displays the sum of the   *
 *  3 integers passed into its parameters.  *
 *****************************************/
void showSum(int num1, int num2, int num3)
{
   cout << "The sum is " << (num1 + num2 + num3) << endl;
}
```

Program Output with Example Input Shown in Bold

```
Enter three integers and I will display their sum: 4 8 7[Enter]
The sum is 19
```

One important point to mention about Program 6-8 is how the `showSum` parameter variables are defined in its function header.

```
void showSum(int num1, int num2, int num3)
```

As you might expect they are each preceded by their data type and they are separated by commas. However, unlike regular variable definitions, they cannot be combined into a single definition even if they all have the same data type. That is, even though all three parameter variables are integers, they *cannot* be defined like this:

```
void showSum(int num1, num2, num3)    // Error!
```

Another point to notice is that whereas the function prototype and function header *must* list the data type of each parameter, the call to the function *must not* list any data types. Each argument in the function call must be a value or something that can be evaluated to produce a value. If `value1`, `value2`, and `value3` hold the values `4`, `8`, and `7` respectively, as they did in the sample run for Program 6-8, the following three function calls would all be legal and would cause the `showSum` function to display the same thing.

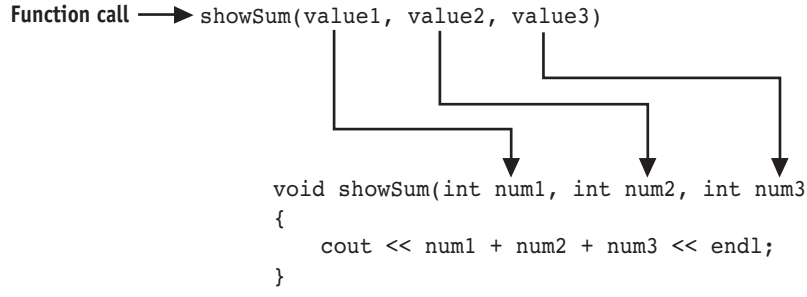
```
showSum(value1, value2, value3);    // Legal The sum is 19
showSum(4, 8, 7);                   // Legal The sum is 19
showSum(3+1, 16/2, 7);              // Legal The sum is 19
```

But the following function call would cause an error.

```
showSum(int value1, int value2, int value3);     // Error!
```

Figure 6-7 shows the difference in the syntax between the function call and the function header when variables are used as arguments. It also illustrates that when a function with multiple parameters is called, the arguments are passed to the parameters in order.

Figure 6-7



The following function call will cause 4 to be passed into the `num1` parameter, 8 to be passed into `num2`, and 7 to be passed into `num3`:

```
showSum(4, 8, 7);
```

However, the following function call will cause 7 to be passed into the `num1` parameter, 8 to be passed into `num2`, and 4 to be passed into `num3`:

```
showSum(7, 8, 4);
```



NOTE: Like all variables, parameters have a scope. The scope of a parameter is limited to the body of the function which uses it.

6.5 Passing Data by Value

CONCEPT: **When an argument is passed into a parameter by value, only a copy of the argument's value is passed. Changes to the parameter do not affect the original argument.**

As you have seen in this chapter, parameters are special-purpose variables that are defined inside the parentheses of a function definition. Their purpose is to hold the information passed to them by the arguments, which are listed inside the parentheses of a function call. Normally when information is passed to a function it is *passed by value*. This means the parameter receives a copy of the value that is passed to it. If a parameter's value is changed inside a function, it has no effect on the original argument. Program 6-9 demonstrates this concept.

It also illustrates that when a function prototype lists variable names along with data types, the names it uses are just *dummy names*. That is, they are not actually used by the compiler and do not have to agree with the names used in the function header. If you compare the `changeMe` function prototype in line 7 with the `changeMe` function header in line 29, you

will notice they both specify that the function has one `int` parameter. However, the name used for it in the prototype does not match the name used for it in the function header.

Program 6-9

```
1  // This program demonstrates that changes to a function
2  // parameter have no effect on the original argument.
3  #include <iostream>
4  using namespace std;
5
6  // Function Prototype
7  void changeMe(int aValue);
8
9  int main()
10 {
11    int number = 12;
12
13    // Display the value in number
14    cout << "In main number is " << number << endl;
15
16    // Call changeMe, passing the value in number as an argument
17    changeMe(number);
18
19    // Display the value in number again
20    cout << "Back in main again, number is still " << number << endl;
21    return 0;
22 }
23
24 /***********************************
25  *              changeMe           *
26  *  This function changes the value  *
27  *  stored in its parameter myValue  *
28  ***********************************/
29 void changeMe(int myValue)
30 {
31    // Change the value of myValue to 0
32    myValue = 0;
33
34    // Display the value in myValue
35    cout << "In changeMe, the value has been changed to "
36         << myValue << endl;
37 }
```

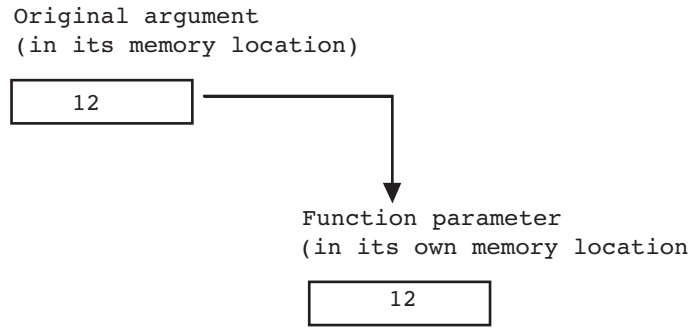
Program Output

```
In main number is 12
In changeMe, the value has been changed to 0
Back in main again, number is still 12
```

Even though the parameter variable `myValue` is changed in the `changeMe` function, the argument `number` is not modified. This occurs because the `myValue` variable contains only a copy of the `number` variable. Just this copy is changed, not the original. The `changeMe` function does not have access to the original argument.

Figure 6-8 illustrates that a parameter variable's storage location in memory is separate from that of the original argument.

Figure 6-8





NOTE: Later in this chapter you will learn ways to give a function access to its original arguments.



Checkpoint

6.5 Indicate which of the following is the function prototype, the function header, and the function call:

```
void showNum(double num)
void showNum(double);
showNum(45.67);
```

6.6 Write a function named `timesTen`. The function should have an integer parameter named `number`. When `timesTen` is called, it should display the product of `number` times 10. (Note: just write the function. Do not write a complete program.)

6.7 Write a function prototype for the `timesTen` function you wrote in question 6.6.

6.8 What is the output of the following program?

```
#include <iostream>
using namespace std;

void showDouble(int value); // Function prototype

int main()
{
   int num;

   for (num = 0; num < 10; num++)
      showDouble(num);
   return 0;
}

// Definition of function showDouble
void showDouble(int value)
{
   cout << value << "   " << (value * 2) << endl;
}
```

6.9 What is the output of the following program?

```
#include <iostream>
using namespace std;

void func1(double, int); // Function prototype

int main()
{
   int x = 0;
   double y = 1.5;

   cout << x << " " << y << endl;
   func1(y, x);
   cout << x << " " << y << endl;
   return 0;
}
void func1(double a, int b)
{
   cout << a << " " << b << endl;
   a = 0.0;
   b = 10;
   cout << a << " " << b << endl;
}
```

6.10 The following program skeleton asks for the number of hours you've worked and your hourly pay rate. It then calculates and displays your wages. The function `showDollars`, which you are to write, formats the output of the wages.

```
#include <iostream>
#include <iomanip>
using namespace std;

void showDollars(double pay); // Function prototype

int main()
{
   double payRate, hoursWorked, wages;

   cout << "How many hours have you worked? "
   cin  >> hoursWorked;
   cout << "What is your hourly pay rate? ";
   cin  >> payRate;
   wages = hoursWorked * payRate;
   showDollars(wages);
   return 0;
}

// You must write the definition of the function showDollars
// here. It should take one parameter of the type double.
// The function should display the message "Your wages are $"
// followed by the value of the parameter. It should be displayed
// with 2 places of precision after the decimal point, in fixed
// notation, and the decimal point should always display.
```

6.6 The return Statement

CONCEPT: The **return** statement causes a function to end immediately.

When the last statement in a function has finished executing, the function terminates. The program returns to the module that called it and continues executing from the point immediately following the function call. It is possible, however, to force a function to return to where it was called from before its last statement has been executed. This can be done with the `return` statement, as illustrated in Program 6-10. In this program, the function `divide` shows the quotient of `arg1` divided by `arg2`. If `arg2` is set to zero, however, the function returns without performing the division.

Program 6-10

```
1  // This program uses a function to perform division.
2  // It illustrates the return statement.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype
7  void divide(double arg1, double arg2);
8
9  int main()
10 {
11    double num1, num2;
12
13    cout << "Enter two numbers and I will divide the first\n";
14    cout << "number by the second number: ";
15    cin  >> num1 >> num2;
16    divide(num1, num2);
17    return 0;
18 }
19
20 /**********************************************************
21  *                        divide                          *
22  *  This function uses two parameters, arg1 and arg2.     *
23  *  If arg2 does not = zero, the function displays the    *
24  *  result of arg1/arg2. Otherwise it returns without     *
25  *  performing the division.                              *
26  **********************************************************/
27 void divide(double arg1, double arg2)
28 {
29    if (arg2 == 0.0)
30    {
31       cout << "Sorry, I cannot divide by zero.\n";
32       return;
33    }
34    cout << "The quotient is " << (arg1 / arg2) << endl;
35 }
```

(program continues)

Program 6-10 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter two numbers and I will divide the first
number by the second number: 12 0[Enter]
Sorry, I cannot divide by zero.
```

In the example running of the program, the user entered 12 and 0 as input. These were stored as double values is variables `num1` and `num2`. In line 16 the `divide` function was called, passing 12.0 into the `arg1` parameter and 0.0 into the `arg2` parameter. Inside the `divide` function, the `if` statement in line 29 executes. Because `arg2` is equal to 0.0, the code in lines 31 and 32 execute. When the `return` statement in line 32 executes, the `divide` function immediately ends. This means the `cout` statement in line 34 does not execute. The program resumes at line 17 in the `main` function.

6.7 Returning a Value from a Function

CONCEPT: A function may send a value back to the part of the program that called the function.



VideoNote
Value-Returning Functions

You've seen that data may be passed into a function by way of parameter variables. Data may also be returned from a function back to the statement that called it. Functions that return a value are known as *value-returning functions*.

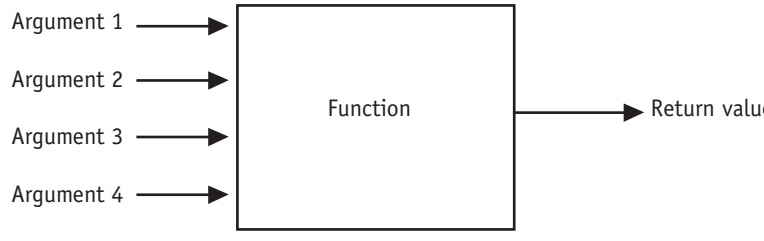
The `pow` function, which you have already used, is an example of a value-returning function. Here is an example:

```
double x;
x = pow(4.0, 2.0);
```

This code calls the `pow` function, passing 4.0 and 2.0 as arguments. The function calculates the value of 4.0 raised to the power of 2.0 and returns that value. The value, which is 16.0, is assigned to the `x` variable by the = operator.

Although several arguments can be passed into a function, only one value can be returned from it. Think of a function as having multiple communication channels for receiving data (parameters), but only one channel for sending data (the return value). This is illustrated in Figure 6-9.

Figure 6-9





NOTE: In order to return multiple values from a function, they must be "packaged" in such a way that they are treated as a single value. You will learn to do this in Chapter 7.

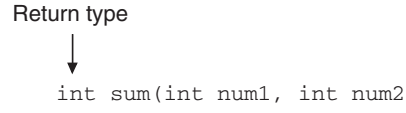
Defining a Value-Returning Function

When you are writing a value-returning function, you must decide what type of value the function will return. This is because you must specify the data type of the return value in the function header and function prototype. Up until now all the functions we have written have been `void` functions. This means they do not return a value. These functions use the key word `void` as the return type in their function header and function prototype. A value-returning function, on the other hand, uses `int`, `double`, `bool`, or any other valid data type in its header. Here is an example of a function that returns an `int` value:

```
int sum(int num1, int num2)
{
   int result;

   result = num1 + num2;
   return result;
}
```

The name of this function is `sum`. Notice in the function header that the return type is `int`, as illustrated in Figure 6-10.

Figure 6-10



This code defines a function named `sum` that accepts two `int` arguments. The arguments are passed into the parameter variables `num1` and `num2`. Inside the function, the variable `result` is defined. Variables that are defined inside a function are called *local variables*. After the variable definition, the values of the parameter variables `num1` and `num2` are added, and their sum is assigned to the `result` variable. The last statement in the function is:

```
return result;
```

This statement causes the function to end, and it sends the value of the `result` variable back to the statement that called the function. A value-returning function must have a `return` statement written in the following general format:

```
return expression;
```

In the general format, *`expression`* is the value to be returned. It can be any expression that has a value, such as a variable, literal, or mathematical expression. The value of the expression is converted to the data type that the function returns, and is sent back to the statement that called the function. In this case, the `sum` function returns the value in the `result` variable.

However, we could have eliminated the `result` variable entirely and returned the expression `num1 + num2`, as shown in the following code:

```
int sum(int num1, int num2)
{
   return num1 + num2;
}
```

The prototype for a value-returning function follows the same conventions that we covered earlier. Here is the prototype for the `sum` function:

```
int sum(int num1, int num2);
```

Calling a Value-Returning Function

Program 6-11 shows an example of how to call the `sum` function.

Program 6-11

```
1  // This program uses a function that returns a value.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype
6  int sum(int num1, int num2);
7
8  int main()
9  {
10    int value1 = 20,    // The first value
11        value2 = 40,    // The second value
12        total;          // Holds the returned total
13
14    // Call the sum function, passing the contents of
15    // value1 and value2 as arguments. Assign the return
16    // value to the total variable.
17    total = sum(value1, value2);
18
19    // Display the sum of the values.
20    cout << "The sum of " << value1 << " and "
21         << value2 << " is " << total << endl;
22    return 0;
23 }
24
25 /**********************************************************
26  *                          sum                           *
27  *  This function returns the sum of its two parameters. *
28  **********************************************************/
29 int sum(int num1, int num2)
30 {
31    return num1 + num2;
32 }
```

Program Output

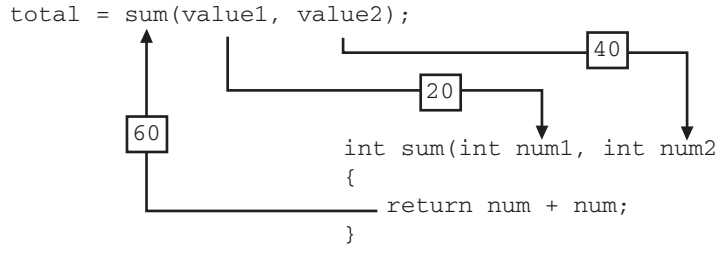
```
The sum of 20 and 40 is 60
```

Here is the statement in line 17, which calls the `sum` function, passing `value1` and `value2` as arguments.

```
total = sum(value1, value2);
```

This statement assigns the value returned by the `sum` function to the `total` variable. In this case, the function will return 60. Figure 6-11 shows how the arguments are passed into the function and how a value is passed back from the function.

Figure 6-11



When you call a value-returning function, you usually want to do something meaningful with the value it returns. Program 6-11 shows a function's return value being assigned to a variable. This is commonly how return values are used, but you can do many other things with them as well. For example, the following code shows a math expression that uses a call to the `sum` function:

```
int x = 10, y = 15;
double average;
average = sum(x, y) / 2.0;
```

In the last statement, the `sum` function is called with `x` and `y` as its arguments. The function's return value, which is 25, is divided by 2.0. The result, 12.5, is assigned to `average`. Here is another example:

```
int x = 10, y = 15;
cout << "The sum is " << sum(x, y) << endl;
```

This code sends the `sum` function's return value to `cout` so it can be displayed on the screen. The message "The sum is 25" will be displayed.

Remember, a value-returning function returns a value of a specific data type. You can use the function's return value anywhere that you can use a regular value of the same data type. This means that anywhere an `int` value can be used, a call to an `int` value-returning function can be used. Likewise, anywhere a `double` value can be used, a call to a `double` value-returning function can be used. The same is true for all other data types.

Let's look at another example. Program 6-12, which calculates the area of a circle, has two functions in addition to `main`. One of the functions is named `square`, and it returns the square of any number passed to it as an argument. The `square` function is called in a mathematical statement. The program also has a function named `getRadius`, which prompts the user to enter the circle's radius. The value entered by the user is returned from the function.

Program 6-12

```
// This program demonstrates two value-returning functions.
// The square function is called in a mathematical statement.
#include <iostream>
#include <iomanip>
using namespace std;

//Function prototypes
double getRadius();
double square(double number);

int main()
{
   const double PI = 3.14159; // Constant for pi
   double radius;             // Holds the circle's radius
   double area;               // Holds the circle's area

   // Set the numeric output formatting
   cout << fixed << showpoint << setprecision(2);

   // Get the radius of the circle
   cout << "This program calculates the area of a circle.\n";
   radius = getRadius();

   // Caclulate the area of the circle
   area = PI * square(radius);

   // Display the area
   cout << "The area is " << area << endl;
   return 0;
}

/****************************************
 *               getRadius              *
 * This function returns the circle radius  *
 * input by the user.                   *
 ****************************************/
double getRadius()
{
   double rad;

   cout << "Enter the radius of the circle: ";
   cin  >> rad;
   return rad;
}

/****************************************
 *                  square              *
 * This function returns the square of the   *
 * double argument sent to it           *
 ****************************************/
```

(program continues)

Program 6-12 *(continued)*

```
51 double square(double number)
52 {
53    return number * number;
54 }
```

Program Output with Example Input Shown in Bold

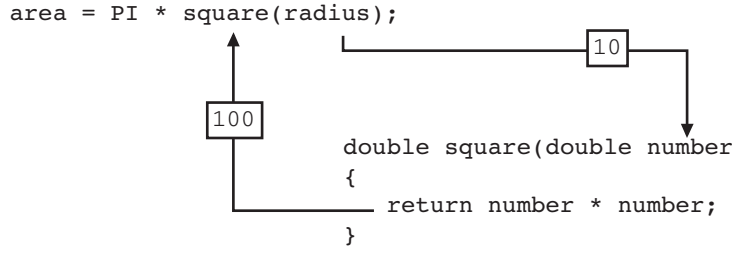
```
This program calculates the area of a circle.
Enter the radius of the circle: 10[Enter]
The area is 314.16
```

First, look at the `getRadius` function which is defined in lines 37 through 44. Notice that there is nothing inside the parentheses of the function header on line 37. This means the function has no parameters, so no arguments are sent to it when it is called. The purpose of this function is to get the circle radius from the user. In line 39 the function defines a local variable, `rad`. Line 41 displays a prompt and line 42 accepts the user's input for the circle's radius, which is stored in the `rad` variable. In line 43 the value of the `rad` variable is returned. The `getRadius` function is called in line 22 of the `main` function. When the value is returned from the `getRadius` function it is assigned to the `radius` variable.

Next look at the `square` function, which is defined in lines 51 through 54. When the function is called, a `double` argument is passed to it. The function stores the argument in its `number` parameter. The `return` statement in line 53 returns the value of the expression `number * number`, which is the square of the value in the `number` parameter. The `square` function is called in line 25 of the `main` function, with the value of `radius` passed as an argument. The `square` function will return the square of the `radius` variable, and that value will be used in the mathematical expression that computes the circle's area.

Assuming the user has entered 10 as the radius, and this value is passed as an argument to the `square` function, the `square` function will return the value 100. Figure 6-12 illustrates how the value 100 is passed back to be used in the mathematical expression.

Figure 6-12



Functions can return values of any type. Both the `getRadius` and `square` functions in Program 6-12 return a `double`. The `sum` function you saw in Program 6-11 returned an `int`. When a statement calls a value-returning function, it should properly handle the return value. For example, if you assign the return value of the `square` function to a variable, the variable should be a `double`. If the return value of the function has a fractional portion and you assign it to an `int` variable, the value will be truncated.

6.8 Returning a Boolean Value

CONCEPT: Functions may return `true` or `false` values.

Frequently there is a need for a function that tests an argument and returns a `true` or `false` value indicating whether or not a condition is satisfied. Such a function would return a `bool` value. For example, the `isValid` function shown below accepts an `int` argument and returns `true` if the argument is within the range of 1 through 100, or `false` otherwise.

```
bool isValid(int number)
{
   bool status;

   if (number >= 1 && number <= 100)
      status = true;
   else
      status = false;
   return status;
}
```

The following code shows an `if/else` statement that makes a call to the function:

```
int value = 20;
if (isValid(value))
   cout << "The value is within range.\n";
else
   cout << "The value is out of range.\n";
```

Because `value` equals 20, this code will display the message "The value is within range." when it executes.

Program 6-13 shows another example of a function whose return type is `bool`. This program has a function named `isEven` which returns `true` if its argument is an even number. Otherwise, the function returns `false`.

Program 6-13

```
1  // This program uses a function that returns true or false.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype
6  bool isEven(int);
7
8  int main()
9  {
10    int val;        // Holds the value to be tested
11
12    // Get a number from the user
13    cout << "Enter an integer and I will tell you ";
14    cout << "if it is even or odd: ";
15    cin  >> val;
16
```

(program continues)

Program 6-13 *(continued)*

```
17    // Indicate whether it is even or odd
18    if (isEven(val))
19       cout << val << " is even.\n";
20    else
21       cout << val << " is odd.\n";
22    return 0;
23 }
24
25 /*****************************************************
26  *                      isEven                       *
27  * This Boolean function tests if the integer argument *
28  * it receives is even or odd. It returns true if the  *
29  * argument is even and false if it is odd.            *
30  *****************************************************/
31 bool isEven(int number)
32 {
33    if (number % 2 == 0)
34        return true;  // The number is even if there's no remainder
35    else
36        return false; // Otherwise, the number is odd
37 }
```

Program Output with Example Input Shown in Bold

```
Enter an integer and I will tell you if it is even or odd: 5[Enter]
5 is odd.
```

Notice how the `isEven` function is called in line 18 with the following statement:

```
if (isEven(val))
```

Recall from Chapter 4 that this is asking if the function call `isEven(val)` returned the value `true`. When the `if` statement executes, `isEven` is called with `val` as its argument. If `val` is even, `isEven` returns `true`, otherwise it returns `false`.

Notice also how the `isEven` function that begins on line 31 uses an `if` statement to return either the value `true` or the value `false`. There are several other ways this function could have been written. Let's compare three different ways to write it.

```
// Program 6-13 code
bool isEven(int number)
{
   if (number % 2 == 0)
      return true;
   else
      return false;
}
```

```
// Version 2
bool isEven(int number)
{  bool answer;

   if (number % 2 == 0)
      answer = true;
   else
      answer = false;

   return answer;
}
```

```
// Version 3
bool isEven(int number)
{  bool answer = false;

   if (number % 2 == 0)
      answer = true;

   return answer;
}
```

Although the code used in Program 6-13 is short and clear, it has two different `return` statements. Many instructors prefer that a value-returning function have only a single `return` statement, placed at the end of the function. Versions 2 and 3 do this. Your instructor will let you know which method you should use.



Checkpoint

6.11 How many return values may a function have?

6.12 Write a header for a function named `distance`. The function should return a `double` and have two `double` parameters: `rate` and `time`.

6.13 Write a header for a function named `days`. The function should return an `int` and have three `int` parameters: `years`, `months`, and `weeks`.

6.14 Write a header for a function named `getKey`. The function should return a `char` and use no parameters.

6.15 Write a header for a function named `lightYears`. The function should return a `long` and have one `long` parameter: `miles`.

6.9 Using Functions in a Menu-Driven Program

CONCEPT: Functions are ideal for use in menu-driven programs. When the user selects an item from a menu, the program can call the appropriate function.

In Chapters 4 and 5 you saw a menu-driven program that calculates the charges for a health club membership. Program 6-14 is an improved modular version of that program.

Program 6-14

```
1  // This is a modular, menu-driven program that computes
2  // health club membership fees.
3  #include <iostream>
4  #include <iomanip>
5  #include <string>
6  using namespace std;
7
8  // Function prototypes
9  void displayMenu();
10 int getChoice();
11 void showFees(string category, double rate, int months);
12
13 int main()
14 {
15    // Constants for monthly membership rates
16    const double ADULT_RATE =  40.00,
17                 SENIOR_RATE = 30.00,
18                 CHILD_RATE =  20.00;
19    int choice,                      // Holds the user's menu choice
20        months;                      // Number of months being paid
21
```

(program continues)

Program 6-14 *(continued)*

```
   // Set numeric output formatting
   cout << fixed << showpoint << setprecision(2);

    do
    {  displayMenu();
       choice = getChoice(); // Assign choice the value returned
                             // by the getChoice function
       if (choice != 4)      // If user does not want to quit, proceed
       {
          cout << "For how many months? ";
          cin  >> months;

          switch (choice)
          {
             case 1:  showFees("Adult", ADULT_RATE, months);
                      break;
             case 2:  showFees("Child", CHILD_RATE, months);
                      break;
             case 3:  showFees("Senior", SENIOR_RATE, months);
          }
       }
   } while (choice != 4);
   return 0;
}

/*******************************************
 *                displayMenu              *
 * This function clears the screen and then *
 * displays the menu choices.               *
 *******************************************/
void displayMenu()
{
   system("cls");          // Clear the screen.
   cout << "\n  Health Club Membership Menu\n\n";
   cout << "1.  Standard Adult Membership\n";
   cout << "2.  Child Membership\n";
   cout << "3.  Senior Citizen Membership\n";
   cout << "4.  Quit the Program\n\n";
}

/*************************************************
 *                   getChoice                   *
 *  This function inputs, validates, and returns *
 *  the user's menu choice.                      *
 *************************************************/
int getChoice()
{
   int choice;

   cin >> choice;
```

(program continues)

Program 6-14 *(continued)*

```
72    while (choice < 1 || choice > 4)
73    {  cout << "The only valid choices are 1-4.  Please re-enter. ";
74       cin  >> choice;
75    }
76    return choice;
77 }
78
79 /***********************************************************
80  *                          showFees                       *
81  * This function uses the membership type, monthly rate, and  *
82  * number of months passed to it as arguments to compute and  *
83  * display a member's total charges. It then holds the screen *
84  * until the user presses the ENTER key. This is necessary    *
85  * because after returning from this function the displayMenu *
86  * function will be called, and it will clear the screen.     *
87  ***********************************************************/
88 void showFees(string memberType, double rate, int months)
89 {
90    cout << endl
91         << "Membership Type : "  << memberType << "    "
92         << "Number of months: "  << months << endl
93         << "Total charges   : $" << (rate * months) << endl;
94
95    // Hold the screen until the user presses the ENTER key.
96     cout << "\nPress the Enter key to return to the menu. ";
97    cin.get();      // Clear the previous \n out of the input buffer
98    cin.get();      // Wait for the user to press ENTER
99 }
```

Program Output with Example Input Shown in Bold

```
      Health Club Membership Menu
1.  Standard Adult Membership
2.  Child Membership
3.  Senior Citizen Membership
4.  Quit the Program

1[Enter]
For how many months? 3[Enter]

Membership Type : Adult    Number of months: 3
Total charges   : $120

Press the Enter key to return to the menu.
```

Notice how each function, or module, of Program 6-14 is designed to perform a specific task. The `displayMenu` function, as its name suggests, displays the menu of choices. The `getChoice` function gets the user's menu choice and validates it before returning it to the `main` function. The `showFees` function computes and displays membership information and fees. Notice, in particular, the versatility of this function, which is called in three different places within the `switch` statement. It is passed three arguments: a `string` holding the membership type, a `double` holding the monthly fee for that membership type, and an

`int` holding the number of months being billed. Without these arguments, we would need a whole set of functions: one to compute adult membership fees, another to compute child membership fees, and a third to compute senior membership fees. Because we can vary the information passed as arguments to the function, however, we are able to create a single general-purpose function that works for all three cases.

Clearing the Screen

Sometimes in a program you want to clear the screen and place the cursor back up at the top. This is particularly useful when you are writing a menu-driven program. After the user has made a menu selection and the function to carry out that choice has been executed, it would be nice to be able to clear the screen before redisplaying the menu. This can be accomplished by inserting a command in your program that asks the operating system to clear the screen for you. Here is the command for Unix-based operating systems, such as Linux and Mac OS:

```
system("clear");
```

And here is the command for Windows operating systems. You may have noticed that it appears in line 54 of Program 6-14, just before the menu is displayed.

```
system("cls");
```

This removes the previous report from the screen before the user selects a new one to be displayed. However, it is important not to clear the screen too quickly after a report displays, or it will disappear before the user has a chance to look at it. Take a look at lines 95 through 98 of Program 6-14. These lines hold the report screen until the user presses the [Enter] key to signal readiness to return to the menu and begin something new.

6.10 Local and Global Variables

CONCEPT: **A local variable is defined inside a function and is not accessible outside the function. A global variable is defined outside all functions and is accessible to all functions in its scope.**

Local Variables

Variables defined inside a function are *local* to that function. They are hidden from the statements in other functions, which normally cannot access them. Program 6-15 shows that because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.

Program 6-15

```
1 // This program shows that variables defined in a function
2 // are hidden from other functions.
3 #include <iostream>
4 using namespace std;}
5
```

(program continues)

Program 6-15 *(continued)*

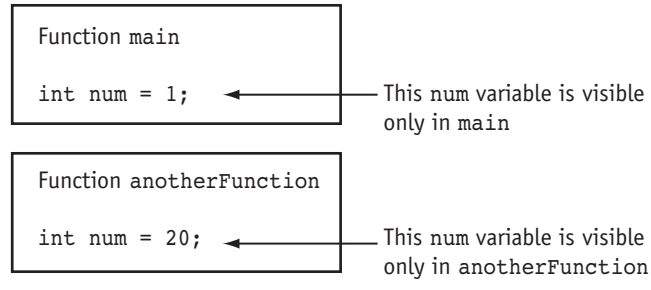
```
6  void anotherFunction();     // Function prototype
7
8  int main()
9  {
10    int num = 1;             // Local variable
11
12    cout << "In main, num is " << num << endl;
13    anotherFunction();
14    cout << "Back in main, num is still " << num << endl;
15    return 0;
16 }
17
18 /*************************************************************
19  *                      anotherFunction                      *
20  * This function displays the value of its local variable num. *
21  *************************************************************/
22 void anotherFunction()
23 {
24    int num = 20;            // Local variable
25
26    cout << "In anotherFunction, num is " << num << endl;
27 }
```

Program Output

```
In main, num is 1
In anotherFunction, num is 20
Back in main, num is still 1
```

Even though there are two variables named `num`, the program can only "see" one of them at a time because they are in different functions. When the program is executing in `main`, the `num` variable defined in `main` is visible. When `anotherFunction` is called, however, only variables defined inside it are visible, so the `num` variable in `main` is hidden. Figure 6-13 illustrates the closed nature of the two functions. The boxes represent the scope of the variables.

Figure 6-13



Local Variable Lifetime

A local variable exists only while the function it is defined in is executing. This is known as the *lifetime* of a local variable. When the function begins, its parameter variables and any local variables it defines are created in memory, and when the function ends, they are destroyed. This means that any values stored in a function's parameters or local variables are lost between calls to the function.

Initializing Local Variables with Parameter Values

It is possible to use parameter variables to initialize local variables. Sometimes this simplifies the code in a function. Here is a modified version of the `sum` function we looked at earlier. In this version, the function's parameters are used to initialize the local variable `result`.

```
int sum(int num1, int num2)
{
   int result = num1 + num2;
   return result;
}
```

Global Variables

A global variable is any variable defined outside all the functions in a program, including `main`. The scope of a global variable is the portion of the program from the variable definition to the end of the entire program. This means that a global variable can be accessed by all functions that are defined after the global variable is defined. Program 6-16 shows two functions, `main` and `anotherFunction`, which access the same global variable, `num`.

Program 6-16

```
1  // This program shows that a global variable is visible to all functions
2  // that appear in a program after the variable's definition.
3  #include <iostream>
4  using namespace std;
5
6  void anotherFunction(); // Function prototype
7  int num = 2;            // Global variable
8
9  int main()
10 {
11    cout << "In main, num is " << num << endl;
12    anotherFunction();
13    cout << "Back in main, num is " << num << endl;
14    return 0;
15 }
16
17 /*************************************************************
18  *                      anotherFunction                      *
19  * This function changes the value of the global variable num. *
20  *************************************************************/
```

(program continues)

Program 6-16 *(continued)*

```
21 void anotherFunction()
22 {
23    cout << "In anotherFunction, num is " << num << endl;
24    num = 50;
25    cout << "But, it is now changed to " << num << endl;
26 }
```

Program Output

```
In main, num is 2
In anotherFunction, num is 2
But, it is now changed to 50
Back in main, num is 50
```

In Program 6-16, `num` is defined outside of all the functions. Because its definition appears before the definitions of `main` and `anotherFunction`, both functions have access to it.

In C++, unless you explicitly initialize numeric global variables, they are automatically initialized to zero. Global character variables are initialized to NULL.* In Program 6-17 the variable `globalNum` is never set to any value by a statement, but because it is global it is automatically set to zero.

Program 6-17

```
1  // This program has an uninitialized global variable.
2  #include <iostream>
3  using namespace std;
4
5  int globalNum;        // Global variable automatically set to zero
6
7  int main()
8  {
9     cout << "globalNum is " << globalNum << endl;
10    return 0;
11 }
```

Program Output

```
globalNum is 0
```



NOTE: Remember that local variables are not automatically initialized as global variables are. The programmer must handle this.

Global Constants

Although global variables can be useful, you should restrict your use of them. When beginning students first learn to write programs with multiple functions, they are sometimes tempted to make all their variables global. This is usually because global

* The NULL character is stored as ASCII 0.

variables can be accessed by any function in the program without being passed as arguments. Although this approach might make a program easier to create, it usually causes problems later.

The reasons are as follows:

- *Global variables make debugging difficult.* Any statement in a program can change the value of a global variable. If you find that the wrong value is being stored in a global variable, you have to track down every statement that accesses it to determine where the bad value is coming from. In a program with thousands of lines of code, this can be difficult.
- *Functions that use global variables are usually dependent on those variables.* If you want to use such a function in a different program, most likely you will have to redesign it so it does not rely on the global variable.
- *Global variables make a program hard to understand.* A global variable can be modified by any statement in the program. So to understand any part of the program that uses a global variable, you have to be aware of all the other parts of the program that access it.

In most cases, you should declare variables locally and pass them as arguments to the functions that need to access them. Certainly this is the case with variables used to store, manipulate, and retrieve data. If you wish to share global information among modules, you should declare a global constant. A *global constant*, defined with the key word `const`, is a read only variable that is visible to all modules, but that cannot be changed by any of them. Therefore modules do not have to worry about who else may be using them.

Program 6-18 shows an example of how global constants might be used. The program calculates gross pay, including overtime, for a company's management trainees. All trainees earn the same amount per hour. In addition to `main`, this program has two functions: `getBasePay` and `getOvertimePay`. The `getBasePay` function accepts the number of hours worked and returns the amount of pay for the non-overtime hours. The `getOvertimePay` function accepts the number of hours worked and returns the amount of pay for the overtime hours, if any.

Program 6-18

```
1  // This program calculates gross pay. It uses global constants.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  // Global constants
7  const double PAY_RATE = 22.55;     // Hourly pay rate
8  const double BASE_HOURS = 40.0;    // Max non-overtime hours
9  const double OT_MULTIPLIER = 1.5;  // Overtime multiplier
10
11 // Function prototypes
12 double getBasePay(double);
13 double getOvertimePay(double);
14
```

(program continues)

Program 6-18 *(continued)*

```
15 int main()
16 {
17    double hours,               // Hours worked
18           basePay,             // Base pay
19           overtimePay = 0.0,   // Overtime pay
20           totalPay;            // Total pay
21
22    // Get the number of hours worked
23    cout << "How many hours did you work? ";
24    cin  >> hours;
25
26    // Get the amount of base pay
27    basePay = getBasePay(hours);
28
29    // Get overtime pay, if any
30    if (hours > BASE_HOURS)
31        overtimePay = getOvertimePay(hours);
32
33    // Calculate the total pay
34    totalPay = basePay + overtimePay;
35
36    // Display the pay
37    cout << setprecision(2) << fixed << showpoint;
38    cout << "Base pay     $" << setw(7) << basePay     << endl;
39    cout << "Overtime pay $" << setw(7) << overtimePay << endl;
40    cout << "Total pay    $" << setw(7) << totalPay    << endl;
41    return 0;
42 }
43
44 /*************************************************************
45  *                          getBasePay                       *
46  * This function uses the hours worked value passed in to    *
47  * compute and return an employee's pay for non-overtime hours.*
48  *************************************************************/
49 double getBasePay(double hoursWorked)
50 {
51    double basePay;
52
53    if (hoursWorked > BASE_HOURS)
54       basePay = BASE_HOURS * PAY_RATE;
55    else
56       basePay = hoursWorked * PAY_RATE;
57
58    return basePay;
59 }
60
61 /*******************************************************
62  *                      getOvertimePay                 *
63  * This function uses the hours worked value passed in  *
64  * to compute and return an employee's overtime pay.    *
65  *******************************************************/
```

(program continues)

Program 6-18 *(continued)*

```
66 double getOvertimePay(double hoursWorked)
67 {
68    double overtimePay;
69
70    if (hoursWorked > BASE_HOURS)
71    {
72       overtimePay =
73          (hoursWorked - BASE_HOURS) * PAY_RATE * OT_MULTIPLIER;
74    }
75    else
76       overtimePay = 0.0;
77
78    return overtimePay;
79 }
```

Program Output with Example Input Shown in Bold

```
How many hours did you work? 48[Enter]
Base pay     $ 902.00
Overtime pay $ 270.60
Total pay    $1172.60
```

Let's take a closer look at the program. Three global constants are defined in lines 7, 8, and 9. The `PAY_RATE` constant is set to the employee's hourly pay rate, which is 22.55. The `BASE_HOURS` constant is set to 40.0, which is the number of hours an employee can work in a week without getting paid overtime. The `OT_MULTIPLIER` constant is set to 1.5, which is the pay rate multiplier for overtime hours. This means that the employee's hourly pay rate is multiplied by 1.5 for all overtime hours.

Because these constants are global, and are defined before all of the functions in the program, all the functions may access them. For example, the `getBasePay` function accesses the `BASE_HOURS` constant in lines 53 and 54, and accesses the `PAY_RATE` constant in lines 54 and 56. The `getOvertimePay` function accesses the `BASE_HOURS` constant in line 70 and all three constants in line 73.

Local and Global Variables with the Same Name

You cannot have two local variables with the same name in the same function. This applies to parameter variables as well. A parameter variable is, in essence, a local variable. So, you cannot give a parameter variable and a local variable in the same function the same name.

However, you can have a parameter or local variable with the same name as a global variable or constant. When you do this, the name of the parameter or local variable *shadows* the name of the global variable or constant. This means that the global variable or constant's name is hidden by the name of the parameter or local variable. So, the global variable or constant can't be seen or used in this part of the program. Program 6-19 illustrates this. It has a global constant named `BIRDS` set to 500 and a local constant in the `california` function named `BIRDS` set to 10000.

Program 6-19

```
1  // This program demonstrates how a local variable or constant
2  // can shadow the name of a global variable or constant.
3  #include <iostream>
4  using namespace std;
5
6  void california();       // Function prototype
7
8  const int BIRDS = 500;   // Global constant
9
10 int main()
11 {
12    cout << "In main there are " << BIRDS << " birds.\n";
13    california();
14    return 0;
15 }
16
17 /*******************************
18  *          california         *
19  *******************************/
20 void california()
21 {
22    const int BIRDS = 10000;
23
24    cout << "In california there are " << BIRDS << " birds.\n";
25 }
```

Program Output

```
In main there are 500 birds.
In california there are 10000 birds.
```

When the program is executing in the `main` function, the global constant `BIRDS`, which is set to 500, is visible. The `cout` statement in line 12 displays "In main there are 500 birds." (My apologies to folks living in Maine for the difference in spelling.) When the program is executing in the `california` function, however, the local constant `BIRDS` shadows the global constant `BIRDS`. When the `california` function accesses `BIRDS`, it accesses the local constant. That is why the `cout` statement in line 24 displays "In california there are 10000 birds."

6.11 Static Local Variables

If a function is called more than once in a program, the values stored in the function's local variables do not persist between function calls. This is because local variables are destroyed when a function terminates and are then re-created when the function starts again. This is shown in Program 6-20.

Program 6-20

```
1  // This program shows that local variables do not retain
2  // their values between function calls.
3  #include <iostream>
4  using namespace std;
5
6  void showLocal();     // Function prototype
7
8  int main()
9  {
10    showLocal();
11    showLocal();
12    return 0;
13 }
14
15 /*****************************************************
16  *                       showLocal                   *
17  * This function sets, displays, and then changes the *
18  * value of local variable localNum before returning. *
19  *****************************************************/
20 void showLocal()
21 {
22    int localNum = 5;   // Local variable
23
24    cout << "localNum is " << localNum << endl;
25    localNum = 99;
26 }
```

Program Output

```
localNum is 5
localNum is 5
```

Even though in line 25 the last statement in the `showLocal` function stores 99 in `localNum`, the variable is destroyed when the function terminates. The next time the function is called, `localNum` is re-created and initialized to 5 all over again.

Sometimes, however, it's desirable for a program to "remember" what value is stored in a local variable between function calls. This can be accomplished by making the variable `static`. Static local variables are not destroyed when a function returns. They exist for the entire lifetime of the program, even though their scope is only the function in which they are defined. Program 6-21 uses a static local variable to count how many times a function is called.

Program 6-21

```
1  // This program uses a static local variable.
2  #include <iostream>
3  using namespace std;
4
5  void showStatic();    // Function prototype
6
```

(program continues)

Program 6-21 *(continued)*

```
 7 int main()
 8 {
 9    // Call the showStatic function five times
10    for (int count = 0; count < 5; count++)
11       showStatic();
12    return 0;
13 }
14
15 /**********************************************
16  *                  showStatic                *
17  * This function keeps track of how many times it *
18  * has been called by incrementing a static local *
19  * variable, numCalls, each time it is called.    *
20  **********************************************/
21 void showStatic()
22 {
23    static int numCalls = 0;   // Static local variable
24
25    cout << "This function has been called "
26         << ++numCalls << " times. " << endl;
27 }
```

Program Output

```
This function has been called 1 times.
This function has been called 2 times.
This function has been called 3 times.
This function has been called 4 times.
This function has been called 5 times.
```

In Program 6-21 `numCalls` is defined and initialized to 0 in line 23. It is incremented in line 26 once each time the `showStatic` function is called and, because it is a static variable, it retains its value between calls. You might think that every time the function is called, `numCalls` would be reinitialized to 0. But this does not happen because a variable is only initialized when it is first created, and static variables are only created once during the running of the program. If we had not initialized `numCalls`, it would automatically have been initialized to 0 because numeric static local variables, like global variables, are initialized to 0 if the programmer does not initialize them.



Checkpoint

6.16 What is the difference between a static local variable and a global variable?

6.17 What is the output of the following program?

```
#include <iostream>
using namespace std;

void myFunc();  // Function prototype
```

```
int main()
{  int var = 100;

   cout << var << endl;
   myFunc();
   cout << var << endl;
   return 0;
}
// Definition of function myFunc
void myFunc()
{  int var = 50;

   cout << var << endl;
}
```

6.18 What is the output of the following program?

```
#include <iostream>
using namespace std;

void showVar(); // Function prototype

int main()
{  for (int count = 0; count < 10; count++)
      showVar();
   return 0;
}
// Definition of function showVar
void showVar()
{  static int var = 10;

   cout << var << endl;
   var++;
}
```

6.12 Default Arguments

CONCEPT: Default arguments are passed to parameters automatically if no argument is provided in the function call.

It's possible to assign *default arguments* to function parameters. A default argument is passed to the parameter when the actual argument is left out of the function call. The default arguments are usually listed in the function prototype. Here is an example:

```
void showArea(double length = 20.0, double width = 10.0);
```

Because parameter names are not required in function prototypes, the example prototype could also be declared like this:

```
void showArea(double = 20.0, double = 10.0);
```

In either case, the default arguments, which must be literal values or constants, have an = operator in front of them.

Notice that in both example prototypes, the function `showArea` has two `double` parameters. The first is assigned the default argument 20.0 and the second is assigned the default argument 10.0. Here is the definition of the function:

```
void showArea(double length, double width)
{
   double area = length * width;
   cout << "The area is " << area << endl;
}
```

The default argument for `length` is 20.0 and the default argument for `width` is 10.0. Because both parameters have default arguments, they may optionally be omitted in the function call, as shown here:

```
showArea();
```

In this function call, both default arguments will be passed to the parameters. Parameter `length` will receive the value 20.0 and `width` will receive the value 10.0. The output of the function will be

```
The area is 200
```

The default arguments are only used when the actual arguments are omitted from the function call. In the following call, the first argument is specified, but the second is omitted:

```
showArea(12.0);
```

The value 12.0 will be passed to `length`, while the default value 10.0 will be passed to `width`. The output of the function will be

```
The area is 120
```

Of course, all the default arguments may be overridden. In the following function call, arguments are supplied for both parameters:

```
showArea(12.0, 5.5);
```

The output of this function call will be

```
The area is 66
```



NOTE: A function's default arguments should be assigned in the earliest occurrence of the function name. This will usually be the function prototype. However, if a function does not have a prototype, default arguments may be specified in the function header. The `showArea` function could be defined as follows:

```
void showArea(double length = 20.0, double width = 10.0)
{
     double area = length * width;
     cout << "The area is " << area << endl;
}
```

Program 6-22 illustrates the use of default function arguments. It has a function that displays asterisks on the screen. This function receives arguments specifying how many rows of asterisks to display and how many asterisks to print on each row. Default arguments are provided to display 1 row of 10 asterisks.

Program 6-22

```
1  // This program demonstrates the use of default function arguments.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype with default arguments
6  void displayStars(int starsPerRow = 10, int numRows = 1);
7
8  int main()
9  {
10    displayStars();     // starsPerRow & numRows use defaults (10 & 1)
11    cout << endl;
12    displayStars(5);    // starsPerRow 5. numRows uses default value 1
13    cout << endl;
14    displayStars(7, 3); // starsPerRow 7. numRows 3. No defaults used.
15    return 0;
16 }
17
18 /**********************************************************
19  *                      displayStars                      *
20  *  This function displays a rectangle made of asterisks.  *
21  *  If arguments are not passed to it, it uses the default *
22  *  arguments 10 for starsPerRow and 1 for numRows.        *
23  **********************************************************/
24 void displayStars(int starsPerRow, int numRows)
25 {
26    // Nested loop. The outer loop controls the rows and
27    // the inner loop controls the number of stars per row.
28    for (int row = 1; row <= numRows; row++)
29    {
30       for (int star = 1; star <= starsPerRow; star++)
31          cout << '*';
32       cout << endl;
33    }
34 }
```

Program Output

```
**********

*****

*******
*******
*******
```

Although C++'s default arguments are very convenient, they are not totally flexible in their use. When an argument is left out of a function call, all arguments that come after it must be left out as well. In the `displayStars` function in Program 6-22, it is not possible to omit the argument for `starsPerRow` without also omitting the argument for `numRows`. For example, the following function call would be illegal:

```
displayStars(, 3);      // Illegal function call!
```

It is possible, however, for a function to have some parameters with default arguments and some without. For example, in the following function, only the last parameter has a default argument:

```
// Function prototype
void calcPay(int empNum, double payRate, double hours = 40.0);

// Definition of function calcPay
void calcPay(int empNum, double payRate, double hours)
{
   double wages;

   wages = payRate * hours;

   cout << "Gross pay for employee number ";
   cout << empNum << " is " << wages << endl;
}
```

When calling this function, arguments must always be specified for the first two parameters (`empNum` and `payRate`) because they have no default arguments. Here are examples of valid calls:

```
calcPay(769, 15.75);      // Uses default argument for hours
calcPay(142, 12.00, 20);  // Specifies number of hours
```

When a function uses a mixture of parameters with and without default arguments, the parameters with default arguments must be declared last. In the `calcPay` function, `hours` could not have been declared before either of the other parameters. The following prototypes are illegal:

```
// Illegal prototype
void calcPay(int empNum, double hours = 40.0, double payRate);

// Illegal prototype
void calcPay(double hours = 40.0, int empNum, double payRate);
```

Here is a summary of the important points about default arguments:

- The value of a default argument must be a literal value or a named constant.
- When an argument is left out of a function call (because it has a default value), all the arguments that come after it must also be left out.
- When a function has a mixture of parameters both with and without default arguments, the parameters with default arguments must be defined last.

6.13 Using Reference Variables as Parameters

CONCEPT: **A reference variable is a variable that references the memory location of another variable. Any change made to the reference variable is actually made to the one it references. Reference variables are sometimes used as function parameters.**

Earlier you saw that arguments are normally passed to a function by value. This means that parameters receive only a copy of the value sent to them, which they store in the function's local memory. Any changes made to the parameter's value do not affect the value of the original argument.

Sometimes, however, we want a function to be able to change a value in the calling function (i.e., the function that called it). This can be done by making the parameter a reference variable.

You learned in Chapter 1 that variables are the names of memory locations that may hold data. When we use a variable we are accessing data stored in the memory location assigned to it. A *reference variable* is an alias for another variable. Instead of having its own memory location for storing data, it accesses the memory location of another variable. Any change made to the reference variable's data is actually made to the data stored in the memory location of the other variable. When we use a reference variable as a parameter, it becomes an alias for the corresponding variable in the argument list. Any change made to the parameter is actually made to the variable in the calling function. When data is passed to a parameter in this manner, the argument is said to be *passed by reference*.

Reference variables are defined like regular variables, except there is an ampersand (`&`) between the data type and the name. For example, the following function definition makes the parameter `refVar` a reference variable:

```
void doubleNum(int &refVar)
{
   refVar *= 2;
}
```

You may place the space either before or after the ampersand. The `doubleNum` function heading could also have been written like this:

```
void doubleNum(int& refVar)
```



NOTE: The variable `refVar` is called "a reference to an `int`."

This function doubles `refVar` by multiplying it by 2. Because `refVar` is a reference variable, this action is actually performed on the variable that was passed to the function as an argument.

The prototype for a function with a reference parameter must have an ampersand as well. As in the function header, it goes between the data type and the variable name. If the variable name is omitted from the prototype, the ampersand simply follows the data type. All of the following prototypes for the `doubleNum` function are correct.

```
void doubleNum(int &refVar);
void doubleNum(int& refVar);
void doubleNum(int &);
void doubleNum(int&);
```

Your instructor will let you know which form to use.



NOTE: The ampersand must appear in both the prototype and the header of any function that uses a reference variable as a parameter. It does not appear in the function call.

Program 6-23 demonstrates the use of a parameter that is a reference variable.

Program 6-23

```
1 // This program uses a reference variable as a function parameter.
2 #include <iostream>
3 using namespace std;
4
```

(program continues)

Program 6-23 *(continued)*

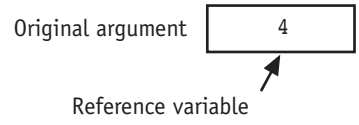
```
5  // Function prototype. The parameter is a reference variable.
6  void doubleNum(int &refVar);
7
8  int main()
9  {
10    int value = 4;
11
12    cout << "In main, value is " << value << endl;
13    cout << "Now calling doubleNum..." << endl;
14    doubleNum(value);
15    cout << "Now back in main, value is " << value << endl;
16    return 0;
17 }
18
19 /*************************************************************
20  *                        doubleNum                          *
21  * This function's parameter is a reference variable. The &   *
22  * tells us that. This means it receives a reference to the   *
23  * original variable passed to it, rather than a copy of that *
24  * variable's data. The statement refVar *= 2 is doubling the *
25  * data stored in the value variable defined in main.         *
26  *************************************************************/
27 void doubleNum (int &refVar)
28 {
29    refVar *= 2;
30 }
```

Program Output

```
In main, value is 4
Now calling doubleNum...
Now back in main, value is 8
```

The parameter `refVar` in Program 6-23 "points" to the `value` variable in function `main`. When a program works with a reference variable, it is actually working with the variable it references, or points to. This is illustrated in Figure 6-14.

Figure 6-14



Using reference variables as function parameters is especially useful when the purpose of the function is to accept input values to be stored in variables of the calling function. Another use of reference parameters is when multiple values must be sent back from the function. If the function is computing and sending back a single value, it is generally considered more appropriate to use a value-returning function and send the value back with a `return` statement.

Program 6-24 is a modification of Program 6-23. It adds a function `getNum`, which accepts an input from the user and stores it in `userNum`. However, the parameter `userNum` is a

reference to `main's` variable `value`, so that is where the input data is actually stored. Program 6-24 also rewrites the function `doubleNum` as a value-returning function. Notice, in line 19, how `main` must now store the value when `doubleNum` returns it.

Program 6-24

```
1  // This program uses 2 functions: a void function with a reference
2  // variable as a parameter, and a value-returning function.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototypes
7  void getNum(int &);
8  int doubleNum(int);
9
10 int main()
11 {
12    int value;
13
14    // Call getNum to get a number and store it in value
15    getNum(value);
16
17    // Call doubleNum, passing it the number stored in value
18    // Assign value the number returned by the function
19    value = doubleNum(value);
20
21    // Display the resulting number
22    cout << "That value doubled is " << value << endl;
23    return 0;
24 }
25
26 /*********************************************************
27  *                          getNum                       *
28  * This function stores user input data in main's value   *
29  * variable by using a reference variable as a parameter. *
30  *********************************************************/
31 void getNum(int &userNum)
32 {
33    cout << "Enter a number: ";
34    cin  >> userNum;
35 }
36
37 /*********************************************************
38  *                         doubleNum                     *
39  * This function doubles the number it receives as an     *
40  * argument and returns it to main thru a return statement.*
41  *********************************************************/
42 int doubleNum (int number)
43 {
44    return number * 2;
45 }
```

Program Output with Example Input Shown in Bold

```
Enter a number: 12[Enter]
That value doubled is 24
```



NOTE: Only variables may be passed by reference. If you attempt to pass a non-variable argument, such as a literal, a constant, or an expression, into a reference parameter, an error will result.

If a function has more than one parameter that is a reference variable, you must use an ampersand for each of them in both the prototype and the function header. Here is the prototype for a function that uses four reference variable parameters:

```
void addThree(int& num1, int& num2, int& num3, int& sum);
```

and here is the function definition:

```
void addThree(int& num1, int& num2, int& num3, int& sum)
{
   cout << "Enter three integer values: ";
   cin  >> num1 >> num2 >> num3;
   sum = num1 + num2 + num3;
}
```



WARNING! Don't get carried away with using reference variables as function parameters. Any time you allow a function to alter a variable that's outside the function, you are creating potential debugging problems. Reference variables should only be used as parameters when the purpose of a function is to accept input data to be stored in the calling function's variables or when it is the programmer's intent that a function alter a value passed to it.

When to Pass Arguments by Reference and When to Pass Arguments by Value

New programmers often have a problem determining when an argument should be passed to a function by reference and when it should be passed by value. The problem is further compounded by the fact that if a value must be "sent back" to the calling function there are two ways to do it: by using a reference parameter or by using a `return` statement. Here are some general guidelines.

- When an argument is a constant, it must be passed by value. Only variables can be passed by reference.
- When a variable passed as an argument should not have its value changed, it should be passed by value. This protects it from being altered.
- When exactly one value needs to be "sent back" from a function to the calling routine, it should generally be returned with a `return` statement rather than through a reference parameter.
- When two or more variables passed as arguments to a function need to have their values changed by that function, they should be passed by reference.
- When a copy of an argument cannot reasonably or correctly be made, such as when the argument is a file stream object, it must be passed by reference.

Here are three common instances when reference parameters are used.

- When data values being input in a function need to be known by the calling function.
- When a function must change existing values in the calling function.
- When a file stream object is passed to a function.

Program 6-25 illustrates the first two of these uses. The `getNums` function uses reference variables as parameters so that it can store the values it inputs into the `main` function's `small` and `big` variables. The `orderNums` function uses reference variables as parameters so that when it swaps the two items passed to it, the values will actually be swapped in the `main` function.

Program 6-25

```
1  // This program illustrates two appropriate uses of passing arguments
2  // by reference.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototypes
7  void getNums (int&, int&);   // Uses reference parameters to input
8                               // values in the function, but to actually
9                               // store them in variables defined in main
10
11 void orderNums(int&, int&);  // Uses reference parameters to change the
12                              // values of existing values stored in main
13
14 int main()
15 {
16    int small, big;
17
18    // Call getNums to input the two numbers
19    getNums(small, big);
20
21    // Call orderNums to put the numbers in order
22    orderNums(small, big);
23
24    // Display the new values
25    cout  << "The two input numbers ordered smallest to biggest are "
26          << small << " and " << big << endl;
27    return 0;
28 }
29
30 /*********************************************************
31  *                        getNums                        *
32  * The arguments passed into input1 and input2 are passed *
33  * by reference so that the values entered into them will *
34  * actually be stored in the memory space of main's small *
35  * and big variables.                                     *
36  *********************************************************/
```

(program continues)

Program 6-25 *(continued)*

```
37 void getNums(int &input1, int &input2)
38 {
39    cout << "Enter an integer: ";
40    cin  >> input1;
41    cout << "Enter a second integer: ";
42    cin  >> input2;
43 }
44
45 /*********************************************************
46  *                     orderNums                         *
47  * The arguments passed into num1 and num2 are passed by  *
48  * reference so that if they are out of order main's      *
49  * variables small and big can be swapped. Just swapping  *
50  * num1 and num2 in orderNum's local memory would not     *
51  * accomplish the desired result.                         *
52  *********************************************************/
53 void orderNums (int &num1, int &num2)
54 {
55    int temp;
56
57    if (num1 > num2)  // If the numbers are out of order, swap them
58    {   temp = num1;
59        num1 = num2;
60        num2 = temp;
61    }
62 }
```

Program Output With Example Input Shown in Bold

```
Enter an integer: 10[Enter]
Enter a second integer: 5[Enter]
The two input numbers ordered smallest to biggest are 5 and 10
```

Passing Files to Functions

As mentioned previously, reference parameters should always be used when a file stream object is passed to a function. Program 6-26 illustrates how to pass a file to a function. The `weather.dat` file used by the program contains the following seven values: 72 83 71 69 75 77 70.

Program 6-26

```
1 // This program reads 7 daily high temperatures from a file
2 // and displays them. It demonstrates how to pass a file to a
3 // function. The function argument, which is a file stream object,
4 // must be passed by reference.
5 #include <iostream>
6 #include <fstream>
7 using namespace std;
8
9 void readFile(ifstream&);        // Function prototype
10
```

(program continues)

Program 6-26 *(continued)*

```
11 int main()
12 {
13    ifstream dataIn;
14
15    dataIn.open("weather.dat");
16    if (dataIn.fail())
17       cout << "Error opening data file.\n";
18    else
19    {  readFile(dataIn);
20       dataIn.close();
21    }
22    return 0;
23 }
24
25 /*******************************************************
26  *                     readFile                        *
27  * This function reads and displays the contents of the *
28  * input file whose file stream object is passed to it. *
29  *******************************************************/
30 void readFile(ifstream &someFile)
31 {
32    int temperature;
33
34    while (someFile >> temperature)
35       cout << temperature << "  ";
36    cout << endl;
37 }
```

Program Output

```
72  83  71  69  75  77  70
```



Checkpoint

6.19 What kinds of values may be specified as default arguments?

6.20 Write the prototype and header for a function called `compute`. The function should have three parameters: an `int`, a `double`, and a `long` (not necessarily in that order). The `int` parameter should have a default argument of 5, and the `long` parameter should have a default argument of 65536. The `double` parameter should not have a default argument.

6.21 Write the prototype and header for a function called `calculate`. The function should have three parameters: an `int`, a reference to a `double`, and a `long` (not necessarily in that order.) Only the `int` parameter should have a default argument, which is 47.

6.22 What is the output of the following program?

```
#include <iostream>
using namespace std;

void test(int = 2, int = 4, int = 6);
```

```
int main()
{
   test();
   test(6);
   test(3, 9);
   test(1, 5, 7);
   return 0;
}

void test (int first, int second, int third)
{
   first += 3;
   second += 6;
   third += 9;
   cout << first << " " << second << " " << third << endl;
}
```

6.23 The following program asks the user to enter two numbers. What is the output of the program if the user enters 12 and 14?

```
#include <iostream>
using namespace std;

void func1(int &, int &);
void func2(int &, int &, int &);
void func3(int, int, int);

int main()
{
   int x = 0, y = 0, z = 0;
   cout << x << " " << y << z << endl;
   func1(x, y);
   cout << x << " " << y << z << endl;
   func2(x, y, z);
   cout << x << " " << y << z << endl;
   func3(x, y, z);
   cout << x << " " << y << z << endl;
   return 0;
}

void func1(int &a, int &b)
{  cout << "Enter two numbers: ";
   cin  >> a >> b;
}

void func2(int &a, int &b, int &c)
{  b++;
   c--;
   a = b + c;
}

void func3(int a, int b, int c)
{  a = b - c;
}
```

6.14 Overloading Functions

CONCEPT: **Two or more functions may have the same name, as long as their parameter lists are different.**

Sometimes you will create two or more functions that perform the same operation, but use a different set of parameters, or parameters of different data types. For instance, in Program 6-12 there is a `square` function that uses a `double` parameter. But suppose you also wanted a `square` function that works exclusively with integers and accepts an `int` as its argument. Both functions would do the same thing, return the square of their argument. The only difference is the data type involved in the operation. If you were to use both these functions in the same program, you could assign a unique name to each function. For example, the function that squares an `int` might be named `squareInt`, and the one that squares a `double` might be named `squareDouble`. C++, however, allows you to *overload* function names. That means you may assign the same name to multiple functions, as long as their parameter lists are different. Program 6-27 uses two overloaded `square` functions.

Program 6-27

```
1  // This program uses overloaded functions.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  // Function prototypes
7  int square(int);
8  double square(double);
9
10 int main()
11 {
12    int userInt;
13    double userReal;
14
15    // Get an int and a double
16    cout << "Enter an integer and a floating-point value: ";
17    cin  >> userInt >> userReal;
18
19    // Display their squares
20    cout << "Here are their squares: ";
21    cout << fixed << showpoint << setprecision(2);
22    cout << square(userInt) << " and " << square(userReal) << endl;
23    return 0;
24 }
25
26 /************************************************
27  *           overloaded function square         *
28  * This function returns the square of the value *
29  * passed into its int parameter.                *
30  ************************************************/
31 int square(int number)
```

(program continues)

Program 6-27 *(continued)*

```
32 {
33    return number * number;
34 }
35
36 /************************************************
37  *           overloaded function square         *
38  * This function returns the square of the value *
39  * passed into its double parameter.             *
40  ************************************************/
41 double square(double number)
42 {
43    return number * number;
44 }
```

Program Output with Example Input Shown in Bold

```
Enter an integer and a floating-point value: 12 4.2[Enter]
Here are their squares: 144 and 17.64
```

Here are the headers for the `square` functions used in Program 6-27:

```
int square(int number)
double square(double number)
```

In C++, each function has a signature. The *function signature* is the name of the function and the data types of the function's parameters in the proper order. The `square` functions in Program 6-27 would have the following signatures:

```
square(int)
square(double)
```

When an overloaded function is called, C++ uses the function signature to distinguish it from other functions with the same name. In Program 6-27, when an `int` argument is passed to `square`, the version of the function that has an `int` parameter is called. Likewise, when a `double` argument is passed to `square`, the version with a `double` parameter is called.

Note that the function's return value is not part of the signature. The following functions could *not* be used in the same program because their parameter lists aren't different.

```
int square(int number)
{
   return number * number
}

double square(int number)  // Wrong! Parameter lists must differ
{
   return number * number
}
```

Overloading is also convenient when there are similar functions that use a different number of parameters. For example, consider a program with functions that return the sum of

integers. One returns the sum of two integers, another returns the sum of three integers, and yet another returns the sum of four integers. Here are their function headers:

```
int sum(int num1, int num2)
int sum(int num1, int num2, int num3)
int sum(int num1, int num2, int num3, int num4)
```

Because the number of parameters is different in each, they may all be used in the same program. Program 6-28 uses two functions, each named `calcWeeklyPay`, to determine an employee's gross weekly pay. One version of the function uses an `int` and a `double` parameter, while the other version only uses a `double` parameter.

Program 6-28

```
1  // This program demonstrates overloaded functions to calculate
2  // the gross weekly pay of hourly-wage or salaried employees.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  // Function prototypes
8  char getChoice();
9  double calcWeeklyPay(int, double);
10 double calcWeeklyPay(double);
11
12 int main()
13 {
14    char selection;              // Menu selection
15    int worked;                  // Weekly hours worked
16    double rate,                 // Hourly pay rate
17           yearly;               // Annual salary
18
19    // Set numeric output formatting
20    cout << fixed << showpoint << setprecision(2);
21
22    // Display the menu and get a selection
23    cout << "Do you want to calculate the weekly pay of\n";
24    cout << "(H) an hourly-wage employee, or \n";
25    cout << "(S) a salaried employee? ";
26    selection = getChoice();
27
28    // Process the menu selection
29    switch (selection)
30    {
31       // Hourly employee
32       case 'H' :
33       case 'h' :  cout << "How many hours were worked? ";
34                   cin  >> worked;
35                   cout << "What is the hourly pay rate? ";
36                   cin  >> rate;
37                   cout << "The gross weekly pay is $";
38                   cout << calcWeeklyPay(worked, rate) << endl;
39                   break;
40
```

(program continues)

Program 6-28 *(continued)*

```
41       // Salaried employee
42       case 'S' :
43       case 's' :     cout << "What is the annual salary? ";
44                      cin  >> yearly;
45                      cout << "The gross weekly pay is $";
46                      cout << calcWeeklyPay(yearly) << endl;
47    }
48    return 0;
49 }
50
51 /**************************************************
52  *                    getChoice                   *
53  * Accepts and returns user's validated menu choice. *
54  **************************************************/
55 char getChoice()
56 {
57    char letter;        // Holds user's letter choice
58
59    // Get the user's choice
60    cin >> letter;
61
62    // Validate the choice
63    while (letter != 'H' && letter != 'h'
64        && letter != 'S' && letter != 's')
65    {
66       cout << "Enter H or S: ";
67       cin  >> letter;
68    }
69    // Return the choice
70    return letter;
71 }
72
73 /************************************************************
74  *            overloaded function calcWeeklyPay             *
75  * This function calculates and returns the gross weekly pay *
76  * of an hourly-wage employee. Parameters hours and payRate  *
77  * hold the number of hours worked and the hourly pay rate.  *
78  ************************************************************/
79 double calcWeeklyPay(int hours, double payRate)
80 {
81    return hours * payRate;
82 }
83
84 /************************************************************
85  *            overloaded function calcWeeklyPay             *
86  * This function calculates and returns the gross weekly pay *
87  * of a salaried employee. The parameter annSalary holds the *
88  * employee's annual salary.                                 *
89  ************************************************************/
90 double calcWeeklyPay(double annSalary)
91 {
92    return annSalary / 52.0;
93 }
```

(program continues)

Program 6-28 *(continued)*

Program Output with Example Input Shown in Bold

```
Do you want to calculate the weekly pay of
(H) an hourly-wage employee, or
(S) a salaried employee? H[Enter]
How many hours were worked? 40[Enter]
What is the hourly pay rate? 18.50[Enter]
The gross weekly pay is $740.00
```

Program Output with Other Example Data Shown in Bold

```
Do you want to calculate the weekly pay of
(H) an hourly-wage employee, or
(S) a salaried employee? S[Enter]
What is the annual salary? 48000.00[Enter]
The gross weekly pay is $923.08
```

6.15 The exit() Function

CONCEPT: The `exit()` function causes a program to terminate, regardless of which function or control mechanism is executing.

A C++ program stops executing when a `return` statement in function `main` is encountered. When other functions end, however, the program does not stop. Control of the program goes back to the place immediately following the function call. Sometimes, however, the programmer wishes, under certain conditions, to terminate a program in a function other than `main`. To accomplish this, the `exit` function is used.

When the `exit` function is called, it causes the program to stop, regardless of which function contains the call. Program 6-29 demonstrates this.

Program 6-29

```
1  // This program shows how the exit function causes a program
2  // to stop executing.
3  #include <iostream>
4  #include <cstdlib>    // Needed to use the exit function
5  using namespace std;
6
7  // Function prototype
8  void someFunction();
9
10 int main()
11 {
12    someFunction();
13    return 0;
14 }
15
```

(program continues)

Program 6-29 *(continued)*

```
16 /***************************************************************
17  *                          someFunction                       *
18  * This function demonstrates that exit() can be used to end   *
19  * a program from a function other than main. This is not      *
20  * considered good programming practice and should normally    *
21  * be done only to signal that an error condition has occurred. *
22  ***************************************************************/
23 void someFunction()
24 {
25    cout << "This program terminates with the exit function.\n";
26    cout << "Bye!\n";
27    exit(0);
28    cout << "This message will never be displayed\n";
29    cout << "because the program has already terminated.\n";
30 }
```

Program Output

```
This program terminates with the exit function.
Bye!
```

To use the `exit` function, be sure to include the `cstdlib` header file. Notice the function takes an integer argument. This argument is the exit code you wish the program to pass back to the computer's operating system. This code is sometimes used outside of the program to indicate whether the program ended successfully or as the result of a failure. In Program 6-29, the exit code zero is passed, which commonly indicates a successful exit. If you are unsure which code to use with the `exit` function, there are two named constants, `EXIT_FAILURE` and `EXIT_SUCCESS`, defined in `cstdlib` for you to use. The constant `EXIT_FAILURE` is defined as the termination code that commonly represents an unsuccessful exit under the current operating system. Here is an example of its use:

```
exit(EXIT_FAILURE);
```

The constant `EXIT_SUCCESS` is defined as the termination code that commonly represents a successful exit under the current operating system. Here is an example:

```
exit(EXIT_SUCCESS);
```

Because it is considered good programming practice to terminate a program at the end of the main function, many programmers use `exit()` only to handle error conditions.



WARNING! `exit()` is a nonstructured programming technique. Use with caution.



Checkpoint

6.24 Is it required that overloaded functions have different return types, different parameter lists, or both?

6.25 What is the output of the following program?

```
#include <iostream>
#include <cstdlib>
using namespace std;

void showVals(double, double);

int main()
{
   double x = 1.2, y = 4.5;

   showVals(x, y);
   return 0;
}

void showVals(double p1, double p2)
{
   cout << p1 << endl;
   exit(0);
   cout << p2 << endl;
}
```

6.26 What is the output of the following program?

```
#include <iostream>
using namespace std;
int manip(int);
int manip(int, int);
int manip(int, double);

int main()
{
   int x = 2, y= 4, z;
   double a = 3.1;

   z = manip(x) + manip(x, y) + manip(y, a);
   cout << z << endl;
   return 0;
}

int manip(int val)
{
   return val + val * 2;
}

int manip(int val1, int val2)
{
   return (val1 + val2) * 2;
}

int manip(int val1, double val2)
{
   return val1 * static_cast<int>(val2);
}
```

6.16 Stubs and Drivers

Stubs and drivers are very helpful tools for testing and debugging programs that use functions. They allow you to test the individual functions in a program, in isolation from the parts of the program that call the functions.

A *stub* is a dummy function that is called instead of the actual function it represents. It usually displays a test message acknowledging that it was called, and nothing more. For example, if a stub were used for the `showFees` function in Program 6-14 (the modular health club membership program), it might look like this:

```
// Stub for the showFees function
void showFees(string memberType, double rate, int months)
{
   cout << "The function showFees was called with arguments:\n"
        << "Member type: "  << memberType << endl
        << "rate: " << rate << endl
        << "months: " << months << endl;
}
```

Here is example output of the program if it were run with this stub instead of with the actual `showFees` function. Input is shown in bold.

```
      Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program
```

1[Enter]

```
For how many months? 3[Enter]
The function showFees was called with arguments:
Member type: Adult
rate: 40
months: 3

      Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program
```

4[Enter]

As you can see, by replacing an actual function with a stub, you can concentrate your testing efforts on the parts of the program that call the function. Primarily, the stub allows you to determine whether your program is calling a function when you expect it to and confirm that valid values are being passed to the function. If the stub represents a function that returns a value, then the stub should return a test value. This helps you confirm that the return value is being handled properly. When the parts of the program that call a function

are debugged to your satisfaction, you can move on to testing and debugging the actual functions themselves. This is where drivers become useful.

A *driver* is a program that tests a function by simply calling it. If the function accepts any arguments, the driver passes test data. If the function returns a value, the driver displays the return value on the screen. This allows you to see how the function performs in isolation from the rest of the program it will eventually be part of. Program 6-30 is a driver for testing the `showFees` function in the health club membership program.

Program 6-30

```
// This program is a driver for testing the showFees function.
#include <iostream>
#include <string>
using namespace std;

// Function prototype
void showFees(string, double, int);

int main()
{
   cout << "Calling the showFees function with arguments "
        << "Adult, 40.00, 3.\n";
   showFees("Adult", 40.00, 3);

   cout << "Calling the showFees function with arguments "
        << "Child, 20.00, 2.\n";
   showFees("Child", 20.00, 2);

   cout << "Calling the showFees function with arguments "
        << "Senior, 30.00, 4.\n";
   showFees("Senior", 30.00, 4);

   return 0;
}

/**************************************************
 *                    showFees                    *
 * This function uses the membership type, monthly *
 * rate and number of months passed to it as arguments *
 * to compute and print a member's total charges.  *
 **************************************************/
void showFees(string memberType, double rate, int months)
{
   cout << endl
        << "Membership Type : " << memberType << "    "
        << "Monthly rate $"     << rate   << endl
        << "Number of months: " << months << endl
        << "Total charges   : $"<< (rate * months)
        << endl << endl;
}
```

(program continues)

Program 6-30 *(continued)*

Program Output

```
Calling the showFees function with arguments Adult, 40.00, 3.

Membership Type : Adult    Monthly rate $40
Number of months: 3
Total charges   : $120

Calling the showFees function with arguments Child, 20.00, 2.

Membership Type : Child    Monthly rate $20
Number of months: 2
Total charges   : $40

Calling the showFees function with arguments Senior, 30.00, 4.

Membership Type : Senior    Monthly rate $30
Number of months: 4
Total charges   : $120
```

As shown in Program 6-30, a driver can be used to thoroughly test a function. It can repeatedly call the function with different test values as arguments. When the function performs as desired, it can be placed into the actual program it will be part of.

6.17 Little Lotto Case Study

Problem Statement

The mathematics department of Jefferson Junior High School wants a program developed that will illustrate basic probability for their students in an entertaining way. In particular they want a program called "Little Lotto" that simulates a lottery. In this program students can specify the number of numbers in the selection set (1-12) and the number of numbers patrons must pick and match to the winning numbers (between 1 and the size of the selection set). The order of the selected numbers is not significant. It will report the patron's chances of winning as both a number and a probability.

Example Output

This example output clarifies exactly what the department wants the program to do.

```
This program will tell you your probability of winning "Little Lotto".

How many numbers (1-12) are there to pick from? 12
How many numbers must you pick to play? 5

Your chance of winning the lottery is 1 chance in 792.
This is a probability of 0.0013
```

Program Design

Program Steps

The program must carry out the following general steps:

1. Get and validate how many numbers there are to choose from (n).
2. Get and validate how many of these numbers must be selected (k).
3. Compute the number of ways a set of k items can be selected from a set of n items.
4. Report to the player his chance of winning and his probability of winning.

Program Modules

The program will be designed as a set of modules, each having a specific function. Here are the modules that will be used:

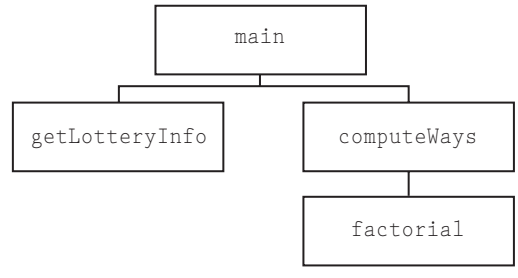
Table 6-1 Little Lotto Program Modules

Function	Description
`main`	This function explains the "game", organizes calls to other functions, and reports results.
`getLotteryInfo`	This function gets and validates the number of numbers to select from (n) and the number that must be chosen (k).
`computeWays`	This function computes the number of different sets of size k that can be chosen from n numbers.
`factorial`	This function computes factorials. It is used by `computeWays`.

Program Organization

In previous chapters hierarchy charts were used to illustrate the relationship of actions that a program must carry out. However, they are more commonly used to illustrate the relationship of program modules in a program that is organized into a set of functions. The following hierarchy chart illustrates the organization of the Little Lotto program. Notice that it clarifies which functions call which other functions.

Figure 6-15



Variables whose values will be input

```
int  pickFrom        // Number of numbers available to select from
int  numPicks        // Number of numbers that must be chosen
```

Variables and values whose values will be output

```
    long int ways          // Number of different possible selections
                           // Only 1 of these can "win"
    1.0 / ways             // Probability of winning
```

Detailed Pseudocode for Each Module

In a modular program, a separate pseudocode routine should be created to capture the logic of each function. Here is the pseudocode for each function in the Little Lotto program.

```
main
   Display information on what the program does
   Call getLotteryInfo  // Puts value in pickFrom and numPicks variables
   Call computeWays     // Returns number of ways numbers can be selected
   Store the returned result in the ways variable
   Display ways and 1 / ways
End main

getLotteryInfo          // Places inputs in reference variables
   Input pickFrom
   While pickFrom < 1 or pickFrom > 12
      Display an error message
      Input pickFrom
   End while
   Input numPicks
   While numPicks < 1 or numPicks > pickFrom
      Display an error message
      Input pickFrom
   End while
End getLotteryInfo

computeWays             // Receives pickFrom as n and numPicks as k
   Call factorial 3 times to get information for its calculations
   Return      factorial(n)
            factorial(k) *  factorial (n-k)
End computeWays

factorial      // Receives number whose factorial is to be calculated
   factTotal = 1
   Loop for count = number down to 1
      factTotal = factTotal * count
   End Loop
   Return factTotal
End factorial
```

The Program

The next step, after the pseudocode has been checked for logic errors, is to expand the pseudocode into the final program. This is shown in Program 6-31.

Program 6-31

```
// This program finds the probability of winning a "mini" lottery
// when the user's set of numbers must exactly match the set drawn
// by the lottery organizers. It uses three functions.
#include <iostream>
#include <iomanip>
using namespace std;

// Function prototypes
void getLotteryInfo(int&, int&);
long int computeWays(int, int);
long int factorial(int);

int main()
{
   int pickFrom,           // The number of numbers to pick from
       numPicks;           // The number of numbers to select
   long int ways;          // The number of different possible
                           // ways to pick the set of numbers

   cout << "This program will tell you your probability of "
        << "winning \"Little Lotto\". \n";
   getLotteryInfo(pickFrom, numPicks);
   ways = computeWays(pickFrom, numPicks);

   cout << fixed << showpoint << setprecision(4);
   cout << "\nYour chance of winning the lottery is "
        << "1 chance in " << ways << ".\n";
   cout << "This is a probability of " << (1.0 / ways)    << "\n";
   return 0;
}

/*****************************************************************
 *                      getLotteryInfo                           *
 * Gets and validates lottery info. from the user and places it in *
 * reference parameters referencing variables in the main function.*
 *****************************************************************/
void getLotteryInfo(int &pickFrom, int &numPicks)
{
   cout << "\nHow many numbers (1-12) are there to pick from? ";
   cin  >> pickFrom;
   while (pickFrom < 1 || pickFrom > 12)
   {
      cout << "There must be between 1 and 12 numbers.\n"
           << "How many numbers (1-12) are there to pick from? ";
      cin  >> pickFrom;
   }
   cout << "How many numbers must you pick to play? ";
   cin  >> numPicks;
   while (numPicks < 1 || numPicks > pickFrom)
```

(program continues)

Program 6-31 *(continued)*

```
50    {
51       if (numPicks < 1)                          // too few picks
52          cout << "You must pick at least one number.\n";
53       else                                       // too many picks
54          cout << "You must pick " << pickFrom << " or fewer numbers.\n";
55
56       cout << "How many numbers must you pick to play? ";
57       cin  >> numPicks;
58    }
59 }
60
61 /*****************************************************************
62  *                       computeWays                             *
63  * Computes and returns the number of different possible sets    *
64  * of k numbers that can be chosen from a set of n numbers.      *
65  * The formula for this is   n!                                  *
66  *                          --------                             *
67  *                          k!(n-k)!                             *
68  *****************************************************************/
69 // Note that the computation is done in a way that does not require
70 // multiplying two factorials together. This is done to prevent any
71 // intermediate result becoming so large that it causes overflow.
72 long int computeWays(int n, int k)
73 {
74    return ( factorial(n) / factorial(k) / factorial (n-k) );
75 }
76
77 /*****************************************************************
78  *                          factorial                            *
79  * Computes and returns the factorial of the non-negative integer *
80  * passed to it. n! means n * (n-1) * (n-2) ... * 1              *
81  * 0! is a special case and is defined to be 1.                  *
82  *****************************************************************/
83 // Notice that if number equals 0, the loop condition will
84 // initially be false and the loop will never be executed.
85 // This will, correctly, leave factTotal = 1.
86
87 long int factorial(int number)
88 {
89    long int factTotal = 1;
90
91    for (int count = number; count >= 1; count--)
92    {
93       factTotal *= count;
94    }
95    return factTotal;
96 }
```

(program continues)

Program 6-31 *(continued)*

Program Output with Example Input Shown in Bold

```
This program will tell you your probability of winning "Little Lotto".

How many numbers (1-12) are there to pick from? 10[Enter]
How many numbers must you pick to play? 3[Enter]

Your chance of winning the lottery is 1 chance in 120.
This is a probability of 0.0083
```

High Adventure Travel Agency Case Study

The following additional case study, which contain applications of material introduced in Chapter 6, can be found on the student CD.

High Adventure Travel Agency offers four vacation packages for thrill-seeking customers. Each has its own rates and options and can be customized to meet individual customer preferences. This case study develops a modular program that calculates and itemizes charges for these packages.

6.18 Tying It All Together: *Glowing Jack-o-lantern*

Functions are not just practical. They are fun. True, they let you simplify programs by breaking them into smaller modules. And they minimize repetitive code. If you need to do the same thing in several different places in your program, you can just write a function to do it, then call that function from different places in the program instead of writing the same block of code more than once. But they also let you do new and fun things. This is because even though the function code is the same, it will behave differently every time it is called with different arguments.

For example, we could write the following `printSpaces` function and each time it will print a different number of spaces depending on the value passed in to its parameter `n`.

```
void printSpaces(int n)
{
   for (int space = 1; space <= n; space++)
      cout << " ";
}
```

Now that may not sound like fun, but let's see how we can use it to modify the colored alphabet program we created in Chapter 5's Tying It All Together section. We will make the letters appear to "climb down a set of stairs" by moving them across the screen as they are displayed. So that they will all fit on one screen, we will print them in pairs.

Recall from Chapter 5, however, that the function we are using to display output in color uses a Windows operating system function, so this program will only run on Windows systems.

Program 6-32

```
// This program writes the ABCs in green, red, and yellow,
// displaying them diagonally across the screen so they
// appear to be climbing down a staircase.
#include <iostream>
#include <windows.h>     // Needed to display colors and call Sleep
using namespace std;

// Prototype
void printSpaces(int n);

int main()
{
   // Bright Green = 10   Bright Red = 12   Bright Yellow = 14

   // Get the handle to standard output device (the console)
   HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);

   // Write the ABCs using 3 colors
   // Display 2 per line, stair stepping across the screen
   int color = 10;     // Staring color = green
   for (char letter = 'A'; letter <= 'Z'; letter+=2)
   {
      SetConsoleTextAttribute (screen, color);    // Set the color
      printSpaces(letter-'A');                    // Indent
      cout << letter                              // Print 2 letters
           << static_cast<char>(letter+1) << endl;
      color +=2;                                  // Choose next color
      if (color > 14)
         color = 10;

      Sleep(280);  // Pause between characters to watch them appear
   }
   // Restore normal text attribute (i.e. white)
   SetConsoleTextAttribute(screen, 7);
   return 0;
}

/*****************************************
 *               printSpaces             *
 * Prints n spaces where n is passed as an  *
 * argument to the function.               *
 *****************************************/
void printSpaces(int n)
{
   for (int space = 1; space <= n; space++)
      cout << " ";
}
```

Run the program and view the results. The output display should look like the one below, but in color of course.

```
AB
  CD
    EF
      GH
        IJ
          KL
            MN
              OP
                QR
                  ST
                    UV
                      WX
                        YZ
```

Now, can you modify the program to make the letters appear to climb UP the stairs? The program will still print starting with the top line and move down the screen, but the final display should look like this:

```
                        YZ
                      WX
                    UV
                  ST
                QR
              OP
            MN
          KL
        IJ
      GH
    EF
  CD
AB
```

If you have trouble figuring this out, the solution can be found in the Program 6-32B. cpp file on the student CD.

Now let's use a function to do something else that's fun. We will modify the Smiley Face we created in Chapter 2's Tying It All Together to become a spooky Jack-o-lantern glowing in the dark, and we will let the user pick what color to display it in.

Program 6-33

```
1  // This program displays a Jack-o-lantern glowing in the dark.
2  // It lets the user select what color it should be.
3  #include <iostream>
4  #include <windows.h>          // Needed to display colors
5  using namespace std;
6
7  // Function prototypes
8  void displayMenu();
9  int getChoice();
10 void makeJackOLantern();
11
12 int main()
13 {
14    const int quit = 6;
15    int colorChoice;
16    // Get the handle to standard output device (the console)
17    HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);
18
```

(program continues)

Program 6-33 *(continued)*

```
   do
   {  SetConsoleTextAttribute(screen, 7);  // Set to white on black
      displayMenu();                       // for menu display
      colorChoice = getChoice();

      if (colorChoice != quit)
      {  SetConsoleTextAttribute (screen, colorChoice + 9);
         makeJackOLantern();
      }
   } while (colorChoice != quit);
   return 0;
}

/*************************************************
 *                    displayMenu                *
 * This function displays the menu of color choices.*
 *************************************************/
void displayMenu()
{  system("cls");                // Clear the screen
   cout << "I will draw a Jack-o-lantern. What color should it be?\n\n"
        << "Enter 1 for Green     2 for Blue     3 for Red \n"
        << "      4 for Purple    5 for Yellow   6 to quit: ";
}

/*************************************************
 *                    getChoice                  *
 *  This function inputs, validates, and returns  *
 *  the user's menu choice.                       *
 *************************************************/
int getChoice()
{
   int choice;

   cin >> choice;
   while (choice < 1 || choice > 6)
   {  cout << "The only valid choices are 1-6.  Please re-enter. ";
      cin  >> choice;
   }
   return choice;
}

/*********************************************
 *             makeJackOLantern              *
 *  This function draws a Jack-o-lantern     *
 *  in whatever color the user selected.     *
 *********************************************/
```

(program continues)

Program 6-33 *(continued)*

```
65 void makeJackOLantern()
66 {
67    cout << "\n\n";
68    cout << "                              ^   ^  \n";
69    cout << "                                *    \n";
70    cout << "                               \\___/   " << endl;
71    cout << "\n\n             Press ENTER to return to the menu." ;
72    cin.get();     // Clear the previous \n out of the input buffer
73    cin.get();     // Wait for the user to press ENTER
74 }
```

Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. The _____ is the part of a function definition that shows the function name, return type, and parameter list.
2. If a function doesn't return a value, the word _____ will appear as its return type.
3. If function `showValue` has the following header: `void showValue(int quantity)` you would use the statement _____ to call it with the argument 5.
4. Either a function's _____ or its _____ must precede all calls to the function.
5. Values that are sent into a function are called _____.
6. Special variables that hold copies of function arguments are called _____.
7. When only a copy of an argument is passed to a function, it is said to be passed by _____.
8. A(n)_____ eliminates the need to place a function definition before all calls to the function.
9. A(n)_____ variable is defined inside a function and is not accessible outside the function.
10. _____ variables are defined outside all functions and are accessible to any function within their scope.
11. _____ variables provide an easy way to share large amounts of data among all the functions in a program.
12. Unless you explicitly initialize numeric global variables, they are automatically initialized to _____.
13. If a function has a local variable with the same name as a global variable, only the _____ variable can be seen by the function.
14. _____ local variables retain their value between function calls.
15. The _____ statement causes a function to end immediately.
16. _____ arguments are passed to parameters automatically if no argument is provided in the function call.

17. When a function uses a mixture of parameters with and without default arguments, the parameters with default arguments must be defined _____.
18. The value of a default argument must be a(n)_____.
19. When used as parameters, _____ variables allow a function to access the parameter's original argument.
20. Reference variables are defined like regular variables, except there is a(n) _____ in front of the name.
21. Reference variables allow arguments to be passed by _____.
22. The _____ function causes a program to terminate immediately.
23. Two or more functions may have the same name, as long as their _____ are different.
24. What is the advantage of breaking your application's code into several small functions?
25. What is the difference between an argument and a parameter variable?
26. When a function accepts multiple arguments, does it matter what order the arguments are passed in?
27. What does it mean to overload a function?
28. If you are writing a function that accepts an argument and you want to make sure the function cannot change the value of the argument, what should you do?
29. Give an example where an argument should be passed by reference.
30. How do you return a value from a function?
31. Can a function have a local variable with the same name as a global variable?
32. When should a `static` local variable be used?

Algorithm Workbench

33. The following statement calls a function named `half`, which returns a value that is half that of the argument passed to it. Assume that `result` and `number` have both been defined to be `double` variables. Write the `half` function.

```
result = half(number);
```

34. A program contains the following function.

```
int cube(int num)
{
     return num * num * num;
}
```

Write a statement that passes the value 4 to this function and assigns its return value to the variable `result`.

35. Write a function, named `timesTen`, that accepts an integer argument. When the function is called, it should display the product of its argument multiplied times 10.
36. A program contains the following function.

```
void display(int arg1, double arg2, char arg3)
{
   cout << "Here are the values: "
        << arg1 << " " << arg2 << " " << arg3 << endl;
}
```

Write a statement that calls the function and passes the following variables to it:

```
int age;
double income;
char initial;
```

37. Write a function named `getNumber`, which uses a reference parameter to accept an integer argument. The function should prompt the user to enter a number in the range of 1 through 100. The input should be validated and stored in the parameter variable.
38. Write a function named `biggest` that receives three integer arguments and returns the largest of the three values.

Find the Errors

39. Each of the following functions has errors. Locate as many errors as you can.

A)
```
void total(int value1, value2, value3)
{
   return value1 + value2 + value3;
}
```

B)
```
double average(int value1, int value2, int value3)
{
   double average;

   average = value1 + value2 + value3 / 3;
}
```

C)
```
void area(int length = 30, int width)
{
   return length * width;
}
```

D)
```
void getValue(int value&)
{
   cout << "Enter a value: ";
   cin  >> value&;
}
```

E)
```
// Overloaded functions
int getValue()
{
   int inputValue;
   cout << "Enter an integer: ";
   cin  >> inputValue;
   return inputValue;
}
double getValue()
{
   double inputValue;

   cout << "Enter a floating-point number: ";
   cin  >> inputValue;
   return inputValue;
}
```

Soft Skills

Programmers need to develop the ability to break a large problem into a set of manageable components, or modules, each of which can focus on handling one specific task. If these tasks are large, they may be divided even further into a set of subtasks. Each component can then be programmed as a separate function. Often there is more than one acceptable way to divide a program into modules and to organize the modules. However, in general, if module A calls module B then module B should carry out some subtask that helps module A perform its function.

40. Read the following program statement and then come to class prepared to discuss how you would design the program. How many modules would you use? What task would each module handle? How would you organize the modules? That is, which modules would call which other modules? Be prepared to state the advantages of the design you have chosen.

Artistic Solutions Paint Job Estimator

Artistic Solutions, a painting company, has determined that for every 160 square feet of wall space, one gallon of paint and three hours of labor are required. The company charges $28.00 per hour for labor. Design a modular program that allows the user to enter the number of rooms that are to be painted, the approximate square feet of wall space in each room (it may differ from room to room), and the price of the paint per gallon. It should then create a report that includes a fancy company header and displays the following information:

- The number of gallons of paint required (rounded up to the next full gallon)
- The hours of labor required
- The cost of the paint
- The labor charges
- The total cost of the paint job

Input validation: The program should not accept a value less than 1 or more than 12 for the number of rooms or a value less than 100 for the square footage of a room. It should not accept a value less than $10.00 or more than $25.00 for the price of a gallon of paint.

Programming Challenges

1. Markup



VideoNote
Solving the Markup Problem

Write a program that asks the user to enter an item's wholesale cost and its markup percentage. It should then display the item's retail price. For example:

- If an item's wholesale cost is 5.00 and its markup percentage is 100%, then the item's retail price is 10.00.
- If an item's wholesale cost is 5.00 and its markup percentage is 50%, then the item's retail price is 7.50.

The program should have a function named `calculateRetail` that receives the wholesale cost and the markup percentage as arguments, and returns the retail price of the item.

Input Validation: Do not accept negative values for either the wholesale cost of the item or the percent markup.

2. Celsius Temperature Table

The formula for converting a temperature from Fahrenheit to celsius is

$$C = \frac{5}{9}(F - 32)$$

where F is the Fahrenheit temperature and C is the Celsius temperature. Write a function named `celsius` that accepts a Fahrenheit temperature as an argument. The function should return the temperature, converted to Celsius. Demonstrate the function by calling it in a loop that displays a table of the Fahrenheit temperatures 0 through 20 and their Celsius equivalents.

myCodeMate

3. Falling Distance

The following formula can be used to determine the distance an object falls due to gravity in a specific time period:

$$d = \frac{1}{2} gt^2$$

The variables in the formula are as follows: d is the distance in meters, g is 9.8, and t is the time in seconds that the object has been falling.

Write a function named `fallingDistance` that accepts an object's falling time (in seconds) as an argument. The function should return the distance, in meters, that the object has fallen during that time interval. Write a program that demonstrates the function by calling it in a loop that passes the values 1 through 10 as arguments, and displays the return value.

4. Kinetic Energy

In physics, an object that is in motion is said to have kinetic energy. The following formula can be used to determine a moving object's kinetic energy:

$$KE = \frac{1}{2} mv^2$$

The variables in the formula are as follows: KE is the kinetic energy in joules, m is the object's mass in kilograms, and v is the object's velocity in meters per second.

Write a function named `kineticEnergy` that accepts an object's mass (in kilograms) and velocity (in meters per second) as arguments. The function should return the amount of kinetic energy that the object has. Demonstrate the function by calling it in a program that asks the user to enter values for mass and velocity.

myCodeMate

5. Winning Division

Write a program that determines which of a company's four divisions (Northeast, Southeast, Northwest, and Southwest) had the greatest sales for a quarter. It should include the following two functions, which are called by `main`.

- `double getSales()` is passed the name of a division. It asks the user for a division's quarterly sales figure, validates the input, then returns it. It should be called once for each division.
- `void findHighest()` is passed the four sales totals. It determines which is the largest and prints the name of the high grossing division, along with its sales figure.

Input Validation: Do not accept dollar amounts less than $0.00.

6. Safest Driving Area

Write a program that determines which of 5 geographic regions within a major city (north, south, east, west, and central) had the fewest reported traffic accidents last year. It should have the following two functions, which are called by `main`.

- `int getNumAccidents()` is passed the name of a region. It asks the user for the number of traffic accidents reported in that region during the last year, validates the input, then returns it. It should be called once for each city region.
- `void findLowest()` is passed the five accident totals. It determines which is the smallest and prints the name of the region, along with its accident figure.

Input Validation: Do not accept an accident number that is less than 0.

myCodeMate

7. Lowest Score Drop

- Write a program that calculates the average of a group of test scores, where the lowest score in the group is dropped. It should use the following functions:
- `void getScore()` should ask the user for a test score, store it in a reference parameter variable, and validate it. This function should be called by `main` once for each of the five scores to be entered.
- `void calcAverage()` should calculate and display the average of the four highest scores. This function should be called just once by `main`, and should be passed the five scores.
- `int findLowest()` should find and return the lowest of the five scores passed to it. It should be called by `calcAverage`, which uses the function to determine one of the five scores to drop.

Input Validation: Do not accept test scores lower than 0 or higher than 100.

8. Star Search

A particular talent competition has 5 judges, each of whom awards a score between 0 and 10 to each performer. Fractional scores, such as 8.3, are allowed. A performer's final score is determined by dropping the highest and lowest score received, then averaging the 3 remaining scores. Write a program that uses these rules to calculate and display a contestant's score. It should include the following functions:

- `void getJudgeData()` should ask the user for a judge's score, store it in a reference parameter variable, and validate it. This function should be called by `main` once for each of the 5 judges.
- `double calcScore()` should calculate and return the average of the 3 scores that remain after dropping the highest and lowest scores the performer received. This function should be called just once by `main`, and should be passed the 5 scores.

The last two functions, described below, should be called by `calcScore`, which uses the returned information to determine which of the scores to drop.

- `int findLowest()` should find and return the lowest of the 5 scores passed to it.
- `int findHighest()` should find and return the highest of the 5 scores passed to it.

Input Validation: Do not accept judge scores lower than 0 or higher than 10.

9. `isPrime` Function

A prime number is an integer greater than 1 that is evenly divisible by only 1 and itself. For example, the number 5 is prime because it can only be evenly divided by 1 and 5. The number 6, however, is not prime because it can be divided by 1, 2, 3, and 6.

Write a Boolean function named `isPrime`, which takes an integer as an argument and returns `true` if the argument is a prime number, and `false` otherwise. Demonstrate the function in a complete program.



TIP: Recall that the % operator divides one number by another, and returns the remainder of the division. In an expression such as num1 % num2, the % operator will return 0 if num1 is evenly divisible by num2.

10. Present Value

Suppose you want to deposit a certain amount of money into a savings account, and then leave it alone to draw interest for the next 10 years. At the end of 10 years you would like to have $10,000 in the account. How much do you need to deposit today to make that happen? To find out you can use the following formula, which is known as the present value formula:

$$P = \frac{F}{(1 + r)^n}$$

The terms in the formula are as follows:

- P is the **present value**, or the amount that you need to deposit today.
- F is the **future value** that you want in the account (in this case, $10,000).
- r is the **annual interest rate** (expressed in decimal form, such as .042).
- n is the **number of years** that you plan to let the money sit in the account.

Write a program with a function named `presentValue` that performs this calculation. The function should accept the future value, annual interest rate, and number of years as arguments. It should return the present value, which is the amount that you need to deposit today. Demonstrate the function in a program that lets the user experiment with different values for the formula's terms.

11. Stock Profit

The profit from the sale of a stock can be calculated as follows:

$$\text{Profit} = ((NS \times SP) - SC) - ((NS \times PP) + PC)$$

where NS is the number of shares, SP is the sale price per share, SC is the sale commission paid, PP is the purchase price per share, and PC is the purchase commission paid. If the calculation yields a positive value, then the sale of the stock resulted in a profit. If the calculation yields a negative number, then the sale resulted in a loss.

Write a function that accepts as arguments the number of shares, the purchase price per share, the purchase commission paid, the sale price per share, and the sale commission paid. The function should return the profit (or loss) from the sale of stock.

Demonstrate the function in a program that asks the user to enter the necessary data and displays the amount of the profit or loss.

12. Multiple Stock Sales

Use the function that you wrote for Programming Challenge 10 (Stock Profit) in a program that calculates the total profit or loss from the sale of multiple stocks. The program should ask the user for the number of stock sales, and the necessary data for each stock sale. It should accumulate the profit or loss for each stock sale and then display the total.

13. Order Status

The Middletown Wholesale Copper Wire Company sells spools of copper wiring for $100 each and ships them for $10 apiece. Write a program that displays the status of an order. It should use two functions. The first function asks for the following data and stores the input values in reference parameters.

- The number of spools ordered.
- The number of spools in stock.
- Any special shipping and handling charges (above the regular $10 rate).

The second function receives as arguments any values needed to compute and display the following information:

- The number of ordered spools ready to ship from current stock.
- The number of ordered spools on backorder (if the number ordered is greater than what is in stock).
- Total selling price of the portion ready to ship (the number of spools ready to ship times $100).
- Total shipping and handling charges on the portion ready to ship.
- Total of the order ready to ship.

The shipping and handling parameter in the second function should have the default argument 10.00.

> *Input Validation: Do not accept numbers less than 1 for spools ordered. Do not accept a number less than 0 for spools in stock or for shipping and handling charges.*

14. Overloaded Hospital

Write a program that computes and displays the charges for a patient's hospital stay. First, the program should ask if the patient was admitted as an in-patient or an out-patient. If the patient was an in-patient the following data should be entered:

- The number of days spent in the hospital
- The daily rate
- Charges for hospital services (lab tests, etc.)
- Hospital medication charges.

If the patient was an out-patient the following data should be entered:

- Charges for hospital services (lab tests, etc.)
- Hospital medication charges.

The program should use two overloaded functions to calculate the total charges. One of the functions should accept arguments for the in-patient data, while the other function accepts arguments for out-patient data. Both functions should return the total charges.

Input Validation: Do not accept negative numbers for any information.



15. Population

In a population, the birth rate is the percentage increase of the population due to births and the death rate is the percentage decrease of the population due to deaths. Write a program that asks for the following:

- The starting size of a population
- The annual birth rate
- The annual death rate
- The number of years to display

The program should then display the starting population and the projected population at the end of each year. It should use a function that calculates and returns the projected new size of the population after a year. The formula is

```
N = P(1 + B)(1 - D)
```

where `N` is the new population size, `P` is the previous population size, `B` is the birth rate, and `D` is the death rate.

Input Validation: Do not accept numbers less than 2 for the starting size. Do not accept negative numbers for birth rate or death rate. Do not accept numbers less than 1 for the number of years.

16. Transient Population

Modify Programming Challenge 15 to also consider the effect on population caused by people moving into or out of a geographic area. Given as input a starting population size, the annual birth rate, the annual death rate, the number of individuals that typically move into the area each year, and the number of individuals that typically leave the area each year, the program should project what the population will be `numYears` from now. You can either prompt the user to input a value for `numYears`, or you can set it within the program.

Input Validation: Do not accept numbers less than 2 for the starting size. Do not accept negative numbers for birth rate, death rate, arrivals or departures.

17. Using Files—Hospital Report

Modify Programming Challenge 14, Overloaded Hospital, to write the report it creates to a file. Print the contents of the file to hand in to your instructor.

Group Project

18. Using Files—Travel Expenses

This program should be designed and written by a team of students. Here are some suggestions:

- One student should design function `main`, which will call the other functions in the program. The rest of the functions should be designed by other team members.
- Analyze the program requirements so each student is given about the same work load.
- Decide on the function names, parameters, and return types in advance.
- Use stubs and drivers to test and debug the program.
- The program can be implemented either as a multi-file program, or all the functions can be cut and pasted into the main file.

Here is the assignment. Write a program that calculates and displays the total travel expenses of a business person on a trip. The program should have functions that ask for and return the following:

- The total number of days spent on the trip
- The time of departure on the first day of the trip, and the time of arrival back home on the last day of the trip
- The amount of any round-trip airfare
- The amount of any car rentals
- Miles driven, if a private vehicle was used. Vehicle allowance is $0.58 per mile.
- Parking fees. (The company allows up to $12 per day. Anything in excess of this must be paid by the employee.)
- Taxi fees. (The company allows up to $40 per day for each day a taxi was used. Anything in excess of this must be paid by the employee.)
- Conference or seminar registration fees
- Hotel expenses. (The company allows up to $90 per night for lodging. Anything in excess of this must be paid by the employee.)
- The cost of each meal eaten. On the first day of the trip, breakfast is allowed as an expense if the time of departure is before 7 a.m. Lunch is allowed if the time of departure is before noon. Dinner is allowed if the time of departure is before 6 p.m. On the last day of the trip, breakfast is allowed if the time of arrival is after 8 a.m. Lunch is allowed if the time of arrival is after 1 p.m. Dinner is allowed if the time of arrival is after 7 p.m. The program should only ask for the costs of allowable meals. (The company allows up to $18 for breakfast, $12 for lunch, and $20 for dinner. Anything in excess of this must be paid by the employee.)

The program should perform the necessary calculations to determine the total amount spent by the business traveler in each category (mileage charges, parking, hotel, meals, etc.) as well as the maximum amount allowed in each category. It should then create a well laid out expense report that includes the amount spent and the amount allowed in each category, as well as the total amount spent and total amount allowed for the entire trip. This report should be written to a file.

> *Input Validation: Do not accept negative numbers for any dollar amount or for miles driven in a private vehicle. Do not accept numbers less than 1 for the number of days. Only accept valid times for the time of departure and the time of arrival.*

CHAPTER 7

Introduction to Classes and Objects

TOPICS

7.1 Abstract Data Types
7.2 Object-Oriented Programming
7.3 Introduction to Classes
7.4 Introduction to Objects
7.5 Defining Member Functions
7.6 Constructors
7.7 Destructors
7.8 Private Member Functions
7.9 Passing Objects to Functions
7.10 Object Composition
7.11 Focus on Software Engineering: *Separating Class Specification, Implementation, and Client Code*
7.12 Input Validation Objects
7.13 Structures
7.14 Home Software Company OOP Case Study
7.15 Introduction to Object-Oriented Analysis and Design
7.16 Screen Control
7.17 Tying It All Together: *Yoyo Animation*

7.1 Abstract Data Types

CONCEPT: **An abstract data type (ADT) is a data type that specifies the values the data type can hold and the operations that can be done on them without the details of how the data type is implemented.**

Abstraction

An *abstraction* is a general model of something. It is a definition that includes only the general characteristics of an object without the details that characterize specific instances of the object.

An automobile provides an illustration of abstraction. Most people understand what an automobile is, and many people know how to drive one. Yet few people understand exactly how an automobile works or what all its parts are. This is a feature of abstraction.

Details of the internal components, organization, and operations of an object are kept separate from the description of what it can do and how to operate it. We are surrounded in our everyday lives with such examples of abstraction, from our microwaves and washing machines to our DVD players and computers. We know what these objects can do and we understand how to operate them, but most of us do not know, or care, how they work inside. We do not need to be concerned with this information.

The Use of Abstraction in Software Development

Abstraction occurs in programming as well. In order to focus on the bigger picture of creating a working application, a programmer needs to be able to use certain objects and routines without having to be concerned with the details of their implementation. You have been doing this since the beginning of this text when you used objects such as `cin` and `cout` and functions such as `sqrt` and `pow`. All you need to know to use the objects or functions is what they do and the interface for using them. For example, to use the `sqrt` function you only have to know its name and that it must be called with one numeric argument, the value whose square root is to be returned. To use the `pow` function you only have to know its name and that it must be called with two numeric arguments, the first being the value to be raised to a power and the second being the exponent. In neither case is it necessary to know what algorithm is used by the function to compute the result it returns.

Abstraction applies to data too. To use any data type you need to know just two things about it: what values it can hold and what operations apply to it. For example, to use a `double` you need to know that it can only hold numeric values, such as `5.0` or `–5.1`, and not strings, such as `"5.1"`. To use a `double` you also need to know what operations can be performed on it. It can be used with the addition, subtraction, multiplication, and division operators, but not with the modulus operator (which only works with integer operands, as in the expression `8 % 3`). You do not have to know anything else about a `double` to use it. You do not have to know how it is stored in memory or how the arithmetic operations that can be performed on it are carried out by the computer. This separation of a data type's logical properties from its implementation details is known as *data abstraction*.

Abstract Data Types

The term *abstract data type* (*ADT*) describes any data type for which the implementation details are kept separate from the logical properties needed to use it. In common usage, however, the term is applied only to data types created by the programmer. Often these data types can hold more than one value, as with classes, which you will learn about in this chapter. The programmer defines a set of values the data type can hold, defines a set of operations that can be performed on the data, and creates a set of functions to carry out these operations. In C++ and other object-oriented languages, programmer created ADTs are normally implemented as classes.

7.2 Object-Oriented Programming

CONCEPT: **Object-oriented programming is centered around objects that encapsulate both data and the functions that operate on them.**

There are two common programming methods in practice today: procedural programming and object-oriented programming (OOP). So far you have been creating procedural programs.

In a procedural program data is typically stored in a collection of variables and there is a set of functions that perform operations on the data. The data and the functions are separate entities.

Usually the variables are passed to the functions that perform the desired operations. As you might imagine, the focus of procedural programming is on creating the functions, or procedures, that operate on the program's data.

Procedural programming has worked well for software developers for many years. However, as programs become larger and more complex, the separation of a program's data and the code that operates on the data can lead to problems. For example, the data in a procedural program is stored in variables, as well as in more complex structures that are created from variables. The procedures that operate on the data must be designed with those variables and data structures in mind. But, what happens if the format of the data is altered? Quite often, a program's specifications change, resulting in redesigned data structures. When the structure of the data changes, the code that operates on the data must also be changed to accept the new format. Finding all the code that needs changing results in additional work for programmers and an opportunity for bugs to be introduced into the code.

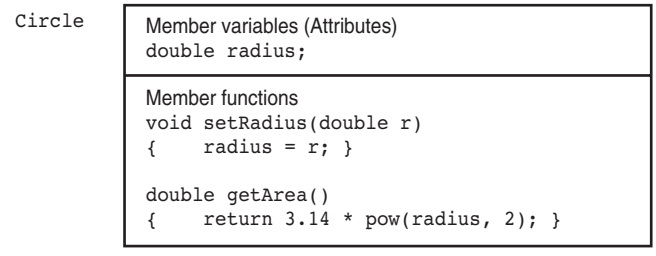
This has helped influence the shift from procedural programming to object-oriented programming. Whereas procedural programming is centered on creating procedures, OOP is centered on creating objects. An *object* is a software entity that combines both data and procedures in a single unit. An object's data items, also referred to as its *attributes*, are stored in *member variables*. The procedures that an object performs are called its *member functions*. This bundling of an object's data and procedures together is called *encapsulation*.



NOTE: In some object-oriented programming languages, the procedures that an object performs are called *methods*.

Figure 7-1 shows a representation of what a `Circle` object we create might look like. It has just one member variable to hold data and two member functions. The `Circle` object's data item is its radius. Its first function sets the radius, and its second function calculates and returns the area.

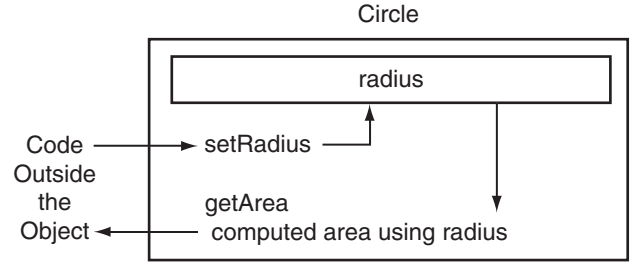
Figure 7-1



The member variable and the member functions are all members of the `Circle` object. They are bound together in a single unit. When an operation is to be performed, such as calculating the area of the circle, a message is passed to the object telling it to perform the `getArea` function. Because `getArea` is a member of the `Circle` object, it automatically has access to the object's member variables. Therefore, there is no need to pass `radius` to the `getArea` function.

Not only do objects encapsulate associated data and functions, they also permit data hiding. *Data hiding* refers to an object's ability to hide its data from code outside the object. Only the object's member functions can directly access and make changes to the object's data. An object typically hides its data, but allows outside code to access it through some of its member functions. As Figure 7-2 illustrates, the object's member functions provide programming statements outside the object with a way to indirectly access the object's data.

Figure 7-2



When an object's internal data is hidden from outside code, and access to that data is restricted to the object's member functions, the data is protected from accidental corruption. In addition, the programming code outside the object does not need to know about the format or internal structure of the object's data. The code only needs to interact with the object's functions. When a programmer changes the structure of an object's internal data, the object's member functions are also modified so they will still properly operate on the data. These changes, however, are hidden from code outside the object. The way in which outside code interacts with the member functions does not change.

Earlier we used the automobile as an example of an abstract object that can be used without having to understand the details of how it works. It has a rather simple interface that consists of an ignition switch, steering wheel, gas pedal, brake pedal, and a gear selector. (Vehicles with manual transmissions also provide a clutch pedal). If you want to drive an automobile, you only have to learn to operate these elements of its interface. To start the motor, you simply turn the key in the ignition switch. What happens internally is irrelevant to the driver. If you want to steer the auto to the left, you rotate the steering wheel left. The movements of all the linkages connecting the steering wheel to the front tires occur without your awareness. If the manufacturer redesigns the vehicle to perform one of the behind-the-scenes operations differently, the driver does not need to learn a new interface.

Because automobiles have simple user interfaces, they can be driven by people who have no mechanical knowledge. This is good for the makers of automobiles because it means more people are likely to become customers. It's good for the users of automobiles because they can learn just a few simple procedures and operate almost any vehicle.

These are also valid concerns in software development. A program is rarely written by only one person. Even the small programs you have created so far weren't written entirely by you. If you incorporated C++ library functions, or objects like `cin` and `cout`, you used code written by someone else. In the world of professional software development, programmers commonly work in teams, buy and sell their code, and collaborate on projects. With OOP, programmers can create objects with powerful engines tucked away "under the hood," but simple interfaces that safeguard the object's algorithms and data.

7.3 Introduction to Classes

CONCEPT: In C++, the class is the construct primarily used to create objects.

A *class* is a programmer-defined data type that describes what an object of the class will look like when it is created. It consists of a set of variables and a set of functions. Here is the general format of a class declaration.

```
class ClassName          // Class declaration begins with
{                        // the key word class and a name.

   Declarations for class member variables
   and member functions go here.

};                       // Notice the required semicolon.
```



VideoNote
Creating a Class

We will learn how to implement a class by building one step by step. Our example will be the simple `Circle` class depicted in Figure 7-1. The first step is to determine what member variables and member functions the class should have. In this case we have determined, as already described, that the class needs a member variable to hold the circle's radius and two member functions: `setRadius` and `getArea`.



NOTE: This information, along with other design information, is sometimes expressed using a modeling tool know as the Unified Modeling Language, or UML. Appendix D on the student CD (Using UML in Class Design) shows how to do this.

Once the class has been designed, the next step is to write the class declaration. This tells the compiler what the class includes. Here is the declaration for our `Circle` class. Notice that the class name begins with a capital letter. Although this is not strictly required, it is conventional to always begin class names with an uppercase letter.

```
class Circle
{  private:
        double radius;

   public:
      void setRadius(double r)
      {   radius = r; }

      double getArea()
      {   return 3.14 * pow(radius, 2);  }
};
```

Access Specifiers

The class declaration looks very much like Figure 7-1 with the addition of two key words, `private` and `public`. These are called *access specifiers*, because they designate who can access various members of the class. Notice that each access specifier is followed by a colon. A public member variable can be accessed by functions outside the class and a public member function can be called by functions outside the class. A private member variable, on the other hand, can only be accessed by a function that is a member of the same class and a private member function can only be called by other functions that are members of the class. If we had omitted the words `public` and `private` altogether, everything would have defaulted to being private. This would not have been very useful because then, except in special circumstances, no functions outside the class could ever use the class.

In our `Circle` class, the member variable `radius` is declared to be private and the member functions are declared to be public. This is common. Member data is usually made private to safeguard it. Public functions are then created to allow carefully controlled access to this data from outside the class. For now, all our class member variables will be declared as private and all our member functions will be declared as public. Later you will see cases where private member functions are used.



NOTE: If a program statement outside a class attempts to access a private member, a compiler error will result. Later you will learn how outside functions may be given special permission to access private class members.

Placement of `private` and `public` Members

It does not matter whether we list the private or public members first. In fact, it is not even required that all members of the same access specification be declared together. Both examples below are legal declarations of the `Circle` class.

```
class Circle
{public:
   void setRadius(double r)
   { radius = r; }

   double getArea()
   { return 3.14 * pow(radius, 2); }

private:
   double radius;
};
```

```
class Circle
{  public:
     void setRadius(double r)
     { radius = r; }

   private:
      double radius;

   public:
      double getArea()
      { return 3.14 * pow
      (radius, 2); }
};
```

However, most programmers consider it more orderly and preferable to separate private and public members. In this text we follow the standard practice of listing private members together first, followed by the public members, as shown in the initial `Circle` declaration.

7.4 Introduction to Objects

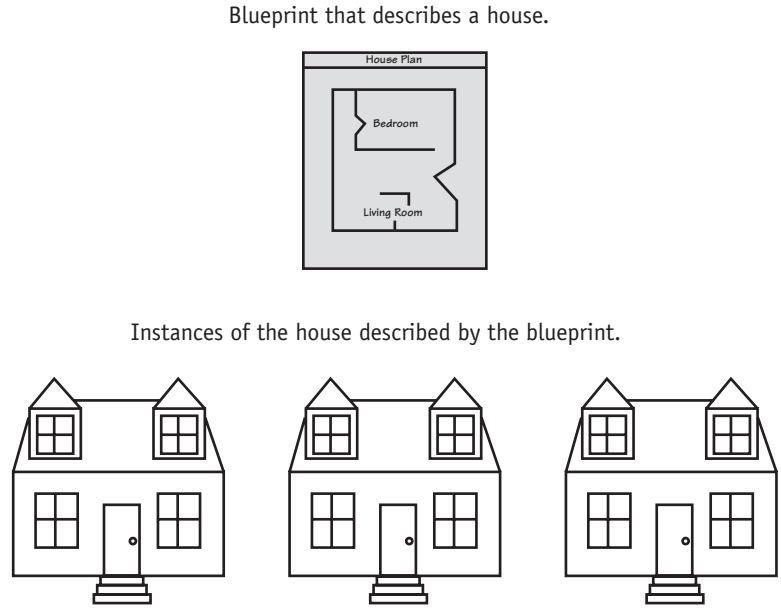
CONCEPT: **Objects are instances of a class. They are created with a definition statement after the class has been declared.**



VideoNote
Creating and Using Class Objects

A class declaration is similar to the blueprint for a house. The blueprint itself is not a house, but is a detailed description of a house. When we use the blueprint to build an actual house, we could say we are constructing an instance of the house described by the blueprint. If we wish, we can construct several identical houses from the same blueprint. Each house is a separate instance of the house described by the blueprint. This idea is illustrated in Figure 7-3.

Figure 7-3



A class declaration serves a similar purpose. It describes what the objects created from the class will look like when they are constructed. Each object created from it is called an *instance* of the class.

Class objects are created with simple definition statements, just like variables. For example, the following statement defines `circle1` and `circle2` to be two objects of the `Circle` class:

```
Circle circle1,
       circle2;
```

Defining a class object is called the *instantiation* of a class.

The objects `circle1` and `circle2` are two distinct instances of the `Circle` class, with different memory assigned to hold the values stored in their member variables.

Accessing an Object's Members

Public members of a class object are accessed with the dot operator. For example, the following statements call the `setRadius` function of `circle1` and `circle2`.

```
circle1.setRadius(1);    // This sets circle1's radius to 1
circle2.setRadius(2.5);  // This sets circle2's radius to 2.5
```

Now that the radii have been set, we can call the `getArea` member function to return the area of the `Circle` objects:

```
cout << "The area of circle1 is " << circle1.getArea() << endl;
cout << "The area of circle2 is " << circle2.getArea() << endl;
```

Program 7-1 is a complete program that demonstrates the `Circle` class. Notice that the statements to create and use `Circle` objects are in `main`, not in the class declaration.

Program 7-1

```
1  // This program demonstrates a simple class.
2  #include <iostream>
3  #include <cmath>
4  using namespace std;
5
6  // Circle class declaration
7  class Circle
8  {  private:
9        double radius;
10
11    public:
12       void setRadius(double r)
13       {  radius = r; }
14
15       double getArea()
16       {  return 3.14 * pow(radius, 2); }
17 };
18
19 int main()
20 {
21    // Define 2 Circle objects
22    Circle circle1,
23           circle2;
24
25    // Call the setRadius function for each circle
26    circle1.setRadius(1);    // This sets circle1's radius to 1
27    circle2.setRadius(2.5);  // This sets circle2's radius to 2.5
28
29    // Call the getArea function for each circle and
30    // display the returned result
31    cout << "The area of circle1 is " << circle1.getArea() << endl;
32    cout << "The area of circle2 is " << circle2.getArea() << endl;
33
34    return 0;
35 }
```

Program Output

```
The area of circle1 is 3.14
The area of circle2 is 19.625
```

Notice in lines 13 and 16 of Program 7-1 how the class member functions `setRadius` and `getArea` use the member variable `radius`. They do not need to use the dot operator to reference it because member functions of a class can access member variables of the same class like regular variables, without any extra notation. Notice also that the class member function `getArea` only uses, but does not modify, the member variable `radius`. A function like this, that uses the value of a class variable but does not change it, is known as an *accessor*. The function `setRadius`, on the other hand, modifies the contents of `radius`. A member function like this, which stores a value in a member variable or changes its value, is known as a *mutator*. Some programmers refer to mutators as *set functions* or *setter functions* because they set the value of a class variable, and refer to accessors as *get functions* or *getter functions* because they just retrieve or use the value.

7.5 Defining Member Functions

CONCEPT: Class member functions can be defined either inside or outside the class declaration.

Class member functions are defined similarly to regular functions. Except for a few special cases we will look at later, they have a function header that includes a return type (which may be `void`), a function name, and a parameter list (which may possibly be empty). The statements that carry out the actions of the function are contained within a pair of braces that follow the function header.

So far, all the class functions we have written have been defined within the class declaration itself. When a class function is defined in this location, it is called an *inline function*. Inline functions provide a convenient way to contain function information within a class declaration, but they can only be used when a function body is very short, usually a single line. When a function body is longer, a prototype for the function should appear in the class declaration, instead of the function definition itself. The function definition is then placed outside the class declaration, either following it or in a separate file.

Even though the two functions in our `Circle` class are short enough to be written as inline functions, we will rewrite them as regular functions, defined outside the class declaration, to illustrate how this is done. Inside the class declaration the functions would be replaced by the following prototypes:

```
void setRadius(double);
double getArea();
```

Following the class declaration we would place a *function implementation* section containing the following function definitions:

```
void Circle::setRadius(double r)
{   radius = r;
}

double Circle::getArea()
{   return 3.14 * pow(radius, 2);
}
```

Notice that these look like ordinary functions except they contain the class name and a double colon (`::`) between the function return type and function name. The `::` symbol is called the *scope resolution operator*. It is needed to indicate that these are class member functions and to tell the compiler which class they belong to.



WARNING! The class name and scope resolution operator are an extension of the function name. When a function is defined outside the class declaration, these must be present and must be located immediately before the function name in the function header.

Here are some additional examples to illustrate how the scope resolution is used when a class function is defined outside the class declaration.

```
double getArea()           // Wrong! The class name and scope
                           // resolution operator are missing.

Circle::double getArea()   // Wrong! The class name and scope
                           // resolution operator are misplaced.

double Circle::getArea()   // Correct!
```

Program 7-2 revises Program 7-1 to define the class member functions outside the class.

Program 7-2

```
1  // This program demonstrates a simple class with member functions
2  // defined outside the class declaration.
3  #include <iostream>
4  #include <cmath>
5  using namespace std;
6
7  // Circle class declaration
8  class Circle
9  {  private:
10       double radius;            // This is a member variable.
11
12    public:
13       void  setRadius(double);  // These are just prototypes
14       double getArea();         // for the member functions.
15 };
16
17 // The member function implementation section follows. It contains the
18 // actual function definitions for the Circle class member functions.
19
20 /*************************************************************
21  *                      Circle::setRadius                    *
22  *  This function copies the argument passed into the parameter to *
23  *  the private member variable radius.                      *
24  *************************************************************/
```

(program continues)

Program 7-2 *(continued)*

```
25 void Circle::setRadius(double r)
26 {  radius = r;
27 }
28
29 /*****************************************************************
30  *                        Circle::getArea                        *
31  * This function calculates and returns the Circle object's area. *
32  * It does not need any parameters because it already has access  *
33  * to the member variable radius.                                 *
34  *****************************************************************/
35 double Circle::getArea()
36 {  return 3.14 * pow(radius, 2);
37 }
38
39 /*****************************************************************
40  *                             main                              *
41  *****************************************************************/
42 int main()
43 {
44    Circle circle1,          // Define 2 Circle objects
45           circle2;
46
47    circle1.setRadius(1);    // This sets circle1's radius to 1
48    circle2.setRadius(2.5);  // This sets circle2's radius to 2.5
49
50                             // Get and display each circle's area
51    cout << "The area of circle1 is " << circle1.getArea() << endl;
52    cout << "The area of circle2 is " << circle2.getArea() << endl;
53
54    return 0;
55 }
```

Program Output is the same as for Program 7-1.

Naming Conventions for Class Member Functions

Program 7-3 provides another example using classes and objects. It declares and implements a `Rectangle` class that has two private member variables and five public member functions. Notice that the names of the member functions in Program 7-3 all begin with the word *set* or the word *get*. Functions `setLength` and `setWidth` are mutator, or set, functions. It is common to name a mutator with the word *set* followed by the name of the member variable whose value it is setting. As you would expect, the `setLength` function sets the value of the `length` member variable and the `setWidth` function sets the value of the `width` member variable.

Member functions `getLength` and `getWidth` are accessor, or get, functions. It is common to name an accessor with the word *get* followed by the name of the member variable whose value it is getting. Function `getLength` returns the value stored in the `length` member variable, while `getWidth` returns the value stored in the `width` member variable. The member function `getArea` also has a name that begins with *get* because it gets and returns the area, even though it calculates this value rather than retrieving it from a member variable.

Program 7-3

```
1  // This program implements a Rectangle class.
2  #include <iostream>
3  using namespace std;
4
5  // Rectangle class declaration
6  class Rectangle
7  {
8     private:
9        double length;
10       double width;
11    public:
12       void setLength(double);
13       void setWidth(double);
14       double getLength();
15       double getWidth();
16       double getArea();
17 };
18
19 // Member function implementation section
20
21 /*************************************************************
22  *                     Rectangle::setLength                  *
23  * This function sets the value of the member variable length.  *
24  * If the argument passed to the function is zero or greater, it is *
25  * copied into length. If it is negative, 1.0 is assigned to length.*
26  *************************************************************/
27 void Rectangle::setLength(double len)
28 {
29    if (len >= 0)
30       length = len;
31    else
32    {   length = 1.0;
33        cout << "Invalid length. Using a default value of 1.\n";
34    }
35 }
36
37 /*************************************************************
38  *                      Rectangle::setWidth                  *
39  * This function sets the value of the member variable width.   *
40  * If the argument passed to the function is zero or greater, it is *
41  * copied into width. If it is negative, 1.0 is assigned to width.  *
42  *************************************************************/
43 void Rectangle::setWidth(double w)
44 {
45    if (w >= 0)
46       width = w;
47    else
48    {   width = 1.0;
49        cout << "Invalid width. Using a default value of 1.\n";
50    }
51 }
52
```

(program continues)

Program 7-3 *(continued)*

```
53 /*************************************************************
54  *                      Rectangle::getLength                 *
55  * This function returns the value in member variable length. *
56  *************************************************************/
57 double Rectangle::getLength()
58 {
59    return length;
60 }
61
62 /*************************************************************
63  *                      Rectangle::getWidth                  *
64  * This function returns the value in member variable width.  *
65  *************************************************************/
66 double Rectangle::getWidth()
67 {
68    return width;
69 }
70
71 /*************************************************************************
72  *                          Rectangle::getArea                           *
73  * This function calculates and returns the area of the rectangle. *
74  *************************************************************************/
75 double Rectangle::getArea()
76 {
77    return length * width;
78 }
79
80 /*************************************************************
81  *                              main                         *
82  *************************************************************/
83 int main()
84 {
85    Rectangle box;         // Declare a Rectangle object
86    double boxLength, boxWidth;
87
88    // Get box length and width
89    cout << "This program will calculate the area of a rectangle.\n";
90    cout << "What is the length? ";
91    cin  >> boxLength;
92    cout << "What is the width? ";
93    cin  >> boxWidth;
94
95    // Call member functions to set box dimensions
96    box.setLength(boxLength);
97    box.setWidth(boxWidth);
98
99    // Call member functions to get box information to display
100   cout << "\nHere is the rectangle's data:\n";
101   cout << "Length: " << box.getLength() << endl;
102   cout << "Width : " << box.getWidth()  << endl;
103   cout << "Area  : " << box.getArea()   << endl;
104   return 0;
105 }
```

(program continues)

Program 7-3 *(continued)*

Program Output with Example Input Shown in Bold

```
This program will calculate the area of a rectangle.
What is the length? 3[Enter]
What is the width? -1[Enter]
Invalid width. Using a default value of 1.

Here is the rectangle's data:
Length: 3
Width : 1
Area  : 3
```

Program Output with Different Example Input Shown in Bold

```
This program will calculate the area of a rectangle.
What is the length? 10.1[Enter]
What is the width? 5[Enter]

Here is the rectangle's data:
Length: 10.1
Width : 5
Area  : 50.5
```

We mentioned earlier that when designing a class it is common practice to make all member variables private and to provide public set and get functions for accessing those variables. This safeguards the data. Functions outside the class can only access the member data through calls to the public member functions and these functions can be written to prevent the data from being corrupted or modified in a way that might adversely affect the behavior of an object of the class. Notice in Program 7-3 how the two set functions are written to filter out invalid data. Rather than allow an invalid value to be stored in a member variable, they use a default value if the data passed to them is not acceptable.

Avoiding Stale Data

In the `Rectangle` class, the `getLength` and `getWidth` member functions return the values stored in the `length` and `width` member variables, but the `getArea` member function returns the result of a calculation. You might wonder why the area of the rectangle is not also stored in a member variable. The area is not stored because it could potentially become stale. When the value of an item is dependent on other data and that item is not updated when the other data is changed, it is said that the item has become *stale*. If the area of the rectangle were stored in a member variable, its value would become incorrect as soon as either the `length` or `width` member variables changed.

When designing a class, you should normally not use a member variable to store a calculated value that could potentially become stale. Instead, provide a member function that calculates the value, using the most current data, and then returns the result of the calculation.

More on Inline Functions

When designing a class, you will need to decide which member functions to write as inline functions within the class declaration and which ones to define outside the class. Inline functions are handled completely differently by the compiler than regular functions are. An understanding of this difference may help you decide which to use when.

A lot goes on behind the scenes each time a regular function is called. A number of special items, such as the address to return to when the function has finished executing and the values of the function arguments, must be stored in a section of memory called the *stack*. In addition, local variables are created and a location is reserved to hold the function's return value. All this overhead, which sets the stage for a function call, takes CPU time. Although the time needed is small, it can add up if a function is called many times, as in a loop.

An inline function, on the other hand, is not called in the conventional sense at all. Instead, in a process known as *inline expansion*, the compiler replaces every call to the function with the actual code of the function itself. This means that if the function is called from multiple places in the program, the entire body of its code will be inserted multiple times, increasing the size of the program. This is why only a function with very few lines of code should be written as an inline function. In fact, if the function is too large to make the inline expansion practical, the compiler will ignore the request to handle the function this way. However, when a member function is small, it can improve performance to write it as an inline function because of the reduced overhead of not having to make actual function calls.*



Checkpoint

7.1 Which of the following shows the correct use of the scope resolution operator in a member function definition?

A) `InvItem::void setOnHand(int units)`
B) `void InvItem::setOnHand(int units)`

7.2 An object's private member variables can be accessed from outside the object by

A) public member functions
B) any function
C) the dot operator
D) the scope resolution operator

7.3 Assuming that `soap` is an instance of the `Inventory` class, which of the following is a valid call to the `setOnHand` member function?

A) `setOnHand(20);`
B) `soap::setOnHand(20);`
C) `soap.setOnHand(20);`
D) `Inventory.setOnHand(20);`

7.4 Complete the following code skeleton to declare a class called `Date`. The class should contain member variables and functions to store and retrieve the month, day, and year components of a date.

```
class Date
{  private:
   public:
}
```

* On systems that use paging, however, inline functions can decrease performance because of the additional page faults caused by the increased program size.

7.6 Constructors

CONCEPT: **A constructor is a member function that is automatically called when a class object is created.**

A *constructor* is a special public member function that is automatically called to *construct* a class object when it is created. If the programmer does not write a constructor, C++ automatically provides one. You never see it, but it runs silently in the background each time your program defines an object. Often, however, programmers write their own constructor when they create a class. If they do this, in addition to constructing each newly created object of the class, it will execute whatever code the programmer has included in it. Most often programmers use a constructor to initialize an object's member variables. However, it can do anything a normal function can do.



NOTE: Appendix D on the student CD shows how to denote a constructor in UML.

A constructor looks like a regular function except that its name must be the same as the name of the class it is a part of. This is how the compiler knows that a particular member function is a constructor. Also, the constructor must have no return type. Program 7-4 includes a class called `Demo` with a constructor that does nothing except print a message. It was written this way to demonstrate when the constructor executes. Because the `Demo` object is created between two `cout` statements, the constructor will print its message between the output lines produced by those two statements.

Program 7-4

```
1  // This program demonstrates when a constructor executes.
2  #include <iostream>
3  using namespace std;
4
5  class Demo
6  {
7   public:
8     Demo()           // Constructor
9     {
10       cout << "Now the constructor is running.\n";
11    }
12 };
13
14 int main()
15 {
16    cout << "This is displayed before the object is created.\n";
17
18    Demo demoObj;    // Define a Demo object
19
20    cout << "This is displayed after the object is created.\n";
21    return 0;
22 }
```

(program continues)

Program 7-4 *(continued)*

Program Output

```
This is displayed before the object is created.
Now the constructor is running.
This is displayed after the object is created.
```

In Program 7-4 the constructor was defined as an inline function inside the class declaration. As with all class member functions, a constructor can be written either as an inline function (if it is short enough), or can have just its prototype placed in the class declaration and be defined outside the class. In this case, as you recall, its name must be preceded by the name of the class the function belongs to and the scope resolution operator. Because the name of the constructor function is the *same* as the class name, the same name appears twice. Here is how the function header for the `Demo` constructor would look if it had been defined outside the class.

```
Demo::Demo()            // Constructor
{
    cout << "Now the constructor is running.\n";
}
```

Program 7-5 modifies Program 7-2 to include a constructor that initializes an object's member data. The constructor is defined outside of the class.

Program 7-5

```
1  // This program uses a constructor to initialize a member variable.
2  #include <iostream>
3  #include <cmath>
4  using namespace std;
5
6  // Circle class declaration
7  class Circle
8  {  private:
9        double radius;
10
11    public:              // Member function prototypes
12       Circle();
13       void setRadius(double);
14       double getArea();
15 };
16
17 // Circle member function implementation section
18
19 /*****************************************
20  *              Circle::Circle           *
21  * This is the constructor. It initializes  *
22  * the member variable radius to 1.         *
23  *****************************************/
24 Circle::Circle()
25 {  radius = 1;
26 }
27
```

(program continues)

Program 7-5 *(continued)*

```
28 /****************************************
29  *            Circle::setRadius         *
30  * This function validates the radius value   *
31  * passed to it before assigning it to member *
32  * variable radius.                      *
33  ****************************************/
34 void Circle::setRadius(double r)
35 {  if (r >= 0)
36       radius = r;
37    // else leave it set to its previous value
38 }
39
40 /****************************************
41  *              Circle::getArea          *
42  * This function calculates and returns the   *
43  * Circle object's area. It does not need any *
44  * parameters because it can directly access  *
45  * the member variable radius.           *
46  ****************************************/
47 double Circle::getArea()
48 {  return 3.14 * pow(radius, 2);
49 }
50
51 /*************************************
52  *              main                 *
53  * The main function creates and uses  *
54  * 2 Circle objects.                  *
55  *************************************/
56 int main()
57 {
58    // Define a Circle object. Because the setRadius function
59    // is never called for it, it will keep the value set
60    // by the constructor.
61    Circle circle1;
62
63    // Define a second Circle object and set its radius to 2.5
64    Circle circle2;
65    circle2.setRadius(2.5);
66
67    // Get and display each circle's area
68    cout << "The area of circle1 is " << circle1.getArea() << endl;
69    cout << "The area of circle2 is " << circle2.getArea() << endl;
70
71    return 0;
72 }
```

Program Output

```
The area of circle1 is 3.14
The area of circle2 is 19.625
```

Overloading Constructors

Recall from Chapter 6 that when two or more functions share the same name, the function name is said to be overloaded. Multiple functions with the same name may exist in a C++ program, as long as their parameter lists are different.

Any class member function may be overloaded, including the constructor. One constructor might take an integer argument, for example, while another constructor takes a `double`. There could even be a third constructor taking two integers. As long as each constructor has a different list of parameters, the compiler can tell them apart.

Program 7-6 declares and uses a class named `Sale`, which has two constructors. The first has a parameter that accepts a sales tax rate. The second, which is for tax-exempt sales, has no parameters. It sets the tax rate to 0. A constructor like this, which has no parameters, is called a *default constructor.*

Program 7-6

```
1  // This program demonstrates the use of overloaded constructors.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  // Sale class declaration
7  class Sale
8  {
9   private:
10    double taxRate;
11
12   public:
13      Sale(double rate)        // Constructor with 1 parameter
14      {  taxRate = rate;       // handles taxable sales
15      }
16
17      Sale()                   // Default constructor
18      {  taxRate = 0;          // handles tax-exempt sales
19      }
20
21      double calcSaleTotal(double cost)
22      {  double total = cost + cost*taxRate;
23         return total;
24      }
25 };
26
27 int main()
28 {
29    Sale cashier1(.06);       // Define a Sale object with 6% sales tax
30    Sale cashier2;            // Define a tax-exempt Sale object
31
32    // Format the output
33    cout << fixed << showpoint << setprecision(2);
34
```

(program continues)

Program 7-6 *(continued)*

```
35    // Get and display the total sale price for two $24.95 sales
36    cout << "\nWith a 0.06 sales tax rate, the total\n";
37    cout << "of the $24.95 sale is $";
38    cout << cashier1.calcSaleTotal(24.95) << endl;
39
40    cout << "\nOn a tax-exempt purchase, the total\n";
41    cout << "of the $24.95 sale is, of course, $";
42    cout << cashier2.calcSaleTotal(24.95) << endl;
43    return 0;
44 }
```

Program Output

```
With a 0.06 sales tax rate, the total
of the $24.95 sale is $26.45

On a tax-exempt purchase, the total
of the $24.95 sale is, of course, $24.95
```

Notice on lines 29 and 30 of Program 7-6 how the two `Sale` objects are defined.

```
Sale cashier1(.06);
Sale cashier2;
```

There is a pair of parentheses after the name `cashier1` to hold the value being sent to the 1-parameter constructor. However, there are no parentheses after the name `cashier2`, which sends no arguments. In C++ when an object is defined using the default constructor, instead of passing arguments, there must *not* be any parentheses.

```
Sale cashier2;          // Correct
Sale cashier2();        // Wrong!
```

Default Constructors

The `Sale` class needed a default constructor to handle tax-free sales. Other classes may appear not to need one—for example, if objects created from them are always expected to pass arguments to the constructors. Yet, any time you design a class that will have constructors, it is considered good programming practice to include a default constructor. If you do not have one, and the program inadvertently tries to create an object without passing any arguments, it will not compile. This is because there must be a constructor to create an object. In order to create an object that passes no arguments, there must be a constructor that expects no arguments—a default constructor. If the programmer writes no constructors for a class, the compiler automatically creates a default constructor for the class. However, if the programmer writes any constructors at all, even ones that all have parameters, the compiler does not create a default constructor. So it is the responsibility of the programmer to do this.

A class may have many constructors, but can only have one default constructor. This is because if multiple functions have the same name, the compiler must be able to determine from their parameter lists which one is being called at any given time. It uses the number

and type of arguments passed to the function to determine which of the overloaded functions to invoke. Because there can be only one function with the class name that is able to accept no arguments, there can be only one default constructor.

Normally, as in the `Sale` class, default constructors have no parameters. However, it is possible to have a default constructor with parameters if all of its parameters have default values, so that it can be called with no arguments. It would be an error to create one constructor that accepts no arguments and another that has arguments but allows default values for all of them. This would create two "default" constructors. The following class declaration illegally does this.

```
class Sale                  // Illegal declaration!
{
  private:
    double taxRate;

  public:
    Sale()                  // Default constructor with no arguments
    {  taxRate = 0.05; }

    Sale(double r = 0.05) // Default constructor with a default argument
    {  taxRate = r; }

    double calcSaleTotal(double cost)
    {  double total = cost + cost * taxRate;
       return total;
};
```

As you can see, the first constructor has no parameters. The second constructor has one parameter, but it has a default argument. If an object is defined with no argument list, the compiler will not be able to resolve which constructor to execute.

7.7 Destructors

CONCEPT: A destructor is a member function that is automatically called when an object is destroyed.

Destructors are public member functions with the same name as the class, preceded by a tilde character (~). For example, the destructor for the `Rectangle` class would be named `~Rectangle`.

Destructors are automatically called when an object is destroyed. In the same way that constructors can be used to set things up when an object is created, destructors are used to perform shutdown procedures when the object goes out of existence.



NOTE: Appendix D on the student CD shows how to denote a destructor in UML.

Program 7-7 shows a simple class with a constructor and a destructor. It illustrates when each is called during the program's execution.

Program 7-7

```
1  // This program demonstrates a destructor.
2  #include <iostream>
3  using namespace std;
4
5  class Demo
6  {
7     public:
8        Demo();       // Constructor prototype
9       ~Demo();       // Destructor prototype
10 };
11
12 Demo::Demo()        // Constructor function definition
13 {  cout << "An object has just been defined, so the constructor"
14         << " is running.\n";
15 }
16
17 Demo::~Demo()       // Destructor function definition
18 {  cout << "Now the destructor is running.\n";
19 }
20
21 int main()
22 {
23    Demo demoObj;    // Declare a Demo object;
24
25    cout << "The object now exists, but is about to be destroyed.\n";
26    return 0;
27 }
```

Program Output

```
An object has just been defined, so the constructor is running.
The object now exists, but is about to be destroyed.
Now the destructor is running.
```

In addition to the fact that destructors are automatically called when an object is destroyed, the following points should be mentioned:

- Like constructors, destructors have no return type.
- Destructors cannot accept arguments, so they never have a parameter list.
- Because destructors can accept no arguments, there can only be one destructor.

Destructors are most useful when working with objects that are dynamically allocated. You will learn about this in Chapter 10.



Checkpoint

7.5 Briefly describe the purpose of a constructor.

7.6 Constructor functions have the same name as the
A) class
B) class instance
C) program
D) none of the above

7.7 A constructor that requires no arguments is called
A) a default constructor
B) an inline constructor
C) a null constructor
D) none of the above

7.8 Assume the following is a constructor:

```
ClassAct::ClassAct(int x)
{
   item = x;
}
```

Define a `ClassAct` object called `sally` that passes the value 25 to the constructor.

7.9 True or false: Like any C++ function, a constructor may be overloaded, providing each constructor has a unique parameter list.

7.10 True or false: A class may have a constructor with no parameter list, and an overloaded constructor whose parameters all take default arguments.

7.11 A destructor function name always starts with
A) a number
B) the tilde character (~)
C) a data type name
D) the name of the class

7.12 True or false: Just as a class can have multiple constructors, it can also have multiple destructors.

7.13 What will the following program display on the screen?

```
#include <iostream>
using namespace std;

class Tank
{
private:
   int gallons;
public:
   Tank()
      { gallons = 50; }
   Tank(int gal)
      { gallons = gal; }
   int getGallons()
      { return gallons; }
};

int main()
{  Tank storage1, storage2, storage3(20);

   cout << storage1.getGallons() << endl;
   cout << storage2.getGallons() << endl;
   cout << storage3.getGallons() << endl;
   return 0;
}
```

7.14 What will the following program display on the screen?

```
#include <iostream>
using namespace std;

class Package
{
private:
   int value;
public:
   Package()
      { value = 7; cout << value << endl; }
   Package(int v)
      { value = v; cout << value << endl; }
   ~Package()
      { cout << "goodbye" << endl; }
};

int main()
{  Package obj1(4);
   Package obj2;
   return 0;
}
```

7.8 Private Member Functions

CONCEPT: Private member functions may only be called from a function that is a member of the same class.

Until now all of the class member functions you have seen have been public functions. This means they can be called by code in programs outside the class. Often, however, a class needs functions for internal processing that should not be called by code outside the class. Such functions should be made private.



NOTE: Appendix D on the student CD shows how to denote private and public members in UML.

Program 7-8 shows an example of a class with a private function. The `SimpleStat` class is designed to find and report information, such as the average and the largest number, from a set of non-negative integers sent to it. However, once a number has been received and added to a running total, it is not kept. So the class cannot later determine which number was the biggest. It must do this by determining, as it receives each number, if it is bigger than any number it previously received. The private `isNewLargest` function does this.

Program 7-8

```
// This program uses a private Boolean function to determine if
// a new value sent to it is the largest value received so far.
#include <iostream>
using namespace std;

class SimpleStat
{
 private:
   int largest;         // The largest number received so far
   int sum;             // The sum of the numbers received
   int count;           // How many numbers have been received

   bool isNewLargest(int);
 public:

   SimpleStat();        // Default constructor
   bool addNumber(int);
   double getAverage();

   int getLargest()
   {  return largest; }

   int getCount()
   {  return count; }
};

// SimpleStat Class Implementation Code

/************************************
 *  SimpleStat Default Constructor   *
 ************************************/
SimpleStat::SimpleStat()
{
   largest = sum = count = 0;
}

/************************************
 *      SimpleStat::addNumber       *
 ************************************/
bool SimpleStat::addNumber(int num)
{  bool goodNum = true;
   if (num >= 0)            // If num is valid
   {
      sum += num;           // Add it to the sum
      count++;              // Count it
      if(isNewLargest(num)) // Find out if it is
         largest = num;     // the new largest
   }
   else                     // num is invalid
      goodNum = false;

   return goodNum;
}

```

(program continues)

Program 7-8 *(continued)*

```
55 /************************************
56  *     SimpleStat::isNewLargest      *
57  ************************************/
58 bool SimpleStat::isNewLargest(int num)
59 {
60    if (num > largest)
61       return true;
62    else
63       return false;
64 }
65
66 /************************************
67  *      SimpleStat::getAverage       *
68  ************************************/
69 double SimpleStat::getAverage()
70 {
71    if (count > 0)
72       return static_cast<double>(sum) / count;
73    else
74       return 0;
75 }
76
77 // Client Program
78
79 /************************************
80  *                main               *
81  ************************************/
82 int main()
83 {
84    int num;
85    SimpleStat statHelper;
86
87    cout << "Please enter the set of non-negative integer \n";
88    cout << "values you want to average. Separate them with \n";
89    cout << "spaces and enter -1 after the last value. \n\n";
90
91    cin >> num;
92    while (num >= 0)
93    {
94       statHelper.addNumber(num);
95       cin >> num;
96    }
97    cout << "\nYou entered " << statHelper.getCount() << " values. \n";
98    cout << "The largest value was " << statHelper.getLargest() << endl;
99    cout << "The average value was " << statHelper.getAverage() << endl;
100
101   return 0;
102 }
```

(program continues)

Program 7-8 *(continued)*

Program Output with Example Input Shown in Bold

```
Please enter the set of non-negative integer
values you want to average. Separate them with
spaces and enter -1 after the last value.

7 6 8 8 9 7 7 8 9 7 -1[Enter]

You entered 10 values.
The largest value was 9
The average value was 7.6
```

In Program 7-8 the private function `isNewLargest` was written to create a more modular class with code that is easy to follow. The program could have been written without this function. However, in that case, the `addNumber` function would have to handle the additional work of comparing the new value with `largest` itself. In later chapters you will encounter many examples where the use of private functions is essential.

7.9 Passing Objects to Functions

CONCEPT: **Class objects may be passed as arguments to functions.**

In Chapter 6 you learned how to use variables as function arguments. Class objects can also be passed as arguments to functions. For example, the following function has a parameter that receives a `Rectangle` object.

```
void displayRectangle(Rectangle r)
{
   cout << "Length = " << r.getLength() << endl;
   cout << "Width  = " << r.getWidth()  << endl;
   cout << "Area   = " << r.getArea()   << endl;
}
```

The following lines of code create a `Rectangle` object with length 15 and width 10, and then pass it to the `displayRectangle` function.

```
Rectangle box(15, 10);
displayRectangle(box);
```

Assuming the `Rectangle` class includes the member functions used in this example, the `displayRectangle` function will output the following information:

```
Length = 15
Width  = 10
Area   = 150
```

As with regular variables, objects can be passed to functions by value or by reference. In the `Rectangle` example, `box` is passed to the `displayRectangle` function by value. This

means that `displayRectangle` receives a copy of `box`. If `displayRectangle` called any `Rectangle` class mutator functions, they would only change the copy of `box`, not the original. If a function needs to store or change data in an object's member variables, the object must be passed to it by reference.

Program 7-9 illustrates this. It has two functions that receive an `InventoryItem` object. The object is passed to `storeValues` by reference because this function needs to call a class mutator function that stores new values into the object. The object is passed to `showValues` by value because this function only needs to use accessor functions that retrieve and use values in the object's data members. Notice in Program 7-9 that the `InventoryItem` class declaration appears before the prototype for the `storeValues` and `showValues` functions. This is important. Because both functions use an `InventoryItem` object as a parameter, the compiler must know what `InventoryItem` is before it encounters any definitions for variables of that type. Otherwise an error will occur.

Program 7-9

```
1  // This program passes an object to a function. It passes it
2  // to one function by reference and to another by value.
3  #include <iostream>
4  #include <iomanip>
5  #include <string>
6  using namespace std;
7
8  class InventoryItem
9  {
10  private:
11    int partNum;          // Part number
12    string description;   // Item description
13    int onHand;           // Units on hand
14    double price;         // Unit price
15
16  public:
17
18    void storeInfo(int p, string d, int oH, double cost); // Prototype
19
20    int getPartNum()
21    {  return partNum; }
22
23    string getDescription()
24    {  return description; }
25
26    int getOnHand()
27    {  return onHand; }
28
29    double getPrice()
30    {  return price; }
31 };
32
```

(program continues)

Program 7-9 *(continued)*

```
// Implementation code for InventoryItem class function storeInfo
void InventoryItem::storeInfo(int p, string d, int oH, double cost)
{  partNum = p;
   description = d;
   onHand = oH;
   price = cost;
}

// Function prototypes for client program
void storeValues(InventoryItem&);  // Receives an object by reference
void showValues (InventoryItem);   // Receives an object by value

//**************** main *****************
int main()
{
   InventoryItem part;             // part is an InventoryItem object

   storeValues(part);
   showValues(part);
   return 0;
}

/*********************************************************
 *                        storeValues                    *
 * This function stores user input data in the members of *
 * an InventoryItem object passed to it by reference.     *
 * *******************************************************/
void storeValues(InventoryItem &item)
{
   int partNum;                // Local variables to hold user input
   string description;
   int qty;
   double price;

   // Get the data from the user
   cout << "Enter data for the new part number \n";
   cout << "Part number: ";
   cin  >> partNum;
   cout << "Description: ";
   cin.get();                  // Move past the '\n' left in the
                               // input buffer by the last input
   getline(cin, description);
   cout << "Quantity on hand: ";
   cin  >> qty;
   cout << "Unit price: ";
   cin  >> price;

   // Store the data in the InventoryItem object
   item.storeInfo(partNum, description, qty, price);
}

```

(program continues)

Program 7-9 *(continued)*

```
84 /*****************************************************
85  *                      showValues                   *
86  * This function displays the member data stored in the *
87  * InventoryItem object passed to it by value.       *
88  *****************************************************/
89 void showValues(InventoryItem item)
90 {
91    cout << fixed << showpoint << setprecision(2) << endl;;
92    cout << "Part Number  : "  << item.getPartNum() << endl;
93    cout << "Description  : "  << item.getDescription() << endl;
94    cout << "Units On Hand: "  << item.getOnHand() << endl;
95    cout << "Price        : $" << item.getPrice() << endl;
96 }
```

Program Output with Example Input Shown in Bold

```
Enter data for the new part number

Part number: 175[Enter]
Description: Hammer[Enter]
Quantity on hand: 12[Enter]
Unit price: 7.49[Enter]

Part Number  : 175
Description  : Hammer
Units On Hand: 12
Price        : $7.49
```

Constant Reference Parameters

In Program 7-9 `part`, the `InventoryItem` object, was passed by value to the `showValues` function. However, passing an object by value requires making a copy of all of the object's members. This can slow down a program's execution time, particularly if it has many members. When an object is passed by reference, on the other hand, no copy has to be made because the function has access to the original object. For this reason it is generally preferable to pass objects by reference.

There is a disadvantage to passing an object by reference, however. Because the function has access to the original object, it can call its mutator functions and alter its member data. This is why we normally do not pass variables by reference when we want to safeguard their contents. Luckily there is a solution. To protect an object when it is passed as an argument, without having to make a copy, it can be passed as a *constant reference*. This means that a reference to the original object is passed to the function, but it cannot call any mutator functions or change any of the object's member data. It can only call accessor functions that have themselves been designated as *constant functions*.

To declare a parameter to be a constant reference parameter, the key word `const` must be placed in the parameter list of both the function prototype and function header. Here is what the function prototype and header of the `showValues` function from Program 7-9 would look like, modified to use a constant reference parameter.

```
void showValues (const InventoryItem&)        // Function prototype
void showValues (const InventoryItem &item)   // Function header
```

Now the `showValues` function can only call `InventoryItem` functions that also have the key word `const` listed in their function prototype and header, like this:

```
double getPrice() const
```

If `showValues` tried to call any other `InventoryItem` functions, a compiler error would occur. Notice that when `showValues` is modified to have a constant reference parameter, only the function prototypes and headers are changed to include the word `const`. The body of the `showValues` function and the call to `showValues` do not change.

Returning an Object from a Function

Just as functions can be written to return an `int`, `string`, `double`, or other data type, they can also be designed to return an object. In this case the function normally creates a local instance of the class, sets its data members, and then returns it. Here is an example of how the `InventoryItem` object used in Program 7-9 could be created within the `storeValues` function and then returned to the calling function.

```
InventoryItem storeValues()
{
   InventoryItem tempItem;    // Local InventoryItem object
   int partNum;               // Local variables to hold user input
   string description;
   int qty;
   double price;


   // Code to get the data from the user goes here.


   tempItem.storeInfo(partNum, description, qty, price);
   return tempItem;
}
```

The main function could then create `part` like this:

```
InventoryItem part = storeValues();
```

Program 7-10 revises Program 7-9 to incorporate the techniques we have just discussed. The function previously named `storeValues` is renamed `createItem`, as it now creates an `InventoryItem` object and returns it to `main`. The `showValues` function now receives `part` as a constant reference, instead of having it passed by value, as before.

Program 7-10

```
1 // This program uses a constant reference parameter.
2 // It also illustrates how to return an object from a function.
3 #include <iostream>
4 #include <iomanip>
5 #include <string>
6 using namespace std;
7
```

(program continues)

Program 7-10 *(continued)*

```
class InventoryItem
{
 private:
   int partNum;               // Part number
   string description;        // Item description
   int onHand;                // Units on hand
   double price;              // Unit price

 public:

   void storeInfo(int p, string d, int oH, double cost); // Prototype

   int getPartNum() const     // The get functions have all been made
   {  return partNum; }       // const functions. This ensures they
                              // cannot alter any class member data.
   string getDescription() const
   {  return description; }

   int getOnHand() const
   {  return onHand; }

    double getPrice() const
   {  return price; }
};

// Implementation code for InventoryItem class function storeInfo
void InventoryItem::storeInfo(int p, string d, int oH, double cost)
{  partNum = p;
   description = d;
   onHand = oH;
   price = cost;
}

// Function prototypes for client program
InventoryItem createItem();              // Returns an InventoryItem object
void showValues (const InventoryItem&);  // Receives a reference to an
                                         // InventoryItem object

//*************** main *****************
int main()
{
   InventoryItem part = createItem();
   showValues(part);
   return 0;
}

/*********************************************************
 *                          createItem                   *
 * This function stores user input data in the members of a *
 * locally defined InventoryItem object, then returns it.   *
 *********************************************************/
```

(program continues)

Program 7-10 *(continued)*

```
59 InventoryItem createItem()
60 {
61    InventoryItem tempItem;   // Local InventoryItem object
62    int partNum;              // Local variables to hold user input
63    string description;
64    int qty;
65    double price;
66
67    // Get the data from the user
68    cout << "Enter data for the new part number \n";
69    cout << "Part number: ";
70    cin  >> partNum;
71    cout << "Description: ";
72    cin.get();                // Move past the '\n' left in the
73                              // input buffer by the last input
74    getline(cin, description);
75    cout << "Quantity on hand: ";
76    cin  >> qty;
77    cout << "Unit price: ";
78    cin  >> price;
79
80    // Store the data in the InventoryItem object and return it
81    tempItem.storeInfo(partNum, description, qty, price);
82    return tempItem;
83 }
84
85 /************************************************************
86  *                        showValues                        *
87  * This function displays the member data in the InventoryItem *
88  * object passed to it. Because it was passed as a constant    *
89  * reference, showValues accesses the original object, not a   *
90  * copy, but it can only call member functions declared to be  *
91  * const. This prevents it from calling any mutator functions. *
92  ************************************************************/
93 void showValues(const InventoryItem &item)
94 {
95    cout << fixed << showpoint << setprecision(2) << endl;;
96    cout << "Part Number  : "  << item.getPartNum() << endl;
97    cout << "Description  : "  << item.getDescription() << endl;
98    cout << "Units On Hand: "  << item.getOnHand() << endl;
99    cout << "Price        : $" << item.getPrice() << endl;
100 }
```

Program Output is the Same as for Program 7-9.



Checkpoint

7.15 A private class member function can be called by
- A) any other function
- B) only public functions in the same class
- C) only private functions in the same class
- D) any function in the same class

7.16 When an object is passed to a function, a copy of it is made if the object is
- A) passed by value
- B) passed by reference
- C) passed by constant reference
- D) any of the above

7.17 If a function receives an object as an argument and needs to change the object's member data, the object should be
- A) passed by value
- B) passed by reference
- C) passed by constant reference
- D) none of the above

7.18 True or false: Objects can be passed to functions, but they cannot be returned by functions.

7.19 True or false: When an object is passed to a function, but the function is not supposed to change it, it should be passed by value.

7.10 Object Composition

CONCEPT: It is possible for a class to have a member variable that is an instance of another class.

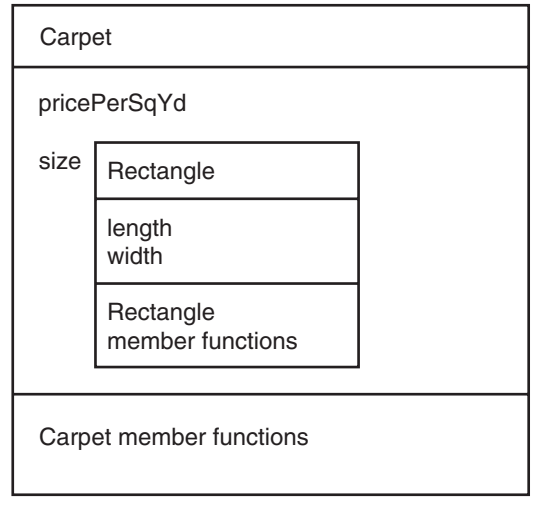
Sometimes it's helpful to nest an object of one class inside another class. For example, consider the following declarations:

```
class Rectangle
{
   private:
      double length;
      double width;
   public:
      void setLength(double);
      void setWidth(double);
      double getLength();
      double getWidth();
      double getArea();
};
```

```
class Carpet
{
   private:
      double pricePerSqYd;
      Rectangle size;           // size is an instance of
                                // the Rectangle class
   public:
      void setPricePerYd(double p);
      void setDimensions(double l, double w);
      double getTotalPrice();
};
```

Notice that the `Carpet` class has a member variable named `size`, which is an instance of the `Rectangle` class. The `Carpet` class can use this object to store the room dimensions and to compute the area for a carpet purchase. Figure 7-4 illustrates how the two classes are related.

Figure 7-4



When one class is nested inside another like this, it is called *object composition*.

Program 7-11 uses these two classes to create an application that computes carpet sale prices.

Program 7-11

```
// This program nests one class inside another. It has a class
// with a member variable that is an instance of another class.
#include <iostream>
using namespace std;

class Rectangle
{
   private:
      double length;
      double width;
   public:
      void setLength(double len)
      {  length = len; }

      void setWidth(double wid)
      {  width = wid; }

      double getLength()
      {  return length; }

      double getWidth()
      {  return width; }

      double getArea()
      {  return length * width; }
};

class Carpet
{
   private:
      double pricePerSqYd;
      Rectangle size;             // size is an instance of
                                  // the Rectangle class
   public:
      void setPricePerYd(double p)
      { pricePerSqYd = p; }

      void setDimensions(double len, double wid)
      {  size.setLength(len/3);   // Convert feet to yards
         size.setWidth (wid/3);
      }

      double getTotalPrice()
      { return (size.getArea() * pricePerSqYd); }
};

```

(program continues)

Program 7-11 *(continued)*

```
47 // ************** Client Program *****************
48 int main()
49 {
50    Carpet purchase;           // This variable is a Carpet object
51    double pricePerYd;
52    double length;
53    double width;
54
55 cout << "Room length in feet: ";
56    cin  >> length;
57    cout << "Room width in feet : ";
58    cin  >> width;
59    cout << "Carpet price per sq. yard: ";
60    cin  >> pricePerYd;
61
62    purchase.setDimensions(length, width);
63    purchase.setPricePerYd(pricePerYd);
64
65    cout << "\nThe total price of my new " << length << " x " << width
66         << " carpet is $" << purchase.getTotalPrice() << endl;
67
68    return 0;
69 }
```

Program Output with Example Input Shown in Bold

```
Room length in feet: 16.5[Enter]
Room width in feet : 12[Enter]
Carpet price per sq. yard: 22.49[Enter]

The total price of my new 16.5 x 12 carpet is $494.78
```

Let's take a closer look at Program 7-11. Notice that the client program, which defines `purchase`, a `Carpet` object, only uses it to call `Carpet` class functions, not `Rectangle` class functions. It does not even know that the `Carpet` class has a `Rectangle` object inside it. Notice also, in lines 39, 40, and 44, how `Carpet` class functions call `Rectangle` functions. Just as the user program calls `Carpet` functions through the name of its `Carpet` object, the `Carpet` class functions must call `Rectangle` functions through the name of its `Rectangle` object. The `Rectangle` object, defined in line 32, is named `size`. That is why the `Carpet` functions make calls like this:

```
size.getArea()
```



Checkpoint

7.20 Assume a `Map` class has a member variable named `position` that is an instance of the `Location` class. The `Location` class has a private member variable named `latitude` and a public member function called `getLatitude`. Which of the following lines of code would correctly get and return the value stored in `latitude`?

A) `return Location.latitude;`
B) `return Location.getLatitude();`
C) `return position.latitude;`
D) `return position.getLatitude();`

7.21 Write a class declaration for a class named `Circle`, which has the data member `radius`, a double, and the following member functions: `setRadius` and `getArea`. Write the code for these as inline functions.

7.22 Write a class declaration for a class named `Pizza` that has the data members `price`, a `double`, and `size`, a Circle object (declared in question 7.21). It also has member functions: `setPrice`, `setSize`, and `costPerSqIn`. Write the code for these as inline functions.

7.23 Write 4 lines of code that might appear in a client program using the `Pizza` class to do the following:
Define an instance of the `Pizza` class named `myPizza`.
Call a `Pizza` function to set the price.
Call a `Pizza` function to set the size (i.e., the radius).
Call a `Pizza` function to return the price per square inch and then print it.

7.11 Focus on Software Engineering: *Separating Class Specification, Implementation, and Client Code*

CONCEPT: Usually class declarations are stored in their own header files and member function definitions are stored in their own `.cpp` files.

In the programs we've looked at so far, the class declaration, the member function definitions, and the application program that uses the class are all stored in one file. A more conventional way of designing C++ programs is to store these in three separate files. Typically, program components are stored in the following fashion:

- Class declarations are stored in their own header files. A header file that contains a class declaration is called a *class specification file*. The name of the class specification file is usually the same as the name of the class, with a `.h` extension. For example, the `Rectangle` class would be declared in the file `Rectangle .h`.
- Any program that uses the class should `#include` this header file.
- The member function definitions for a class are stored in a separate `.cpp` file, which is called the *class implementation file*. The file usually has the same name as the class, with the `.cpp` extension. For example the `Rectangle` class member functions would be defined in the file `Rectangle.cpp`.
- The class `.cpp` file should be compiled and linked with the application program that uses the class. This program, also known as the *client program*, or *client code*, is the one that includes the `main` function. This process can be automated with a `project` or `make` utility. Integrated development environments such as Visual Studio also provide the means to create multi-file projects.

Let's see how we could rewrite Program 7-3, the rectangle program, using this design approach. First, the `Rectangle` class declaration would be stored in the following `Rectangle.h` file.

Contents of `Rectangle.h`

```
1  // File Rectangle.h -- Rectangle class specification file
2  #ifndef RECTANGLE_H
3  #define RECTANGLE_H
4
5  // Rectangle class declaration
6  class Rectangle
7  {
8     private:
9        double length;
10       double width;
11    public:
12       bool setLength(double);
13       bool setWidth(double);
14       double getLength();
15       double getWidth();
16       double getArea();
17 };
18 #endif
```

This is the specification file for the `Rectangle` class. It contains only the declaration of the `Rectangle` class. It does not contain any member function definitions. When we write other programs that use the `Rectangle` class, we can have an `#include` directive that includes this file. That way, we won't have to write the class declaration in every program that uses the `Rectangle` class.

This file also introduces two new preprocessor directives: `#ifndef` and `#endif`. The `#ifndef` directive that appears in line 2 is called an *include guard*. It prevents the header file from accidentally being included more than once. When your main program file has an `#include` directive for a header file, there is always the possibility that the header file will have an `#include` directive for a second header file. If your main program file also has an `#include` directive for the second header file, the preprocessor will include the second header file twice. Unless an include guard has been written into the second header file, an error will occur because the compiler will process the declarations in the second header file twice. Let's see how an include guard works.

The word `ifndef` stands for "if not defined". It is used to determine whether or not a specific constant has already been defined with a `#define` directive. When the `Rectangle.h` file is being compiled, the `#ifndef` directive checks for the existence of a constant named `RECTANGLE_H`. If this constant has not been defined yet, it is immediately defined in line 3, and the rest of the file is included. However, if the constant has already been defined, it means that this file has already been included. In that case, it is not included a second time. Instead, everything between the `#ifndef` and `#endif` directives is skipped. Note that the constant used in the `#infdef` and `#define` directives should be written in all capital letters and is customarily named `FILENAME_H`, where `FILENAME` is the name of the header file.

Next we need an implementation file that contains the class member function definitions. The implementation file for the `Rectangle` class is `Rectangle.cpp`.

Contents of `Rectangle.cpp`

```
1  // File Rectangle.cpp -- Rectangle class function implementation file
2  #include "Rectangle.h"
3
4  /*****************************************************************
5   *                    Rectangle::setLength                       *
6   * If the argument passed to the setLength function is zero or   *
7   * greater, it is copied into the member variable length, and true *
8   * is returned. If the argument is negative, the value of length  *
9   * remains unchanged and false is returned.                      *
10  *****************************************************************/
11 bool Rectangle::setLength(double len)
12 {
13    bool validData = true;
14
15    if (len >= 0)               // If the len is valid
16       length = len;            // copy it to length
17    else
18       validData = false;       // else leave length unchanged
19
20   return validData;
21 }
22
23 /*****************************************************************
24  *                    Rectangle::setWidth                        *
25  * If the argument passed to the setWidth function is zero or    *
26  * greater, it is copied into the member variable width, and true *
27  * is returned. If the argument is negative, the value of width  *
28  * remains unchanged and false is returned.                      *
29  *****************************************************************/
30 bool Rectangle::setWidth(double w)
31 {
32    bool validData = true;
33
34    if (w >= 0)                 // If w is valid
35       width = w;               // copy it to width
36    else
37       validData = false;       // else leave width unchanged
38
39    return validData;
40 }
41
42 /*****************************************************************
43  *                     Rectangle::getLength                      *
44  * This function returns the value in member variable length.    *
45  *****************************************************************/
46 double Rectangle::getLength()
47 {
48    return length;
49 }
50
```

```
51 /**************************************************************
52  *                      Rectangle::getWidth                   *
53  * This function returns the value in member variable width.  *
54  **************************************************************/
55 double Rectangle::getWidth()
56 {
57    return width;
58 }
59
60 /**************************************************************
61  *                        Rectangle::getArea                  *
62  * This function calculates and returns the area of the rectangle. *
63  **************************************************************/
64 double Rectangle::getArea()
65 {
66    return length * width;
67 }
```

Look at the code for the five functions. Notice that the three accessor functions, `getLength`, `getWidth`, and `getArea`, are the same as those that appeared in Program 7-3. However, a change has been made to the two mutator functions, `setLength` and `setWidth`, to illustrate another way in which public class functions can safeguard private member data. In Program 7-3, the `setLength` and `setWidth` functions use a default value for `length` and `width` if invalid data is passed to them. In the `Rectangle.cpp` code, these two functions return a Boolean value indicating whether or not the value received was stored in the member variable. If a valid argument is received, it is stored in the member variable and `true` is returned. If an invalid argument is received, the member variable is left unchanged and `false` is returned. The client program that uses this class must test the returned Boolean value to determine how to proceed.

Now look at line 2, which has the following `#include` directive:

```
#include "Rectangle.h"
```

This directive includes the `Rectangle.h` file, which contains the `Rectangle` class declaration. Notice that the name of the header file is enclosed in double-quote characters (`" "`) instead of angled brackets (< >). When you are including a C++ system header file, such as `iostream`, you enclose the name of the file in angled brackets. This indicates that the file is located in the compiler's *include file directory*. That is the directory or folder where all of the standard C++ header files are located. When you are including a header file that you have written, such as a class specification file, you enclose the name of the file in double-quote marks. This indicates that the file is located in the current project directory.

Any file that uses the `Rectangle` class must have an `#include` directive for the `Rectangle.h` file. We need to include `Rectangle.h` in the class specification file because the functions in this file belong to the `Rectangle` class. Before the compiler can process a function with `Rectangle::` in its name, it must have already processed the `Rectangle` class declaration.

With the `Rectangle` class stored in its own specification and implementation files, we can see how to use them in a program. Program 7-12 is a modified version of Program 7-3. Notice that Program 7-12 is much shorter than Program 7-3 because it does not contain the `Rectangle` class declaration or member function definitions. Instead, it is designed to be

compiled and linked with the class specification and implementation files. Program 7-12 only needs to contain the client code that creates and uses a `Rectangle` object.

Program 7-12

```
1  // This program that uses the Rectangle class.
2  // The Rectangle class declaration is in file Rectangle.h.
3  // The Rectangle member function definitions are in Rectangle.cpp
4  // These files should all be combined into a project.
5  #include <iostream>
6  #include "Rectangle.h"         // Contains Rectangle class declaration
7  using namespace std;
8
9  int main()
10 {
11    Rectangle box;              // Declare a Rectangle object
12    double boxLength, boxWidth;
13
14    //Get box length and width
15    cout << "This program will calculate the area of a rectangle.\n";
16    cout << "What is the length? ";
17    cin  >> boxLength;
18    cout << "What is the width? ";
19    cin  >> boxWidth;
20
21    // Call member functions to set box dimensions.
22    // If the function call returns false, it means the
23    // argument sent to it was invalid and not stored.
24    if (!box.setLength(boxLength))               // Store the length
25       cout << "Invalid box length entered.\n";
26    else if (!box.setWidth(boxWidth))            // Store the width
27       cout << "Invalid box width entered.\n";
28    else                                         // Both values were valid
29    {
30       // Call member functions to get box information to display
31       cout << "\nHere is the rectangle's data:\n";
32       cout << "Length: " << box.getLength() << endl;
33       cout << "Width : " << box.getWidth()  << endl;
34       cout << "Area  : " << box.getArea()   << endl;
35    }
36    return 0;
37 }
```

Notice that line 6 of Program 7-12 has an `#include` directive for the `Rectangle.h` file. This is needed so the `Rectangle` class declaration will be included in the file.

Now that we have created the three files for this program, the following steps must be taken to create an executable program.

- First, the implementation file, `Rectangle.cpp`, should be compiled to create an object file. This file would typically be named `Rectangle.obj`.
- Next, the main program file, located in file `pr7-12.cpp`, must be compiled to create an object file. This file would typically be named `pr7-12.obj`.
- Finally, the object files `pr7-12.obj` and `Rectangle.obj` are linked together to create an executable file, which would be named something like `pr7-12.exe`.

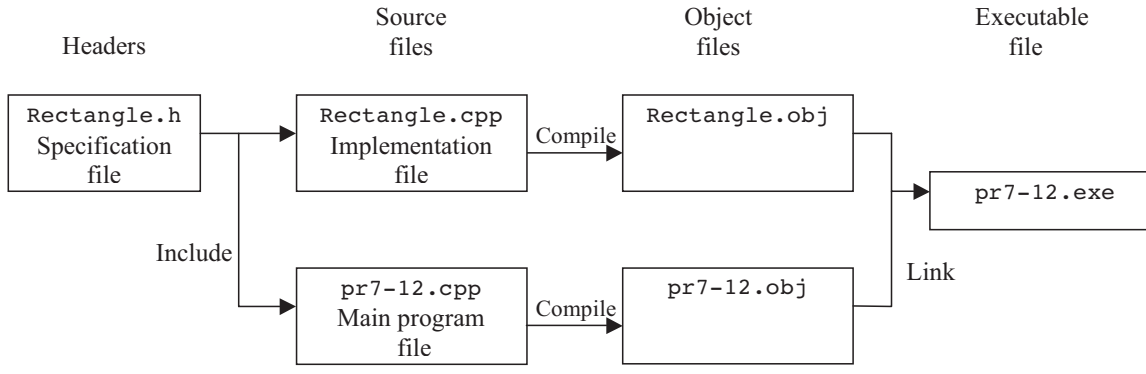
Table 7-1 summarizes how the different files of Program 7-12 are organized and compiled on a typical Windows computer.

Table 7-1 Files Used in Program 7-12

`Rectangle.h`	Contains the `Rectangle` class declaration. This file is included by `Rectangle.cpp` and `pr7-12.cpp`.
`Rectangle.cpp`	Contains the definitions of the `Rectangle` class member functions. This file is compiled to create an object file, such as `Rectangle.obj`.
`pr7-12.cpp`	Contains the application program that uses the class. In this case, the application program consists of just the function `main`. This file is compiled to create an object file, such as `pr7-12.obj`.
`Linking the .obj files`	The two object code files created by compiling `Rectangle.cpp` and `pr7-12.cpp` are linked to make the executable file `pr7-12.exe`

Figure 7-5 further illustrates this process.

Figure 7-5



The exact details on how these steps take place are different for each C++ development system. Fortunately, most systems perform all of these steps automatically for you. For example, in Microsoft Visual C++ you create a project, and then you simply add all of the files to the project. When you compile the project, the steps are taken care of for you and an executable file is generated. Once the executable file has been created, you can run the program. When valid values are entered for `boxLength` and `boxWidth` the output should be identical to that shown for Program 7-3.



NOTE: Appendix J, Multi-Source File Programs, which is located on the student CD, provides additional information on creating multifile projects.

Advantages of Using Multiple Files

Separating a client program from the details of a class it uses is considered good programming practice. A class is an example of an abstract data type and, as you learned earlier in this chapter, the only thing a programmer writing an application that uses the class needs to know is what the class does, what kind of data it can hold, what functions it provides,

and how to call them. Programmers, and any programs using the class, do not need to know anything about the implementation of the class. In addition, often many different programs use a particular class. If the implementation code that defines the class member functions is in its own separate file, this code does not have to be in all of these programs. They can each simply `#include` the file containing the defintions.

Separating a class into a specification file and an implementation file is also considered good programming practice. If you wish to give your class to other programmers, you don't have to share all of your source code. You can just provide a copy of the specification file and the compiled object file for the class's implementation. The other programmers simply insert the necessary `#include` directive into their programs, compile them, and link them with your class object file. This prevents other programmers, who might not understand all the details of your code, from making changes that introduce bugs.

Separating a class into specification and implementation files also makes things easier when class member functions must be modified. It is only necessary to modify the implementation file and recompile it to create a new object file. Programs that use the class don't have to be recompiled. They just have to be linked with the new object file.

Performing Input/Output in a Class Object

You may have noticed in Program 7-12 that we avoided doing any I/O inside the `Rectangle` class. In general is it considered good design to have class member functions avoid using `cin` and `cout`. This is so anyone writing a program that uses the class will not be locked into the way the class performs input or output. Unless a class is specifically designed to perform I/O, operations like user input and output are best left to the person designing the application. In general, classes should provide member functions for retrieving data values without displaying them on the screen. Likewise, they should provide member functions that store data into private member variables without using `cin`. Program 7-12 follows both of these practices.



NOTE: There are instances where it is appropriate for a class to perform I/O. For example, a class might be designed to display a menu on the screen and get the user's selection. Another example is a class designed to handle a program's file I/O. Classes that hold and manipulate data, however, should not be tied to any particular I/O routines. This allows them to be more versatile.



Checkpoint

7.24 Assume the following class components exist in a program:

`BasePay` class declaration
`BasePay` member function definitions
`Overtime` class declaration
`Overtime` member function definitions

What files would you store each of the above components in?

7.25 What header files should be included in the client program that uses the `BasePay` and `Overtime` classes?

7.12 Input Validation Objects

CONCEPT: Classes may be designed to validate user input.

One application of OOP is to design general-purpose objects that can be used by a variety of applications. An example of such an object is one that performs input validation. For example, assume a program displays a menu that allows the user to select items A, B, C, or D. The program should validate any character entered by the user and only accept one of these four letters.

In this section, we will design a general-purpose class to handle this type of input validation. Before we discuss the class, let's look at a C++ library function we will need.

The `toupper` Function

Our input validation class will use the `toupper` library function (pronounced "to upper"). To use the `toupper` function, you must place the following `#include` directive in your program:

```
#include <cctype>
```

The `toupper` function accepts a single character as its argument and returns the uppercase equivalent of that character. For example, the following statement will display the character `M` on the screen:

```
cout << toupper('m');
```

And the following statement assigns the character `A` to the variable `letter`:

```
letter = toupper('a');
```

If the character passed to `toupper` is already uppercase or is not a letter, that character is returned unchanged.

Among its other uses, the `toupper` function can simplify the task of comparing characters without regard to case. For example, look at the following code segment. It asks the user to enter a letter in the range K through Q, then attempts to validate the character that was entered.

```
cout << "Enter a letter in the range K-Q: ";
cin  >> letter;
while (letter < 'K' || letter > 'Q')
{  cout << "That is not a valid letter.\n";
   cout << "Enter a letter from K to Q: ";
   cin  >> letter;
}
```

This code will successfully validate the user's input as long as the character entered is uppercase. However, if the user enters a lowercase character in the range k through q, the code will fail. Those characters are higher on the ASCII chart than their uppercase equivalents, so the contents of `letter` will be greater than '`Q`' anytime a lowercase letter is entered. We can use the `toupper` function to fix this problem by comparing the uppercase form of the input to our range of valid characters. The following code shows how.

```
cout << "Enter a letter in the range K-Q: ";
cin  >> letter;
while (toupper (letter) < 'K' || toupper(letter) > 'Q')
{  cout << "That is not a valid letter.\n";
   cout << "Enter a letter from K to Q: ";
   cin  >> letter;
}
```

Note that the `toupper` function does not actually convert its argument to uppercase but returns the uppercase equivalent of it. After the function `toupper(letter)` has been executed, the content of `letter` is still the same as it was before the function was called. If we wish to actually convert a variable's contents to uppercase, we must use a statement like this:

```
letter = toupper(letter);
```

This statement assigns the return value of `toupper(letter)` to `letter`, thus converting the contents of the variable to uppercase. This approach is demonstrated in the following code.

```
cout << "Enter a letter in the range K-Q: ";
cin  >> letter;
letter = toupper(letter);
while (letter < 'K' || letter > 'Q')
{  cout << "That is not a valid letter.\n";
   cout << "Enter a letter from K to Q: ";
   cin  >> letter;
   letter = toupper(letter);
}
```

The `toupper` function will be used to implement this type of input validation in the `CharRange` class, which is discussed next.



NOTE: `toupper`, as well as other character-handling library functions, are discussed further in Chapter 12.

The `CharRange` Input Validation Class

Now we will examine the `CharRange` class. An object of the `CharRange` class allows the user to enter a character, then validates that it is within a specified range of characters. When the user enters a character outside the designated range, the object displays an error message and waits for the user to re-enter the character. The header file containing the class declaration and the implementation file holding the member function definitions are shown here.

Contents of `CharRange.h`

```
1  // CharRange.h -- CharRange class specification file
2  #ifndef CHARRANGE_H
3  #define CHARRANGE_H
4
5  class CharRange
6  {
7    private:
8       char input;      // User input
9       char lower;      // Lowest valid character
10      char upper;      // Highest valid character
11   public:
12      CharRange(char, char);
13      char getChar();
14 };
15 #endif
```

Contents of `CharRange.cpp`

```
1  // CharRange.cpp -- CharRange class function implementation file
2  #include <iostream>
3  #include <cctype>           // Needed to use toupper
4  #include "CharRange.h"
5  using namespace std;
6
7  /*********************************************
8   *       CharRange default constructor       *
9   *********************************************/
10 CharRange::CharRange(char l, char u)
11 {
12    lower = toupper(l);      // Set the valid lower
13    upper = toupper(u);      // and upper bounds
14 }
15
16 /*****************************************************
17  *                  CharRange ::getChar                *
18  * This function inputs a character, validates it is in *
19  * the correct range, then returns the valid character. *
20  *****************************************************/
21 char CharRange::getChar()
22 {
23    cin.get(input);          // Get a character
24    cin.ignore();            // Ignore the '\n' in the input buffer
25    input = toupper(input);  // Convert the character to uppercase
26
27    // Ensure character is in the correct range.
28    // If not, require the user to re-enter it.
29    while (input < lower || input > upper)
30    {
31       cout << "That is not a valid character.\n";
32       cout << "Enter a value from " << lower;
33       cout << " to " << upper << ".\n";
34       cin.get(input);
35       cin.ignore();
36       input = toupper(input);
37    }
38    return input;
39 }
```

Let's look at the class member functions. On lines 10 through 14, the constructor establishes the range of valid characters:

```
CharRange::CharRange(char l, char u)
{
   lower = toupper(l);
   upper = toupper(u);
}
```

The `toupper` function is used to get the uppercase equivalents of the values in the parameters `l` and `u`. These uppercase values are copied into the private member variables `lower` and `upper`. These mark the lower and upper ends of the valid range of characters.

Here's an example of how to create an object of the `CharRange` class:

```
CharRange input('A', 'D');
```

This statement creates a `CharRange` object named `input`. The object allows the user to enter a character in the ranges 'A' through 'D' and 'a' through 'd' (valid characters are A, B, C, D, a, b, c, and d).

Accepting and validating input is done by the `getChar` member function, whose code is found in lines 21 through 39. This function reads a character from the keyboard, converts it to uppercase, and validates it. If the character is outside the range specified by the `lower` and `upper` variables, whose values were passed to the constructor when the `CharRange` object was created, an error message is displayed and another character is read.



NOTE: As mentioned before, in general it is best to avoid using input and output statements in class member functions. This, however, is a class specifically designed to perform input and output.

Program 7-13 is a simple program that demonstrates the `CharRange` class. (This class will be used again later in other programs.)

Program 7-13

```
1  // This program uses the CharRange class and demonstrates its
2  // capabilities. This file should be combined into a project
3  // along with the CharRange.h and CharRange.cpp files.
4  #include <iostream>
5  #include "CharRange.h"
6  using namespace std;
7
8  int main()
9  {
10    char ch;                      // Holds user input
11
12    CharRange input('J', 'N');    // Create a CharRange object named
13                                  // input that ensures entered
14                                  // characters are in the range J - N
15
16    // Prompt the user to enter a letter
17    cout << "Enter any of the characters J, K, L, M, or N.\n";
18    cout << "Entering N will stop this program.\n";
19
20    // Call an input member function to accept the user input and
21    // validate it before returning it to be assigned to variable ch
22    ch = input.getChar();
23
24    // Continue allowing letters to be entered until an 'N' is input
25    while (ch != 'N')
26    {   cout << "You entered " << ch << endl;
27       ch = input.getChar();
28    }
29    return 0;
30 }
```

(program continues)

Program 7-13 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter any of the characters J, K, L, M, or N.
Entering N will stop this program.
j[Enter]
You entered J
L[Enter]
You entered L
p[Enter]
That is not a valid character.
Enter a value from J to N.
m[Enter]
You entered M
n[Enter]
```

7.13 Structures

CONCEPT: C++ allows a set of variables to be combined together into a single unit called a structure.

A *structure* is a programmer-defined data type that can hold many different data values. In the past, before the use of object-oriented programming became common, programmers typically used these to group logically connected data together into a single unit. Once a structure type is declared and its data members identified, multiple variables of this type can be created, just as multiple objects can be created for the same class.

Although structures are less commonly used today, it is important that you know what they are and how to use them. Not only may you encounter them in older programs, but there are actually some instances in which classes will not work and you must use structures. You will see an example of this later in this chapter.

The way a structure is declared is similar to the way a class is declared, with the following differences:

- The key word `struct` is used instead of the key word `class`.
- Although structures *can* include member functions, they rarely do. So normally a structure declaration only declares member variables.
- Structure declarations normally do not include the access specifiers `public` or `private`. Unlike class members, which are private by default, members of a structure default to being public. Programmers normally want them to remain public, and simply use the default.

Here is an example of a declaration for a structure that bundles together five variables holding payroll data for an employee. The name of this particular structure is `PayRoll`. Notice that it begins with a capital letter. The convention is to begin structure names, just like class names, with an uppercase letter. Notice also that, like a class declaration, there must be a semicolon after the closing brace of the declaration.

```
struct PayRoll
{
   int     empNumber;
   string  name;
   double  hours,
           payRate,
           grossPay;
};
```

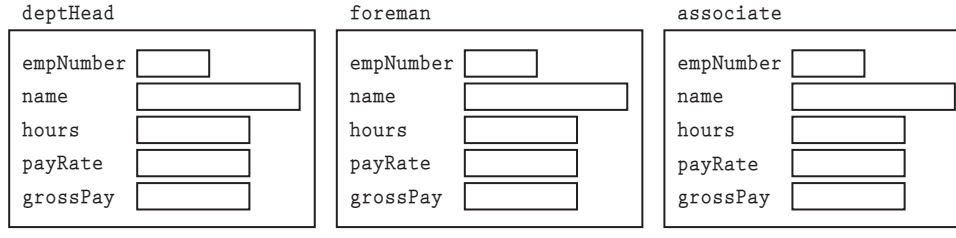
Just as a class declaration is not instantiated until objects of the class are created, a structure declaration does not create any instances of the structure. The structure declaration in our example simply tells the compiler what a `Payroll` structure looks like. It in essence creates a new data type named `Payroll`.

You define variables that are `Payroll` structures the way you define any variable, by first listing the data type, and then the variable name. The following definition creates three variables that are `Payroll` structures.

```
Payroll deptHead, foreman, associate;
```

Each is an instance of a `Payroll` structure, with its own memory allocated to hold its member data. Notice that although the three structure variables have *distinct* names, each contains members with the *same* name. Figure 7.6 illustrates this.

Figure 7-6



Accessing Structure Members



VideoNote
Creating and Using Structures

Members of a structure are accessed just like public members of a class, with the dot operator. However, the data members of a class are normally private and must be accessed through functions. Because structure data members are public, they are accessed directly and can be used like regular variables. The following statements assign values to the `empNumber` member of each of the `Payroll` variables we created.

```
deptHead.empNumber = 475;
foreman.empNumber = 897;
associate.empNumber = 729;
```

And the following statements display the contents of all the `deptHead`'s members.

```
cout << deptHead.empNumber << endl;
cout << deptHead.name << endl;
cout << deptHead.hours << endl;
cout << deptHead.payRate << endl;
cout << deptHead.grossPay << endl;
```

Program 7-14 is a complete program that uses the `PayRoll` structure. Notice how the individual structure members are used just like regular variables in `cin` statements, in `cout` statements, and in mathematical operations.

Program 7-14

```
1  // This program demonstrates the use of a structure.
2  #include <iostream>
3  #include <iomanip>
4  #include <string>
5  using namespace std;
6
7  struct PayRoll
8  {
9     int    empNumber;  // Employee number
10    string name;       // Employee name
11    double hours,      // Hours worked
12           payRate;    // Hourly pay rate
13 };
14
15 int main()
16 {
17    PayRoll employee;  // Employee is a PayRoll structure
18    double grossPay;   // Gross amount the employee earned this week
19
20    //Get the employee's data
21    cout << "Enter the employee's number: ";
22    cin  >> employee.empNumber;
23
24    cout << "Enter the employee's name: ";
25    cin.ignore();      // Skip the '\n' character left in the input buffer
26    getline(cin, employee.name);
27
28    cout << "Hours worked this week: ";
29    cin  >> employee.hours;
30
31    cout << "Employee's hourly pay rate: ";
32    cin  >> employee.payRate;
33
34    // Calculate the employee's gross pay
35    grossPay = employee.hours * employee.payRate;
36
37    // Display the results
38    cout << "\nHere is the employee's payroll data:\n";
39    cout << "Name:            " << employee.name << endl;
40    cout << "Employee number: " << employee.empNumber << endl;
41    cout << "Hours worked:    " << employee.hours << endl;
42    cout << "Hourly pay rate: " << employee.payRate << endl;
43    cout << fixed << showpoint  << setprecision(2);
44    cout << "Gross pay:      $" << grossPay << endl;
45    return 0;
46 }
```

(program continues)

Program 7-14 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter the employee's number: 2214[Enter]
Enter the employee's name: Jack Smith[Enter]
Hours worked this week: 40[Enter]
Employee's hourly pay rate: 12.50[Enter]

Here is the employee's payroll data:
Name:            Jack Smith
Employee number: 2214
Hours worked:    40
Hourly pay rate: 12.5
Gross pay:      $500.00
```

In Program 7-14 the variable `employee` is defined in line 17 to be an instance of a `Payroll` structure. Its five data members can then be accessed with the dot operator through the name of the variable. For example, in line 22 the following statement reads a value into the variable's `empNumber` member.

```
cin  >> employee.empNumber;        // Correct
```

It would have been wrong to try to access this member through the name of the structure type.

```
cin  >> Payroll.empNumber;         // Wrong!
```

Displaying and Comparing Structure Variables

In Program 7-14 each member of the `employee` structure variable was displayed separately. This is necessary because the entire contents of a structure variable cannot be displayed by simply passing the whole variable to `cout`. For example, the following statement will *not* work.

```
cout << employee << endl;                // Error!
```

Likewise, while it is possible to compare the contents of two individual structure members, you cannot perform comparison operations on entire structures. For example, if `employee1` and `employee2` are both `Payroll` structure variables, this comparison will cause an error.

```
if (employee1 == employee2)              // Error!
```

The following comparison, on the other hand, is perfectly legal.

```
if (employee1.hours == employee2.hours) // Legal
```

Initializing a Structure

There are two ways a structure variable can be initialized when it is defined: with an initialization list or with a constructor.

The simplest way to initialize the members of a structure variable is to use an initialization list. An *initialization list* is a list of values used to initialize a set of memory locations. The

items in the list are separated by commas and surrounded by braces. Suppose, for example, the following `Date` structure has been declared:

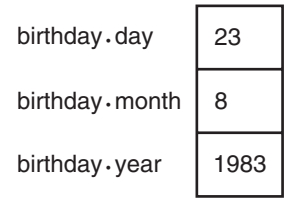
```
struct Date
{  int day,
       month,
       year;
};
```

A `Date` variable can now be defined and initialized by following the variable name with the assignment operator and an initialization list, as shown here:

```
Date birthday = {23, 8, 1983};
```

This statement defines `birthday` to be a variable which is a `Date` structure. The values inside the curly braces are assigned to its members in order. So the data members of `birthday` have been initialized as shown in Figure 7-7.

Figure 7-7



It is also possible to initialize just some of the members of a structure variable. For example, if we know the birthday to be stored is August 23, but do not know the year, the variable could be defined and initialized like this:

```
Date birthday = {23, 8};
```

Only the `day` and `month` members are initialized here. The `year` member is not initialized. If you leave a structure member uninitialized, however, you must leave all the members that follow it uninitialized as well. C++ does not provide a way to skip members when using an initialization list. The following statement, which attempts to skip the initialization of the `month` member, is *not* legal.

```
Date birthday = {23, , 1983};    // Illegal!
```

It is important to note that you cannot initialize a structure member in the declaration of the structure because the structure declaration just creates a new data type. No variables of this type exist yet. For example, the following declaration is illegal:

```
// Illegal structure declaration
struct Date
{  int day   = 23,
       month = 8,
       year  = 1983;
};
```

Because a structure declaration only declares what a structure "looks like", the member variables are not created in memory until the structure is instantiated by defining a variable of that structure type. Until then there is no place to store an initial value.

Although an initialization list is easy to use, it has two drawbacks:

1. It does not allow you to leave some members uninitialized and still initialize others that follow.
2. It will not work on many compilers if the structure includes any objects, such as strings.

In these cases you can initialize structure member variables the same way you initialize class member variables—by using a constructor. As with a class constructor, a constructor for a structure must be a public member function with the same name as the structure and no return type. Because all structure members are public by default, however, the key word `public` does not need to be used. Here is a structure declaration for a structure named `Employee`. It includes a 2-argument constructor that provides default values in case an `Employee` variable is created without passing any arguments to the constructor.

```
struct Employee
{
   string name;              // Employee name
   int vacationDays,         // Vacation days allowed per year
       daysUsed;             // Vacation days used so far

   Employee(string n = "", int d = 0)  // Constructor
   {  name = n;
      vacationDays = 10;
      daysUsed = d;
   }
};
```

Nested Structures

Just as objects of one class can be nested within another class, instances of one structure can be nested within another structure. For example, consider the following structure declarations:

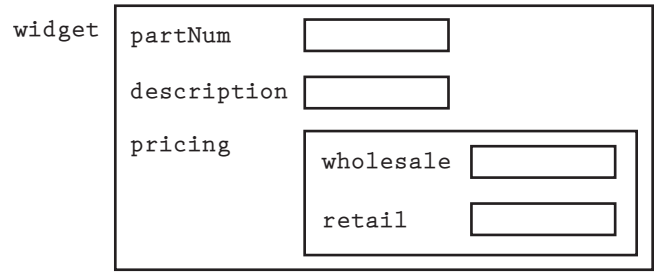
```
struct Costs
{
   double wholesale;
   double retail;
};

struct Item
{
   string partNum;
   string description;
   Costs pricing;
};
```

The `Costs` structure has two `double` members, `wholesale` and `retail`. The `Item` structure has three members. The first two, `partNum` and `description`, are `string` objects. The third, `pricing`, is a nested `Costs` structure. Assume variable `widget` is defined to be an `Item` structure:

```
Item widget;
```

Figure 7-8 illustrates its members.

Figure 7-8



They would be accessed as follows:

```
widget.partnum = "123A";
widget.description = "iron widget";
widget.pricing.wholesale = 100.0;
widget.pricing.retail = 150.0;
```

Notice that `wholesale` and `retail` are not members of `widget`; `pricing` is. To access `wholesale` and `retail`, `widget`'s `pricing` member must first be accessed and then, because it is a `Costs` structure, its `wholesale` and `retail` members can be accessed. Notice also, as with all structures, it is the *member* name, not the *structure* name, that must be used in accessing a member. The following statements would not be legal.

```
cout << widget.retail;            // Wrong!
cout << widget.Costs.wholesale;   // Wrong!
```

When you are deciding whether to use nested structures or not, think about how various members are related. A structure bundles together items that logically belong together. Normally the members of a structure are attributes describing some object. In our example, the object was a widget and its part number, description, and wholesale and retail prices were its attributes. When some of the attributes are related and form a logical subgroup of the object's attributes, it makes sense to bundle them together and use a nested structure. Notice the relatedness of the attributes in the inner structure of Program 7-15, which uses a nested structure.

Program 7-15

```
1 // This program demonstrates the use of a nested structure.
2 #include <iostream>
3 #include <iomanip>
4 #include <string>
5 using namespace std;
6
```

(program continues)

Program 7-15 *(continued)*

```
 7 struct CostInfo
 8 {
 9    double food,     // Food costs
10           medical,  // Medical costs
11           license,  // License fee
12           misc;     // Miscellaneous costs
13 };
14
15 struct PetInfo
16 {
17    string name;     // Pet name
18    string type;     // Pet type
19    int    age;      // Pet age
20
21    CostInfo cost;   // A PetInfo structure has a CostInfo structure
22                     // nested inside as one of its members
23
24    PetInfo()        // Default constructor
25    {   name = "unknown";
26        type = "unknown";
27        age = 0;
28        cost.food = cost.medical = cost.license = cost.misc = 0.00;
29    }
30 };
31
32 int main()
33 {
34    // Define a PetInfo structure variable called pet
35    PetInfo pet;
36
37    // Assign values to the pet member variables.
38    // Notice that cost.misc is not assigned a value,
39    // so it remains 0, as set by the constructor.
40    pet.name = "Sassy";
41    pet.type = "cat";
42    pet.age = 5;
43    pet.cost.food = 300.00;
44    pet.cost.medical = 200.00;
45    pet.cost.license = 7.00;
46
47    // Display the total annual costs for the pet
48    cout << fixed << showpoint << setprecision(2);
49    cout << "Annual costs for my " << pet.age << "-year-old "
50         << pet.type << " " << pet.name  << " are $"
51         << (pet.cost.food + pet.cost.medical +
52             pet.cost.license + pet.cost.misc) << endl;
53    return 0;
54 }
```

Sample Output

```
Annual costs for my 5-year-old cat Sassy are $507.00
```



Checkpoint

7.26 Write a structure declaration for a structure named `Student` that holds the following data about a student:

ID (`int`)
entry year (`int`)
GPA (`double`)

Then write definition statements that create the following two `Student` variables and initialize them using initialization lists.

Variable `s1` should have ID number 1234, entry year 2008, and GPA 3.41.
Variable `s2` should have ID number 5678 and entry year 2010. The GPA member should be left uninitialized.

7.27 Write a structure declaration for a structure named `Account` that holds the following data about a savings account. Include a constructor that allows data values to be passed in for all four members.

Account number (`string`)
Account balance (`double`)
Interest rate (`double`)
Average monthly balance (`double`)

Then write a definition statement for an `Account` variable that initializes the members with the following data:

Account number: ACZ42137
Account balance: $4512.59
Interest rate: 4%
Average monthly balance: $4217.07

7.28 The following program skeleton, when complete, asks the user to enter the following information about his or her favorite movie:

Name of the movie
Name of the movie's director
The year the movie was released

Complete the program by declaring the structure that holds this information, defining a structure variable, and writing the required individual statements.

```
#include <iostream>
#include <string>
using namespace std;

// Write the structure declaration to hold the movie information.

int main()
{
   // Define the structure variable here.

   cout << "Enter the following information about your "
        << " favorite movie.\n" << "Name: ";
   // Write a statement here that lets the user enter a movie name.
   // Store it in the appropriate structure member.
```

```
    cout << "Director: ";
    // Write a statement here that lets the user enter the director's
    // name. Store it in the appropriate structure member.

    cout << "Year of Release: ";
    // Write a statement here that lets the user enter the movie
    // release year. Store it in the appropriate structure member.

    cout << "\nHere is information on your favorite movie:\n";
    // Write statements here that display the information
    // just entered into the structure variable.
    return 0;
}
```

7.29 Write a structure declaration named `Measurement`, with the following members:
`miles`, an integer
`meters`, a `long` integer

7.30 Write a structure declaration named `Destination`, with the following members:
`city`, a string
`distance`, a `Measurement` structure (declared in question 7.29)

Now define a variable of this structure type.

7.31 Write assignment statements that store the following information in the variable you defined in question 7.30:
City: Tupelo
Miles 375
Meters: 603,375

Passing Structures to Functions

Structure variables, just like class objects, can be passed to functions by value, by reference, and by constant reference. By default, they are passed by value. This means that a copy of the entire original structure is made and passed to the function. Because it is not desirable to take the time to copy an entire structure, unless it is quite small, structures are normally passed to functions by reference. This, however, gives the function access to the member variables of the original structure, allowing it to change them. If you do not want a function to change any member variable values, the structure variable should be passed to the function as a constant reference.

Program 7-16 is a modification of Program 7-9 that defines a structure variable and passes it to two functions.

Program 7-16

```
1 // This program passes a structure variable to one function
2 // by reference and to another as a constant reference.
3 #include <iostream>
4 #include <iomanip>
5 #include <string>
6 using namespace std;
7
```

(program continues)

Program 7-16 *(continued)*

```
 8 struct InvItem               // Holds data for an inventory item
 9 {
10    int partNum;              // Part number
11    string description;       // Item description
12    int onHand;               // Units on hand
13    double price;             // Unit price
14 };
15
16 // Function prototypes
17 void getItemData(InvItem &); // Function getItemData will receive an
18                               // InvItem structure passed to it by
19                               // reference so new values can be stored
20                               // in its member variables.
21
22 void showItem(const InvItem &);
23                               // Function showItem will receive an
24                               // InvItem structure passed to it as a
25                               // constant reference because showItem
26                               // just needs display member variable
27                               // values, not change them.
28 int main()
29 {
30    InvItem part;             // Define an InvItem structure variable.
31
32    getItemData(part);
33    showItem(part);
34    return 0;
35 }
36
37 /*****************************************************************
38  *                          getItemData                          *
39  * This function stores data input by the user in the members of an *
40  * InvItem structure variable passed to the function by reference.  *
41  * ***************************************************************/
42 void getItemData(InvItem &item)
43 {
44    cout << "Enter the part number: ";
45    cin  >> item.partNum;
46    cout << "Enter the part description: ";
47    cin.get();                // Move past the '\n' left in the
48                              // input buffer by the last input.
49    getline(cin, item.description);
50    cout << "Enter the quantity on hand: ";
51    cin  >> item.onHand;
52    cout << "Enter the unit price: ";
53    cin  >> item.price;
54 }
55
```

(program continues)

Program 7-16 *(continued)*

```
56 /***************************************************************
57  *                          showItem                           *
58  * This function displays the data stored in the members of an *
59  * InvItem structure variable passed to it as a constant reference. *
60  * *************************************************************/
61 void showItem(const InvItem &item)
62 {
63    cout << fixed << showpoint << setprecision(2) << endl;;
64    cout << "Part Number  : "  << item.partNum << endl;
65    cout << "Description  : "  << item.description << endl;
66    cout << "Units On Hand: "  << item.onHand << endl;
67    cout << "Price        : $" << item.price << endl;
68 }
```

Program Output with Example Input Shown in Bold

```
Enter the part number: 800[Enter]
Enter the part description: Screwdriver[Enter]
Enter the quantity on hand: 135[Enter]
Enter the unit price: 1.25[Enter]

Part Number  : 800
Description  : Screwdriver
Units On Hand: 135
Price        : $1.25
```

Returning a Structure from a Function

A structure variable can also be returned from a function. In this case the return type of the function is the name of the structure. Program 7-16 could have been written to allow the `getItemData` function to create a local instance of an `InvItem` structure, place data values into its member variables, and then pass it back to `main`, instead of receiving it from `main` as a reference variable. This is what the revised `getItemData` function would look like.

```
/***********************************************************
 *                       getItemData                       *
 * This function stores data input by the user in the members  *
 * of a local InvItem structure variable and then returns it.  *
 * *********************************************************/

InvItem getItemData()
{
   InvItem item;          // Create a local InvItem variable
                          // to hold data until it can be returned.
   cout << "Enter the part number: ";
   cin  >> item.partNum;
   cout << "Enter the part description: ";
   cin.get();             // Move past the '\n' left in the
                          // input buffer by the last input.
   getline(cin, item.description);
```

```
   cout << "Enter the quantity on hand: ";
   cin  >> item.onHand;
   cout << "Enter the unit price: ";
   cin  >> item.price;

   return item;
}
```

And here is how it would be called from `main`.

```
part = getItemData();
```

This version of Program 7-16 can be found on the student CD as Program 7-16B.



NOTE: In Chapter 6 you learned that C++ only allows you to return a single value from a function. Structures, however, provide a way around this limitation. Even though a structure may have several members, it is technically a single object. By packaging multiple values inside a structure, you can return as many values as you need from a function.



Checkpoint

```
struct Rectangle
{
   int length;
   int width;
};
```

7.32 Write a function that accepts the `Rectangle` structure defined above as its argument and displays the structure's contents on the screen.

7.33 Write a function that uses a `Rectangle` structure reference variable as its parameter and stores the user's input in the structure's members.

7.34 Write a function that returns a `Rectangle` structure. The function should create a local `Rectangle` variable, store the user's input in its members, and then return it.

Unions

A union is like a structure, except all the member variables occupy the *same* memory area, so only one member can be used at a time. A union might be used in an application where the program needs to work with two or more values (of different data types), but only needs to use one of the values at a time. Unions conserve memory by storing all their members in the same memory location.

Unions are declared just like structures, except the key word `union` is used instead of `struct`. Here is an example:

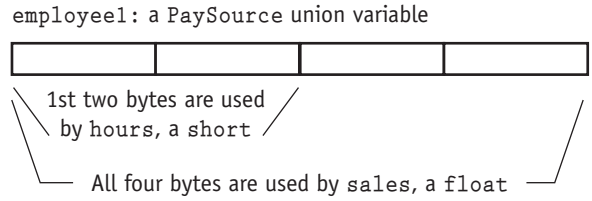
```
union PaySource
{
   short hours;
   float sales;
};
```

A union variable of the data type shown can then be defined like this:

```
PaySource employee1;
```

The `PaySource` union variable defined here has two members: `hours` (a `short`) and `sales` (a `float`). The entire variable will only take up as much memory as the largest member (in this case, a `float`). The way this variable is stored on a typical computer is illustrated in Figure 7-9.

Figure 7-9



As shown in Figure 7-9, this union uses four bytes on a typical computer. It can store a `short` or a `float`, depending on which member is used. When a value is stored in the `sales` member, all four bytes are needed to hold the data. When a value is stored in the `hours` member, only the first two bytes are used. Obviously, both members can't hold values at the same time. This union is demonstrated in Program 7-17.

Program 7-17

```
1  // This program demonstrates a union.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  union PaySource            // Declare a union
7  {
8     short hours;            // These two variables share
9     float sales;            // the same memory space
10 };
11
12 int main()
13 {
14    PaySource employee1;    // employee1 is a PaySource union
15                            // This employee can have hours or
16                            // sales, but not both at once
17
18    char  hourlyType;       // 'y' if hourly, 'n' if on commission
19    float payRate, grossPay;
20
21    cout << fixed << showpoint << setprecision(2);
22    cout << "This program calculates either hourly wages or "
23         << "sales commission.\n";
24    cout << "Is this an hourly employee (y or n)? ";
25    cin  >> hourlyType;
26
```

(program continues)

Program 7-17 *(continued)*

```
27    if (hourlyType == 'y')  // This is an hourly employee
28    {
29       cout << "What is the hourly pay rate? ";
30       cin  >> payRate;
31       cout << "How many hours were worked? ";
32       cin  >> employee1.hours;
33       grossPay = employee1.hours * payRate;
34       cout << "Gross pay: $" << grossPay << endl;
35    }
36    else                    // Employee works on commission
37    {
38       cout << "What are the total sales for this employee? ";
39       cin  >> employee1.sales;
40       grossPay = employee1.sales * 0.10;
41       cout << "Gross pay: $" << grossPay << endl;
42    }
43    return 0;
44 }
```

Program Output with Example Input Shown in Bold

```
This program calculates either hourly wages or sales commission.
Is this an hourly employee (y or n)? y[Enter]
What is the hourly pay rate? 20[Enter]
How many hours were worked? 40[Enter]
Gross pay: $800.00
```

Program Output with Other Example Input Shown in Bold

```
This program calculates either hourly wages or sales commission.
Is this an hourly employee (y or n)? n[Enter]
What are the total sales for this employee? 5000[Enter]
Gross pay: $500.00
```

Everything else you already know about structures applies to unions.

7.14 Home Software Company OOP Case Study

You are a programmer for the Home Software Company assigned to develop a class that models the basic workings of a bank account. The class should perform the following tasks:

- Save the account balance.
- Save the number of transactions performed on the account.
- Allow deposits to be made to the account.
- Allow withrawals to be taken from the account.
- Calculate interest for the period.
- Report the current account balance at any time.
- Report the current number of transactions at any time.

Private Member Variables

Table 7-2 lists the private member variables needed by the class.

Table 7-2 Private Member Variables of the Account Class

Variable	Description
`balance`	A `double` that holds the current account balance
`intRate`	A `double` that holds the interest rate for the period
`interest`	A `double` that holds the interest earned for the current period
`transactions`	An integer that holds the current number of transactions

Public Member Functions

Table 7-3 lists the public member functions in the class.

Table 7-3 Public Member Functions of the Account Class

Function	Description
constructor	Takes arguments to be initially stored in the `balance` and `intRate` members. The default value for the balance is zero and the default value for the interest rate is 0.045.
`makeDeposit`	Takes a `double` argument that is the amount of the deposit. This argument is added to `balance`.
`withdraw`	Takes a `double` argument that is the amount of the withdrawal. This value is subtracted from the balance, unless the withdrawal amount is greater than the balance. If this happens, the function reports an error.
`calcInterest`	Takes no arguments. This function calculates the amount of interest for the current period, stores this value in the `interest` member, and then adds it to the `balance` member.
`getBalance`	Returns the current balance (stored in the `balance` member).
`getInterest`	Returns the interest earned for the current period (stored in the `interest` member).
`getTransactions`	Returns the number of transactions for the current period (stored in the `transactions` member).

The Class Declaration

The following listing shows the class declaration.

Contents of `Account.h`

```
1 // File Account.h -- Account class specification file
2 class Account
3 {
4  private:
5    double balance;
6    double intRate;
7    double interest;
8    int transactions;
9
```

```
10  public:
11
12     Account(double irate = 0.045, double bal = 0.0)
13     {  balance = bal; intRate = irate;
14        interest = 0; transactions = 0;
15     }
16
17     void makeDeposit(double amount)
18     {  balance += amount; transactions++;
19     }
20
21     bool withdraw(double amount);   // Defined in account.cpp
22
23     void calcInterest()
24     {  interest = balance * intRate; balance += interest;
25     }
26
27     double getBalance()
28     {  return balance;
29     }
30
31     double getInterest()
32     {  return interest;
33     }
34
35     int getTransactions()
36     {  return transactions;
37     }
38  };
```

The `withdraw` Member Function

The only member function not defined inline in the class declaration is `withdraw`. The purpose of that function is to subtract the amount of a withdrawal from the `balance` member. If the amount to be withdrawn is greater than the current balance, however, no withdrawal is made. The function returns true if the withdrawal is made or false if there is not enough in the account.

Contents of `Account.cpp`

```
1  // File Account.cpp -- Account class function implementation file
2  #include "Account.h"
3
4  bool Account::withdraw(double amount)
5  {
6     if (balance < amount)
7        return false;   // Not enough in the account
8     else
9     {
10       balance -= amount;
11       transactions++;
12       return true;
13    }
14 }
```

The Class Interface

The `balance`, `intRate`, `interest`, and `transactions` member variables are private, so they are hidden from the world outside the class. This is because a programmer with direct access to these variables might unknowingly commit any of the following errors:

- A deposit or withdrawal might be made without the `transactions` member being incremented.
- A withdrawal might be made for more than is in the account. This will cause the `balance` member to have a negative value.
- The interest rate might be calculated and the `balance` member adjusted, but the amount of interest might not get recorded in the `intRate` member.
- The wrong interest rate might be used.

Because of the potential for these errors, the class contains public member functions that ensure the proper steps are taken when the account is manipulated.

Implementing the Class

Program 7-18 shows an implementation of the `Account` class. It presents a menu for displaying a savings account's balance, number of transactions, and interest earned. It also allows the user to deposit an amount into the account, make a withdrawal from the account, and calculate the interest earned for the current period. (Note that the `CharRange` input validation class introduced earlier in this chapter is also used in the program.)

Program 7-18

```
1  // This client program uses the Account class to perform simple banking
2  // operations. This file should be combined into a project along with
3  // the Account.h, Account.cpp, CharRange.h, and CharRange.cpp files.
4  #include <iostream>
5  #include <iomanip>
6  #include <cctype>
7  #include "Account.h"
8  #include "CharRange.h"
9  using namespace std;
10
11 // Function prototypes
12 void displayMenu();
13 void makeDeposit(Account &);
14 void withdraw(Account &);
15
16 int main()
17 {
18    Account savings;           // Account object to model savings account
19    CharRange input('A', 'G'); // CharRange object to do input validation
20    char choice;
21
22    cout << fixed << showpoint << setprecision(2);
```

(program continues)

Program 7-18 *(continued)*

```
23    do
24    {
25       displayMenu();
26       choice = input.getChar(); // This returns only an uppercase A - G
27       switch(choice)
28       {
29          case 'A': cout << "The current balance is $";
30                    cout << savings.getBalance() << endl;
31                    break;
32          case 'B': cout << "There have been ";
33                    cout << savings.getTransactions()
34                         << " transactions.\n";
35                    break;
36          case 'C': cout << "Interest earned for this period: $";
37                    cout << savings.getInterest() << endl;
38                    break;
39          case 'D': makeDeposit(savings);
40                    break;
41          case 'E': withdraw(savings);
42                    break;
43          case 'F': savings.calcInterest();
44                    cout << "Interest added.\n";
45       }
46    } while(choice != 'G');
47    return 0;
48 }
49
50 /*************************************************************
51  *                       displayMenu                         *
52  * This function displays the user's menu on the screen.     *
53  *************************************************************/
54 void displayMenu()
55 {
56    cout << "\n                MENU\n\n";
57    cout << "a) Display the account balance\n";
58    cout << "b) Display the number of transactions\n";
59    cout << "c) Display interest earned for this period\n";
60    cout << "d) Make a deposit\n";
61    cout << "e) Make a withdrawal\n";
62    cout << "f) Add interest for this period\n";
63    cout << "g) Exit the program\n\n";
64    cout << "Enter your choice: ";
65 }
66
67 /*************************************************************
68  *                       makeDeposit                         *
69  * This function accepts a reference to an Account object.   *
70  * The user is prompted for the dollar amount of the deposit,*
71  * and the makeDeposit member of the Account object is       *
72  * then called.                                              *
73  *************************************************************/
74 void makeDeposit(Account &account)
75 {
76    double dollars;
77
```

(program continues)

Program 7-18 *(continued)*

```
78    cout << "Enter the amount of the deposit: ";
79    cin  >> dollars;
80    cin.ignore();
81    account.makeDeposit(dollars);
82 }
83
84 /*************************************************************
85  *                        withdraw                           *
86  * This function accepts a reference to an Account object.   *
87  * The user is prompted for the dollar amount of the withdrawal,*
88  * and the withdraw member of the Account object is then called.*
89  *************************************************************/
90 void withdraw(Account &account)
91 {
92    double dollars;
93
94    cout << "Enter the amount of the withdrawal: ";
95    cin  >> dollars;
96    cin.ignore();
97    if (!account.withdraw(dollars))
98       cout << "ERROR: Withdrawal amount too large.\n\n";
99 }
```

Program Output with Example Input Shown in Bold

```
                Menu
a) Display the account balance
b) Display the number of transactions
c) Display interest earned for this period
d) Make a deposit
e) Make a withdrawal
f) Add interest for this period
g) Exit the program

Enter your choice: d[Enter]
Enter the amount of the deposit: 500[Enter]

                Menu
a) Display the account balance
b) Display the number of transactions
c) Display interest earned for this period
d) Make a deposit
e) Make a withdrawal
f) Add interest for this period
g) Exit the program

Enter your choice: a[Enter]
The current balance is $500.00

                Menu
a) Display the account balance
b) Display the number of transactions
c) Display interest earned for this period
d) Make a deposit
e) Make a withdrawal
f) Add interest for this period
g) Exit the program

Enter your choice: e[Enter]
Enter the amount of the withdrawal: 700[Enter]
ERROR: Withdrawal amount too large.
```

(program output continues)

Program 7-18 *(continued)*

```
                Menu
a) Display the account balance
b) Display the number of transactions
c) Display interest earned for this period
d) Make a deposit
e) Make a withdrawal
f) Add interest for this period
g) Exit the program

Enter your choice: e[Enter]
Enter the amount of the withdrawal: 200[Enter]

                Menu
a) Display the account balance
b) Display the number of transactions
c) Display interest earned for this period
d) Make a deposit
e) Make a withdrawal
f) Add interest for this period
g) Exit the program

Enter your choice: f[Enter]
Interest added.

                Menu
a) Display the account balance
b) Display the number of transactions
c) Display interest earned for this period
d) Make a deposit
e) Make a withdrawal
f) Add interest for this period
g) Exit the program

Enter your choice: a[Enter]
The current balance is: $313.50

                Menu
a) Display the account balance
b) Display the number of transactions
c) Display interest earned for this period
d) Make a deposit
e) Make a withdrawal
f) Add interest for this period
g) Exit the program

Enter your choice: g[Enter]
```

7.15 Introduction to Object-Oriented Analysis and Design

The planning phase of any program's development is extremely important. It is during this phase that the programmer carefully examines the program requirements and determines what the program must be able to do. In OOP terminology, this phase of program development is known as the *object-oriented analysis phase*. It is during this time that the programmer determines what classes will be used.

The process of object-oriented analysis includes the following steps:

1. Identify the classes and objects to be used in the program.
2. Define the attributes for each class.
3. Define the behaviors for each class.
4. Define the relationships between classes.

Let's look at each step more closely.

1. Identify the Classes and Objects.

Remember, a class is a package that consists of data and procedures that perform operations on the data. In order to determine the classes that will appear in a program, the programmer should think of the major data elements and decide what procedures or actions are required for each class. For example, consider a restaurant that uses an object-oriented program to enter customer orders. A customer order is a list of menu items with their respective prices. The restaurant uses this list to charge the customer, so a class could be created to model it. Also, the restaurant's menu has several main entrees, appetizers, side dishes, and beverages to choose from. A class could be designed to represent menu items as well.

Classes can be easily designed to model real-world objects, such as customer orders and a restaurant's menu items. Here are some other types of items that may be candidates for classes in a program:

- User-interface components, such as windows, menus, and dialog boxes
- Input/output devices, such as the keyboard, mouse, display, and printer
- Physical objects, such as vehicles, machines, or manufactured products
- Recordkeeping items, such as customer histories, and payroll records
- A role played by a human (employee, client, teacher, student, and so forth).

2. Define Each Class's Attributes.

A class's *attributes* are the data elements used to describe an object instantiated from the class. They are all the values needed for the object to function properly in the program. Using the restaurant example, here is the beginning of a possible specification for a `menuItem` class.

Class name: `MenuItem`

Attributes:
```
itemName
price
category    // 1 = appetizer, 2 = salad,   3 = entrée
            // 4 = side dish, 5 = dessert, 6 = beverage
```

And here is the beginning of a possible specification for a `CustomerOrder` class.

Class name: `CustomerOrder`

Attributes:
```
orderNumber
tableNumber
serverNumber
date
items       // a list of MenuItem objects
totalPrice
tip
```

3. Define Each Class's Behaviors.

Once the class's attributes have been defined, the programmer must identify the activities, or *behaviors,* each class must be capable of performing. For example, some of the the activities the `MenuItem` class should be able to perform include

- changing a price
- displaying a price

Some of the activities the `CustomerOrder` class should be able to perform include

- accepting the information for a new order
- adding an item to an existing order
- returning any information on a previously stored order
- calculating the total price of all items on an order
- printing a list of ordered items for the kitchen
- printing a bill for the patron.

In C++, a class's behaviors are its *member functions*.

4. Define the Relationships Between Classes.

The last step in our object-oriented analysis phase is to define the relationships that exist between and among the classes in a program. The possible relationships may be formally stated as

- Access
- Ownership (Composition)
- Inheritance.

Informally, these three relationships can be described as

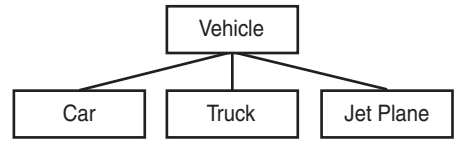
- Uses-a
- Has-a
- Is-a.

The first relationship, access, allows an object to modify the attributes of another object. Normally, an object has attributes not accessible to parts of the program outside the object. These are known as private attributes. An access relationship between two objects means that one object will have access to the other object's private attributes. When this relationship exists, it can be said that one object *uses* the other.

The second relationship, ownership, means that one object has another object as one of its members. For example, in our restaurant example, the `CustomerOrder` class has a list of `MenuItem` objects as one of its attributes. In OOP terminology, this type of relationship is also called *composition*.

The third relationship is inheritance. Sometimes a class is based on another class. This means that one class is a specialized case of the other. For example, consider a program that uses classes representing cars, trucks, and jet planes. Although those three types of classes in the real world are very different, they have many common characteristics: They are all modes of transportation, and they all carry some number of passengers. So each of the three classes could be based on a Vehicle class that has attributes and behaviors common to them all. This is illustrated in Figure 7-10.

Figure 7-10



In OOP terminology, the Vehicle class is the *base class* and the Car, Truck and Jet Plane classes are *derived* classes. All of the attributes and behaviors of the Vehicle class are inherited by the Car, Truck, and Jet Plane classes. The relationship implies that a car *is a* vehicle, a truck *is a* vehicle and a jet plane *is a* vehicle.

In addition to inheriting the attributes and behaviors of the base class, derived classes add their own. For example, the Car class might have attributes and behaviors that set and indicate the number of passengers it can carry, whether it is a sedan or a coupe, and the type of engine it has. The truck class might have attributes and behaviors that set and indicate the maximum amount of weight it can carry, and how many miles it can travel between refuelings. The Jet Plane class might have attributes and behaviors that set and indicate its altitude and heading. These added components of the derived classes make them more specialized than the base class.

These three types of relationships between classes, *access*, *ownership*, and *inheritance*, are discussed further in Chapter 11.

Once an enterprise and its operations have been analyzed, each class can be designed, and a set of programs can be developed to automate some of these operations.

Finding the Classes

Let's look further at step 1 in the analysis process: identifying the classes. Over the years, software professionals have developed numerous techniques for doing this, but they all involve identifying the different types of real-world objects present in the problem, so that classes can be created for them. One simple and popular technique involves the following steps:

1. Get a written description of the problem domain.
2. Identify all the nouns (including pronouns and noun phrases) in the description. Each of these is a potential class.
3. Refine the list to include only the classes that are relevant to the problem.

Let's take a closer look at each of these steps.

Write a Description of the Problem Domain

The *problem domain* is the set of real-world objects, parties, and major events related to the problem. If you adequately understand the nature of the problem you are trying to solve, you can write a description of the problem domain yourself. If you do not thoroughly understand the nature of the problem, you should have an expert write the description for you.

For example, suppose we are programming an application that the manager of Joe's Automotive Shop will use to print service quotes for customers. Here is a description that an expert, perhaps Joe himself, might have written:

Joe's Automotive Shop services foreign cars and specializes in servicing cars made by Mercedes, Porsche, and BMW. When a customer brings a car to the shop, the manager gets the customer's name, address, and telephone number. The manager then determines the make, model, and year of the car, and gives the customer a service quote. The service quote shows the estimated parts charges, estimated labor charges, sales tax, and total estimated charges.

The problem domain description should include any of the following:

- Physical objects such as vehicles, machines, or products
- Any role played by a person, such as manager, employee, customer, teacher or student.
- The results of a business event, such as a customer order, or in this case a service quote
- Record-keeping items, such as customer histories and payroll records.

Identify All of the Nouns

The next step is to identify all of the nouns and noun phrases. (If the description contains pronouns, include them too.) Here's another look at the previous problem domain description. This time the nouns and noun phrases appear in bold.

> **Joe's Automotive Shop** services **foreign cars,** and specializes in servicing **cars** made by **Mercedes, Porsche,** and **BMW.** When a **customer** brings a **car** to the **shop,** the **manager** gets the **customer's name, address,** and **telephone number**. The **manager** then determines the **make, model,** and **year** of the **car,** and gives the **customer** a **service quote.** The **service quote** shows the **estimated parts charges, estimated labor charges, sales tax,** and **total estimated charges.**

Notice that some of the nouns are repeated. The following lists all of the nouns without duplicating any of them.

address	foreign cars	Porsche
BMW	Joe's Automotive Shop	sales tax
car	make	service quote
cars	manager	shop
customer	Mercedes	telephone number
estimated labor charges	model	total estimated charges
estimated parts charges	name	year

Refine the List of Nouns

The nouns that appear in the problem description are merely candidates to become classes. It might not be necessary to make classes for them all. The next step is to refine the list to include only the classes that are necessary to solve the particular problem at hand. We will look at the common reasons that a noun can be eliminated from the list of potential classes.

Some of the nouns really mean the same thing.

In this example, the following sets of nouns refer to the same thing:

- **cars** and **foreign cars** both refer to the general concept of a car.
- **Joe's Automotive Shop** and **shop** both refer to the same shop.

We can settle on a single class for each of these. In this example we will arbitrarily eliminate **foreign cars** from the list, and use the word **cars**. Likewise we will eliminate **Joe's Automotive Shop** from the list and use the word **shop**. The updated list of potential classes is:

address	~~foreign cars~~	Porsche
BMW	~~Joe's Automotive Shop~~	sales tax
car	make	service quote
cars	manager	shop
customer	Mercedes	telephone number
estimated labor charges	model	total estimated charges
estimated parts charges	name	year

Some nouns might represent items that we do not need to be concerned with in order to solve the problem.

Next review the problem description and eliminate any unnecessary classes.

- We can cross **shop** off the list because our application only needs to be concerned with individual service quotes. It doesn't need to work with or determine any company-wide information. If the problem description asked us to keep a total of all the service quotes, then it would make sense to have a class for the shop.
- We will also not need a class for the **manager** because the problem statement does not ask us to process any information about the manager. If there were multiple shop managers, and the problem description asked us to record which manager wrote each service quote, it would make sense to have a class for the manager.

The updated list of potential classes at this point is:

address	~~foreign cars~~	Porsche
BMW	~~Joe's Automotive Shop~~	sales tax
car	make	service quote
cars	~~manager~~	~~shop~~
customer	Mercedes	telephone number
estimated labor charges	model	total estimated charges
estimated parts charges	name	year

Some of the nouns might represent objects, not classes.

We can eliminate **Mercedes, Porsche,** and **BMW** as classes because, in this example, they all represent specific cars, and can be considered instances of a **cars** class. Also, we can eliminate the word **car** from the list. In the description it refers to a specific car brought to the shop by a customer. Therefore, it would also represent an instance of a **cars** class. At this point the updated list of potential classes is:

address	~~foreign cars~~	~~Porsche~~
~~BMW~~	~~Joe's Automotive Shop~~	sales tax
~~car~~	make	service quote
cars	~~manager~~	~~shop~~
customer	~~Mercedes~~	telephone number
estimated labor charges	model	total estimated charges
estimated parts charges	name	year



NOTE: Some object-oriented designers take note of whether a noun is plural or singular. Sometimes a plural noun will indicate a class and a singular noun will indicate an object.

Some of the nouns might represent simple values that can be stored in a variable and do not require a class.

Remember, a class contains attributes and member functions. Attributes are related items, stored within an object of the class, that define the object's state. Member functions are actions or behaviors that may be performed by an object of the class. If a noun represents a type of item that would not have any identifiable attributes or member functions, then it can probably be eliminated from the list. To help determine whether a noun represents an item that would have attributes and member functions, ask the following questions about it:

- Would you use a group of related values to represent the item's state?
- Are there any obvious actions to be performed by the item?

If the answers to both of these questions are no, then the noun probably represents a value that can be stored in a simple variable. If we apply this test to each of the nouns that remain in our list, we can conclude that the following are probably not classes: **address, estimated labor charges, estimated parts charges, make, model, name, sales tax, telephone number, total estimated charges** and **year.** These are all simple string or numeric values that can be stored in variables. Here is the updated list of potential classes:

~~address~~	~~foreign cars~~	~~Porsche~~
~~BMW~~	~~Joe's Automotive Shop~~	~~sales tax~~
~~car~~	~~make~~	service quote
cars	~~manager~~	~~shop~~
customer	~~Mercedes~~	~~telephone number~~
~~estimated labor charges~~	~~model~~	~~total estimated charges~~
~~estimated parts charges~~	~~name~~	~~year~~

As you can see from the list, we have eliminated everything except **cars, customer,** and **service quote.** This means that in our application, we will need classes to represent cars, customers, and service quotes. Ultimately, we will write a `Car` class, a `Customer` class, and a `ServiceQuote` class.

Identifying Class Responsibilities

Once the classes have been identified, the next task is to identify each class's responsibilities. Class *responsibilities* are

- the things that the class is responsible for knowing
- the actions that the class is responsible for doing

When you have identified the things that a class is responsible for knowing, then you have identified the class's attributes. Likewise, when you have identified the actions that a class is responsible for doing, you have identified its member functions.

It is often helpful to ask the questions "In the context of this problem, what must the class know? What must the class do?" The first place to look for the answers is in the description of the problem domain. Many of the things that a class must know and do will be mentioned. Some class responsibilities, however, might not be directly mentioned in the problem domain, so additional analysis is often required. Let's apply this methodology to the classes we previously identified from our problem domain.

The `Customer` Class

In the context of our problem domain, what must the `Customer` class know? The description directly mentions the following items, which are all attributes of a customer:

- the customer's name
- the customer's address
- the customer's telephone number.

These are all values that can be represented as strings and stored in the class's member variables. The `Customer` class can potentially know many other things. One mistake that can be made at this point is to identify too many things that an object is responsible for knowing. In some applications, a `Customer` class might know the customer's email address. This particular problem domain does not mention that the customer's email address is used for any purpose, so we should not include it as an attribute.

Now let's identify the class's member functions. In the context of our problem domain, what must the `Customer` class do? The only obvious actions are:

- create an object of the `Customer` class
- set and get the customer's name
- set and get the customer's address
- set and get the customer's telephone number.

From this list we can see that the `Customer` class will need a constructor, as well as accessor and mutator functions for each of its attributes.

The `Car` Class

In the context of our problem domain, what must an object of the `Car` class know? The following items are all attributes of a car, and are mentioned in the problem domain:

- the car's make
- the car's model
- the car's year.

Now let's identify the class's member functions. In the context of our problem domain, what must the `Car` class do? Once again, the only obvious actions are the standard set of member functions that we will find in most classes (constructors, accessors, and mutators). Specifically, the actions are:

- create an object of the `Car` class
- set and get the car's make
- set and get the car's model
- set and get the car's year.

The `ServiceQuote` Class

In the context of our problem domain, what must an object of the `ServiceQuote` class know? The problem domain mentions the following items:

- the estimated parts charges
- the estimated labor charges
- the sales tax
- the total estimated charges.

Careful thought will reveal that two of these items are the results of calculations: sales tax and total estimated charges. These items are dependent on the values of the estimated parts and labor charges. In order to avoid the risk of holding stale data, we will not store these values in member variables. Rather, we will provide member functions that calculate these values and return them. The other member functions that we will need for this class are a constructor and the accessors and mutators for the estimated parts charges and estimated labor charges attributes.

Object Reusability

We have mentioned several advantages offered by object-oriented programming. Still another is *object reusability*. A class is not a stand-alone program. It is a mechanism for creating objects used by programs that need its service. Ideally, a class created for use in one program can be made general enough to be used by other programs as well. For example, the `Customer` class can be designed to create objects used by many different applications that have customers. The `Car` class can be designed to create objects used by many different programs that involve vehicles.

Object-Oriented vs. Object-Based Programming

Although classes and objects form the basis of object-oriented programming, by themselves they are not sufficient to constitute true object-oriented programming. Using them might more correctly be referred to as object-based programming. When we add the ability to define relationships among different classes of objects, to create classes of objects from other classes (inheritance) and to determine the behavior of a member function depending on which object calls it (polymorphism), it becomes true object-oriented programming. You will learn about these more advanced object-oriented programming features later in the book.



Checkpoint

7.35 What is a problem domain?

7.36 When designing an object-oriented application, who should write a description of the problem domain?

7.37 How do you identify the potential classes in a problem domain description?

7.38 What two questions should you ask to determine a class's responsibilities?

7.39 Look at the following description of a problem domain:

> A doctor sees patients in her practice. When a patient comes to the practice, the doctor performs one or more procedures on the patient. Each procedure that the doctor performs has a description and a standard fee. As the patient leaves the practice, he or she receives a statement from the office manager. The statement shows the patient's name and address, as well as the procedures that were performed, and the total charge for the procedures.

Assume that you are writing an application to generate a statement that can be printed and given to the patient.

A) Identify all of the potential classes in this problem domain.
B) Refine the list to include only the necessary class or classes for this problem.
C) Identify the responsibilities of the class or classes that you identified in step B.

7.16 Screen Control

CONCEPT: **Operating system functions allow you to control how output appears on the console screen.**

Positioning the Cursor on the Screen

In Chapter 5's Tying It All Together section you learned that C++ compilers provide special libraries for calling on operating system functions. So far, in Chapters 5 and 6, we have used the Windows `SetConsoleTextAttribute` function to display screen output in color. Now we will look at a Windows operating system function for positioning the cursor on the screen. This function is `SetConsoleCursorPosition`.



NOTE: Recall from Chapter 5 that operating system functions are tailored to specific operating systems. So programs that use them will only run on the system for which they were written. The functions described here work with Windows 2000, XP, Vista, and 7. If you are using Linux or Mac OS, your instructor may be able to provide you with similar functions that work on those systems.

Until now, all the programs you have created display output beginning on the top line of the screen. They then move down the screen, one line at a time, when the user presses the [Enter] key or when the program outputs an `endl` or `"\n"`. But what if you are writing on the fifth row of the screen and want to go back to the second row? Or what if you want to display something in the very middle of the screen? You can do these things on a Windows system by using the `SetConsoleCursorPosition` function to move the cursor to the desired location before writing the output.

To use this function, you will need to do the same two things you did in Chapters 5 and 6 to use color. You must

- `#include <windows.h>` in your program.
- Create a *handle* to the standard output screen by including the following definition in your program.

```
HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);
```

A typical text screen has 25 rows, or lines, with 80 print positions per row. Each of these positions is called a cell. A *cell* is a little block that can display a single character and it is identified by its row number and its position on that row. The rows range from 0 to 24, with 0 being the top row of the screen. The print positions on each row, usually referred to as columns, range from 0 to 79, with 0 being at the far left-hand side. The row and column of a cell, which identifies its location on the screen, are called its *coordinates*.

To place the cursor in a specific screen cell, you must specify its cell coordinates by setting two variables in a `COORD` structure that is already defined in Windows. This structure has two member variables named `X` and `Y`, with `X` holding the column and `Y` holding the row. Here is what the structure looks like.

```
struct COORD
{
   short int X;  // Column position
   short int Y;  // Row position
};
```

Here is how you use it. The following code segment writes the word `Hello` centered on the standard output screen.

```
HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);
COORD position;    // position is a COORD structure

position.X = 38;   // Set column near screen center
position.Y = 11;   // Set row near screen center
                   // Place cursor there, then print
SetConsoleCursorPosition(screen, position);
cout << "Hello" << endl;
```



NOTE: When you set a screen position, you must follow all output that your program writes there with an `endl`. This is necessary to ensure that the output is actually displayed at this location. If you do not use an `endl`, the output may be buffered and written to the screen much later, after the cursor position has changed. Following your output with the new line character `'\n'` does not work because it does not flush the screen buffer like `endl` does.

Program 7.19 positions the cursor to display a set of nested boxes near the center of the screen. Notice that it uses the `Sleep` function, previously seen in Chapter 5 and Chapter 6's Tying it All Together programs. This function pauses the program execution for part of a second so things do not happen too fast for the user to see them. The argument passed to the function tells it how many milliseconds it should pause. A *millisecond* is a thousandth of a second. So, for example, to pause execution of a program for a half second the following function call would work.

```
Sleep(500);
```

Program 7.19 uses the command `Sleep(750)` to pause the program execution for $\frac{3}{4}$ of a second after each box displays.

Program 7-19

```
1 // This program demonstrates the use of Windows functions
2 // for positioning the cursor. It displays a series of nested
3 // boxes near the center of the screen.
4 #include <iostream>
5 #include <windows.h>    // Needed to set cursor positions & call Sleep
6 using namespace std;
7
```

(program continues)

Program 7-19 *(continued)*

```
 8 void placeCursor(HANDLE, int, int);   // Function prototypes
 9 void printStars(int);
10
11 int main()
12 {
13    const int midRow = 12,
14              midCol = 40,
15              numBoxes = 3;
16    int width, startRow, endRow;
17
18    // Get the handle to standard output device (the console)
19     HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);
20
21    // Each loop prints one box
22    for (int box = 1, height = 1; box <= numBoxes; box++, height+=2)
23    {  startRow = midRow - box;
24       endRow = midRow + box;
25       width = box*5 + (box+1)%2;  // Adds 1 if box*5 is an even number
26
27       // Draw box top
28       placeCursor(screen, startRow, midCol-width/2);
29       printStars(width);
30
31       // Print box sides
32       for (int sideRow = 1; sideRow <= height; sideRow++)
33       {  placeCursor(screen, startRow + sideRow, midCol-width/2);
34          cout << '*' << endl;
35          placeCursor(screen, startRow + sideRow, midCol+width/2);
36          cout << '*' << endl;
37       }
38       // Draw box bottom
39       placeCursor(screen, endRow, midCol-width/2);
40       printStars(width);
41
42       Sleep(750);   // Pause 3/4 second between boxes displayed
43    }
44
45    placeCursor(screen, 20, 0);   // Move cursor out of the way
46    return 0;
47 }
48
49 /*************************************************
50  *                      placeCursor              *
51  *************************************************/
52 void placeCursor(HANDLE screen, int row, int col)
53 {                     // COORD is a defined C++ structure that
54    COORD position;    // holds a pair of X and Y coordinates
55    position.Y = row;
56    position.X = col;
57    SetConsoleCursorPosition(screen, position);
58 }
59
```

(program continues)

Program 7-19 *(continued)*

```
60 /*****************************************************
61  *                      printStars                   *
62  *****************************************************/
63 void printStars(int numStars)
64 {
65    for (int star = 1; star <= numStars; star++)
66       cout << '*';
67    cout << endl;
68 }
```

Program Output

```
***************
* *********** *
* *  *****  * *
* *  *   *  * *
* *  *****  * *
* *********** *
***************
```

Creating a Screen Input Form

Program 7-19 is fun to run, but Program 7-20 demonstrates a more practical application of positioning the cursor on the screen. Instead of prompting the user to input a series of entries one prompt at a time, we can design a *screen input form*. This more professional looking way of getting input from the user involves creating and displaying a screen that shows all the prompts at once. The cursor is then placed beside a particular prompt the user is expected to respond to. When the user enters the data for this prompt and presses [Enter], the cursor moves to the next prompt.

Program 7-20

```
1  // This program uses a screen input form to get information
2  // from the user.
3  #include <iostream>
4  #include <windows.h>      // Needed to set cursor positions
5  #include <string>
6  using namespace std;
7
8  struct UserInfo
9  {  string name;
10    int age;
11    char gender;
12 };
13
14 void placeCursor(HANDLE, int, int);   // Function prototypes
15 void displayPrompts(HANDLE);
16 void getUserInput(HANDLE, userInfo&);
17 void displayData (HANDLE, userInfo);
18
```

(program continues)

Program 7-20 *(continued)*

```
19 int main()
20 {
21    userInfo input;           // input is a UserInfo structure
22                              // that has 3 member variables
23
24    // Get the handle to standard output device (the console)
25    HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);
26
27    displayPrompts(screen);
28    getUserInput(screen, input);
29    displayData (screen, input);
30
31    return 0;
32 }
33
34 /*****************************************************
35  *                    placeCursor                    *
36  *****************************************************/
37 void placeCursor(HANDLE screen, int row, int col)
38 {                      // COORD is a defined C++ structure that
39    COORD position;     // holds a pair of X and Y coordinates
40    position.Y = row;
41    position.X = col;
42    SetConsoleCursorPosition(screen, position);
43 }
44
45 /*****************************************************
46  *                   displayPrompts                  *
47  *****************************************************/
48 void displayPrompts(HANDLE screen)
49 {
50    placeCursor(screen, 3, 25);
51    cout << "******* Data Entry Form *******" << endl;
52    placeCursor(screen, 5, 25);
53    cout << "Name: " << endl;
54    placeCursor(screen, 7, 25);
55    cout << "Age:            Gender (M/F): " << endl;
56 }
57
58 /*****************************************************
59  *                    getUserInput                   *
60  *****************************************************/
61 void getUserInput(HANDLE screen, userInfo &input)
62 {
63    placeCursor(screen, 5, 31);
64    getline(cin, input.name);
65    placeCursor(screen, 7, 30);
66    cin >> input.age;
67    placeCursor(screen, 7, 55);
68    cin >> input.gender;
69 }
```

(program continues)

Program 7-20 *(continued)*

```
70
71 /*************************************************
72  *                    displayData                *
73  *************************************************/
74 void displayData(HANDLE screen, userInfo input)
75 {
76    placeCursor(screen, 10, 0);
77    cout << "Here is the data you entered.\n";
78    cout << "Name  : " << input.name   << endl;
79    cout << "Age   : " << input.age    << endl;
80    cout << "Gender: " << input.gender << endl;
81 }
```

Initial Screen Display

```
                ******* Data Entry Form *******

                Name:

                Age:            Gender (M/F):
```

Program Output with Example Input Shown in Bold

```
                 ******* Data Entry Form *******

                 Name: Mary Beth Jones[Enter]

                 Age: 19[Enter]  Gender (M/F): F[Enter]

Here is the data you entered.
Name  : Mary Beth Jones
Age   : 19
Gender: F
```

7.17 Tying It All Together: *Yoyo Animation*

With what you have learned in this chapter you can now create simple *text-based graphics*. To do that, simply arrange characters in different patterns to form images on the screen. Then *animate* those images, giving the illusion of motion, by erasing them from their old position and redisplaying them somewhere else on the screen. To erase a character from the screen simply write a blank " " on top of it.

Program 7-21 uses Windows operating system functions to simulate a yoyo unwinding and then winding back up. The `Sleep` function is used to pause execution between moves, so that the user can watch the motion taking place.

Program 7-21

```
// This program creates a simple animation using Windows
// functions to simulate a yoyo moving down and up.
#include <iostream>
#include <windows.h>   // Needed to set cursor positions
using namespace std;

int main()
{
   HANDLE screen = GetStdHandle(STD_OUTPUT_HANDLE);
   COORD pos = {40, 3};    // Start position
   SetConsoleCursorPosition(screen, pos);
   cout << "O" << endl;
   Sleep(500);

   // Watch the yoyo go down & back up 3 times
   for (int tossIt = 1; tossIt <= 3; tossIt++)
   {
      // Yoyo unwinds
      while (pos.Y <= 20)  // pos.Y is the row
      {
         // Move the yoyo down 1 position and then pause
         SetConsoleCursorPosition(screen, pos);
         cout << "|" << endl;
         pos.Y++;
         SetConsoleCursorPosition(screen, pos);
         cout << "O" << endl;
         Sleep(100);
       }

       // Yoyo winds back up
       while (pos.Y > 3)
       {
          // Erase character at current position
          // Move yoyo up one position, then pause
          SetConsoleCursorPosition(screen, pos);
          cout << " " << endl;
          pos.Y --;
          SetConsoleCursorPosition(screen, pos);
          cout << "O" << endl;
          Sleep(100);
       }
    }
    return 0;
}
```

You will need to run the program to see the animation as the yoyo unwinds and then winds back up on its string.

:
:
:
:
O

Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. What does ADT stand for?
2. Which of the following must a programmer know about an ADT to use it?
   A) What values it can hold
   B) What operations it can perform
   C) How the operations are implemented
3. The two common programming methods in practice today are _____ and _____.
4. _____ programming is centered around functions, or procedures, whereas _____ programming is centered around objects.
5. An object is a software entity that combines both _____ and _____in a single unit.
6. An object is a(n) _____ of a class.
7. Creating a class object is often called the _____ of a class.
8. Once a class is declared, how many objects can be created from it?
   A) 1
   B) 2
   C) Many
9. An object's data items are stored in its _____.
10. The procedures, or functions, an object performs are called its _____.
11. Bundling together an object's data and procedures is called _____.
12. An object's members can be declared `public` or `private`.
    A public member can be accessed by _____.
    A private member can be accessed by _____.
13. Normally a class's _____ are declared to be private and its _____ are declared to be public.
14. A class member function that uses, but does not change, the value of a member variable is called a(n) _____.
15. A class member function that changes the value of a member variable is called a(n) _____.
16. When a member function's body is written inside a class declaration, the function is a(n) _____ function.
17. A class constructor is a member function with the same name as the _____.
18. A constructor is automatically called when an object is _____.
19. Constructors cannot have a(n) _____ type.
20. A(n) _____ constructor is one that requires no arguments.

21. A destructor is a member function that is automatically called when an object is _____.
22. A destructor has the same name as the class, but is preceded by a(n) _____ character.
23. A constructor whose parameters all have default values is a(n) _____ constructor.
24. A class may have more than one constructor, as long as each has a different _____.
25. A class may only have one default _____ and one _____.
26. In general it is considered good practice to have member functions avoid doing _____.
27. When a member function forms part of the interface through which a client program can use the class, the function must be _____.
28. When a member function performs a task internal to the class and should not be called by a client program, the function should be made _____.
29. True or false: A class object can be passed to a function, but cannot be returned by a function.
30. True or false: C++ class objects are always passed to functions by reference.
31. It is considered good programming practice to store the declaration for a class, its function definitions, and the client program that uses the class in _____ files.
32. If you were writing a class declaration for a class named `Canine` and wanted to place it in its own file, what should you name the file? _____
33. If you were writing the definitions for the `Canine` class member functions and wanted to place these in their own file, what should you name the file? _____
34. A structure is like a class, but normally only contains member variables and no _____.
35. By default, are the members of a structure public or private? _____
36. Before a structure variable can be created, the structure must be _____.
37. When a structure variable is created its members can be initialized with either a(n) _____ or a(n) _____.
38. The _____ operator is used to access structure members.
39. An `Inventory` structure is declared as follows:

```
struct Inventory
{
   int itemCode;
   int qtyOnHand;
};
```

Write a definition statement that creates an `Inventory` variable named `trivet` and initializes it with an initialization list so that its code is 555 and its quantity is 110.

40. A `Car` structure is declared as follows:

```
struct Car
{
   string make,
          model;
   int    year;
   double cost;

   Car(string mk, string md, int y, double c)
   {  make = mk; model = md; year = y; cost = c; }
};
```

Write a definition statement that defines a `Car` structure variable initialized with the following information:

Make: Ford	Model: Mustang
Year: 2010	Cost: $22,495

41. Declare a structure named `TempScale`, with the following members:

`fahrenheit:` a `double`
`celsius:` a `double`

Next, declare a structure named `Reading`, with the following members:

`windSpeed:` an `int`
`humidity:` a `double`
`temperature:` a `TempScale` structure variable

Next, define a `Reading` structure variable named `today`.

Now write statements that will store the following data in the `Reading` variable.

Wind speed: 37 mph
Humidity: 32%
Fahrenheit temperature: 32 degrees
Celsius temperature: 0 degrees

42. Write a function called `showReading`. It should have a parameter that accepts a `Reading` structure variable (see question 41) and should display the values of the structure's member variables on the screen.

43. Write a function called `inputReading` that has a parameter to accept a `Reading` structure reference variable (see question 41). The function should ask the user to enter values for each member of the structure.

44. Write a function called `getReading`, which returns a `Reading` structure (see question 41). The function should ask the user to enter values for each member of a `Reading` structure, and then return the structure.

45. Write the declaration of a union called `Items` with the following members. Then define an `Items` union variable named `anItem`.

```
alpha: a character          // 1 byte
num: an integer             // 4 bytes
bigNum: a long integer      // 4 bytes
real: a double              // 8 bytes
```

46. How many bytes of memory will be allocated for `anItem`?

Algorithm Workbench

47. Assume a class named `Inventory` keeps track of products in stock for a company. It has member variables `prodID`, `prodDescription`, and `qtyInStock`. Write a constructor that initializes a new `Inventory` object with the values passed as arguments, but which also includes a reasonable default value for each parameter.

48. Write a `remove` member function that accepts an argument for a number of units and removes that number of units of an item from inventory. If the operation is completed successfully it should return the number of units remaining in stock for that item. However, if the number of units passed to the function is less than the number of units in stock, it should not make the removal and should return –1 as an error signal.

Find the Errors

Each of the following declarations, programs, and program segments has errors. Locate as many as you can.

49. A)
```
struct
{  int x;
   double y;
};
```
B)
```
struct Values
{  string name;
   int age;
}
```

50. A)
```
struct TwoVals
{
   int a, b;
};
int main()
{
   TwoVals.a = 10;
   TwoVals.b = 20;
   return 0;
}
```
B)
```
#include <iostream>
using namespace std;

struct ThreeVals
{
   int a, b, c;
   void ThreeVals()
   {a = 1; b = 2; c = 3;}
};
int main()
{
   ThreeVals vals;
   cout << vals << endl;
   return 0;
}
```

51. A)
```
struct Names
{  string first;
   string last;
};
int main()
{
   Names customer ("Smith", "Orley");
   cout << Names.first << endl;
   cout << Names.last << endl;
   return 0;
}
```

B)
```
struct TwoVals
{
   int a = 5;
   int b = 10;
};

int main()
{
   TwoVals v;
   cout << v.a << " " << v.b;
   return 0;
}
```

52. A)
```
class Circle:
{
private
   double centerX;
   double centerY;
   double radius;
public
   setCenter(double, double);
   setRadius(double);
}
```

B)
```
#include <iostream>
using namespace std;
Class Moon;
{
Private;
   double earthWeight;
   double moonWeight;
Public;
   moonWeight(double ew);// Constructor
   { earthWeight = ew; moonWeight = earthWeight / 6; }
   double getMoonWeight();
   { return moonWeight; }
}
```

```
int main()
{
   double earth;

   cout >> "What is your weight? ";
   cin  << earth;
   Moon lunar(earth);
   cout << "On the moon you would weigh "
        <<lunar.getMoonWeight() << endl;
   return 0;
}
```

53. A)
```
#include <iostream>
using namespace std;

class DumbBell;
{
   int weight;
public:
   void setWeight(int);
};
void setWeight(int w)
{  weight = w; }

int main()
{
   DumBell bar;

   DumbBell.setWeight(200);
   cout << "The weight is " << bar.weight << endl;
   return 0;
}
```

B)
```
class Change
{
private:
   int pennies;
   int nickels;
   int dimes;
   int quarters;
   Change()
      { pennies = nickels = dimes = quarters = 0; }
   Change(int p = 100, int n = 50, d = 50, q = 25);
};
void Change::Change(int p, int n, d, q)
{
   pennies = p;
   nickels = n;
   dimes = d;
   quarters = q;
}
```

54. If the items on the following list appeared in a problem domain description, which would be potential classes?

Animal	Medication	Nurse
Inoculate	Operate	Advertise
Doctor	Invoice	Measure
Patient	Client	Customer

55. Look at the following description of a problem domain:

    The bank offers the following types of accounts to its customers: savings accounts, checking accounts, and money market accounts. Customers are allowed to deposit money into an account (thereby increasing its balance), withdraw money from an account (thereby decreasing its balance), and earn interest on the account. Each account has an interest rate.

    Assume that you are writing an application that will calculate the amount of interest earned for a bank account.

    A) Identify the potential classes in this problem domain.
    B) Refine the list to include only the necessary class or classes for this problem.
    C) Identify the responsibilities of the class or classes.

Soft Skills

Working in a team can often help individuals better understand new ideas related to programming. Others can explain things that you do not understand. Also, you will find that by explaining something to someone else, you actually understand it better.

56. Write down one question you have about the object-oriented programming material from Chapter 7. For example, you could mention something you want explained about how classes are designed and created, about how objects are related to classes, or about how overloaded constructors work. Then form a group with three to four other students. Each person in the group should participate in answering the questions posed by the other members of the group.

Programming Challenges



1. Movie Data

Write a program that uses a structure named `MovieData` to store the following information about a movie:

Title
Director
Year Released
Running time (in minutes)

Include a constructor that allows all four of these member data values to be specified at the time a `MovieData` variable is created. The program should create two `MovieData` variables and pass each one in turn to a function that displays the information about the movie in a clearly formatted manner.

2. Movie Profit

Modify the Movie Data program written for Programming Challenge 1 to include two additional members that hold the movie's production costs and first-year revenues. The constructor should be modified so that all six member values can be specified when a `MovieData` variable is created. Modify the function that displays the movie data to display the title, director, release year, running time, and first year's profit or loss.

3. Corporate Sales Data

Write a program that uses a structure named `CorpData` to store the following information on a company division:

Division name (such as East, West, North, or South)
First quarter sales
Second quarter sales
Third quarter sales
Fourth quarter sales
Total annual sales
Average quarterly sales.

Include a constructor that allows the division name and four quarterly sales amounts to be specified at the time a `CorpData` variable is created.

The program should create four variables of this structure, each representing one of the following corporate divisions: East, West, North, and South. Each variable should be passed in turn to a function that calculates and stores the total sales and average quarterly sales for that division. Once this has been done for each division, each variable should be passed in turn to a function that displays the division name, total sales, and quarterly average.

4. Monthly Budget

A student has established the following monthly budget:

Housing	500.00
Utilities	150.00
Household expenses	65.00
Transportation	50.00
Food	250.00
Medical	30.00
Insurance	100.00
Entertainment	150.00
Clothing	75.00
Miscellaneous	50.00

Write a program that declares a `MonthlyBudget` structure with member variables to hold each of these expense categories. The program should create two `MonthlyBudget` structure variables. The first will hold the budget figures given above. The second will be passed to a function that has the user enter the amounts actually spent in each budget category during the past month. The program should then pass both structure variables to a function that displays a report indicating the amount over or under budget the student spent in each category, as well as the amount over or under for the entire monthly budget.

5. Ups and Downs

Write a program that displays the word UP on the bottom line of the screen a couple of inches to the left of center and displays the word DOWN on the top line of the screen a couple of inches to the right of center. Moving about once a second, move the word UP up a line and the word DOWN down a line until UP disappears at the top of the screen and DOWN disappears at the bottom of the screen.

6. Wrapping Ups and Downs

Modify the program you wrote for Programming Challenge 5, so that after disappearing off of the screen, the word UP reappears at the bottom of the screen and the word DOWN reappears at the top of the screen. The words should continue moving until the user presses the [Enter] key to terminate the program.

7. Left and Right

Modify the program you wrote for Programming Challenge 5 to display the words LEFT (starting at the right-hand side of the screen a row or two down from the middle) and RIGHT (starting at the left-hand side of the screen a row or two up from the middle). Moving about 6 moves per second, move LEFT to the left and RIGHT to the right until both words disappear off the screen.

8. Numeric Input Validation Class

Use the `CharRange` class presented in this chapter as a model for an `IntRange` class. The `IntRange` class should allow the user to enter integer values within a specified range. An error message should be displayed when the user enters a value outside the range. Demonstrate the class by writing a simple program that uses it.



9. Date

Design a class called `Date` that has integer data members to store month, day, and year. The class should have a three-parameter default constructor that allows the date to be set at the time a new `Date` object is created. If the user creates a `Date` object without passing any arguments, or if any of the values passed are invalid, the default values of 1, 1, 2001 (i.e., January 1, 2001) should be used. The class should have member functions to print the date in the following formats:

```
3/15/10
March 15, 2010
15 March 2010
```

Demonstrate the class by writing a program that uses it.

Input Validation: Only accept values between 1 and 12 for the month, between 1 and 31 for the day, and between 1950 and 2020 for the year.

10. Report Heading

Design a class called `Heading` that has data members to hold the company name and the report name. A two-parameter default constructor should allow these to be specified at the time a new `Heading` object is created. If the user creates a `Heading` object without passing any arguments, "`ABC Industries`" should be used as a default value for the company name and "`Report`" should be used as a default for the report name. The class should have member functions to print a heading in either one-line format, as shown here:

```
Pet Pals Payroll Report
```

or in four-line "boxed" format, as shown here:

```
****************************************
                Pet Pals
             Payroll Report
****************************************
```

Try to figure out a way to center the headings on the screen, based on their lengths. Demonstrate the class by writing a simple program that uses it.

11. Widget Factory

Design a class for a widget manufacturing plant. Assuming that 10 widgets may be produced each hour, the class object will calculate how many days it will take to produce any number of widgets. (The plant operates two shifts of eight hours each per day.) Write a program that asks the user for the number of widgets that have been ordered and then displays the number of days it will take to produce them.

Input Validation: Do not accept negative values for the number of widgets ordered.



VideoNote
Solving the `Car` Class Problem

12. `Car` Class

Write a class named `Car` that has the following member variables:

- **`year`.** An `int` that holds the car's model year.
- **`make`.** A `string` that holds the make of the car.
- **`speed`.** An `int` that holds the car's current speed.

In addition, the class should have the following member functions.

- **`Constructor`.** The constructor should accept the car's `year` and make as arguments and assign these values to the object's `year` and `make` member variables. The constructor should initialize the `speed` member variable to 0.
- **`Accessors`.** Appropriate accessor functions should be created to allow values to be retrieved from an object's `year`, `make`, and `speed` member variables.
- **`accelerate`.** The `accelerate` function should add 5 to the `speed` member variable each time it is called.
- **`brake`.** The `brake` function should subtract 5 from the `speed` member variable each time it is called.

Demonstrate the class in a program that creates a `Car` object, and then calls the `accelerate` function five times. After each call to the `accelerate` function, get the current speed of the car and display it. Then, call the `brake` function five times. After each call to the `brake` function, get the current speed of the car and display it.



13. Population

In a population, the birth rate and death rate are calculated as follows:

Birth Rate = Number of Births ÷ Population
Death Rate = Number of Deaths ÷ Population

For example, in a population of 100,000 that has 8,000 births and 6,000 deaths per year,

Birth Rate = 8,000 ÷ 100,000 = 0.08
Death Rate = 6,000 ÷ 100,000 = 0.06

Design a `Population` class that stores a current population, annual number of births, and annual number of deaths for some geographic area. The class should allow these three values to be set in either of two ways: by passing arguments to a three-parameter constructor when a new `Population` object is created or by calling the `setPopulation`, `setBirths`, and `setDeaths` class member functions. The class should also have `getBirthRate` and `getDeathRate` functions that compute and return the birth and death rates. Write a short program that uses the `Population` class and illustrates its capabilities.

Input Validation: If a population figure less than 2 is passed to the class, use a default value of 2. If a birth or death figure less than 0 is passed in, use a default value of 0.

14. Gratuity Calculator

Design a `Tips` class that calculates the gratuity on a restaurant meal. Its only class member variable, `taxRate`, should be set by a one-parameter constructor to whatever rate is passed to it when a `Tips` object is created. If no argument is passed, a default tax rate of .065 should be used. The class should have just one public function, `computeTip`. This function needs to accept two arguments, the total bill amount and the tip rate. It should use this information to compute the meal cost before any tax was added. It should then apply the tip rate to just the meal cost portion of the bill to compute and return the tip amount. Demonstrate the class by creating a program that creates a single `Tips` object, then loops multiple times to allow the program user to retrieve the correct tip amount using various bill totals and desired tip rates.

Input Validation: If a tax rate of less than 0 is passed to the constructor, use the default rate of .065. Do not allow the total bill or the tip rate to be less than 0.

15. Inventory Class

Design an `Inventory` class that can hold information for an item in a retail store's inventory. The class should have the following private member variables.

Variable Name	Description
`itemNumber`	An `int` that holds the item's number.
`quantity`	An `int` that holds the quantity of the item onhand.
`cost`	A `double` that holds the wholesale per-unit cost of the item
`totalCost`	A `double` that holds the total inventory cost of the item (calculated as `quantity` times `cost`).

The class should have the following public member functions.

Member Function	Description
default constructor	Sets all the member variables to 0.
constructor #2	Accepts an item's number, quantity, and cost as arguments. The function should call other class functions to copy these values to the appropriate member variables and then call the `setTotalCost` function.
`setItemNumber`	Accepts an `int` argument that is copied to the `itemNumber` member variable.
`setQuantity`	Accepts an `int` argument that is copied to the `quantity` member variable.
`setCost`	Accepts a `double` argument that is copied to the `cost` member variable.
`setTotalCost`	Calculates the total inventory cost for the item (`quantity` times `cost`) and stores the result in `totalCost`.
`getItemNumber`	Returns the value in `itemNumber`.
`getQuantity`	Returns the value in `quantity`.
`getCost`	Returns the value in `cost`.
`getTotalCost`	Returns the value in `totalCost`.

Demonstrate the class by writing a simple program that uses it.

Input Validation: Do not accept negative values for item number, quantity, or cost.

Group Project

16. Patient Fees

This program should be designed and written by a team of students. Here are some suggestions:

- One or more students may work on a single class.
- The requirements of the program should be analyzed so each student is given about the same work load.
- The names, parameters, and return types of each function and class member function should be decided in advance.
- The program will be best implemented as a multifile program.

Write a program that computes a patient's bill for a hospital stay. The different components of the program are

- The `PatientAccount` class will keep a total of the patient's charges. It will also keep track of the number of days spent in the hospital. The group must decide on the hospital's daily rate.
- The `Surgery` class will have stored within it the charges for at least five types of surgery. It can update the `charges` variable of the `PatientAccount` class.
- The `Pharmacy` class will have stored within it the price of at least five types of medication. It can update the `charges` variable of the `PatientAccount` class.
- The `main` program.

The student who designs the main program will design a menu that allows the user to enter a type of surgery, enter one or more types of medication, and check the patient out of the hospital. When the patient checks out, the total charges should be displayed.

This page intentionally left blank

CHAPTER 8

Arrays

TOPICS

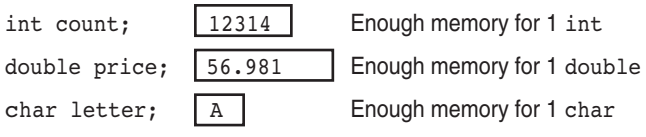
8.1 Arrays Hold Multiple Values
8.2 Accessing Array Elements
8.3 Inputting and Displaying Array Contents
8.4 Array Initialization
8.5 Processing Array Contents
8.6 Using Parallel Arrays
8.7 The `typedef` Statement
8.8 Arrays as Function Arguments
8.9 Two-Dimensional Arrays
8.10 Arrays with Three or More Dimensions
8.11 Vectors
8.12 Arrays of Class Objects
8.13 National Commerce Bank Case Study
8.14 Tying It All Together: *Rock, Paper, Scissors*

8.1 Arrays Hold Multiple Values

CONCEPT: **An array allows you to store and work with multiple values of the same data type.**

The variables you have worked with so far are designed to hold only one value at a time. Each of the variable definitions in Figure 8-1 cause only enough memory to be reserved to hold one value of the specified data type.

Figure 8-1

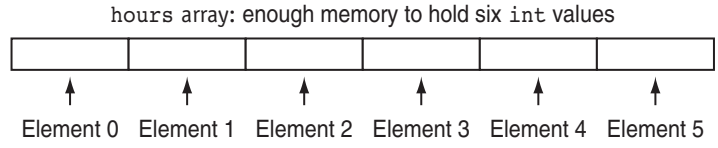


An array works like a variable that can store a group of values, all of the same type. The values are stored together in consecutive memory locations. Here is a definition of an array of integers:

```
int hours[6];
```

The name of this array is `hours`. The number inside the brackets is the array's *size declarator*. It indicates the number of *elements*, or values, the array can hold. The `hours` array can store six elements, each one an integer. This is depicted in Figure 8-2.

Figure 8-2



An array's size declarator must be a constant integer expression with a value greater than zero. It can be either a literal, as in the previous example, or a named constant, as shown here:

```
const int SIZE = 6;
int hours[SIZE];
```

Arrays of any data type can be defined. The following are all valid array definitions:

```
float temperature[100];    // Array of 100 floats
char letter[26];           // Array of 26 characters
long unit[50];             // Array of 50 long integers
double size[1200];         // Array of 1200 doubles
string name[10];           // Array of 10 string objects
```

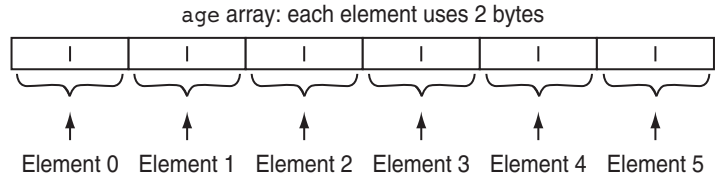
Memory Requirements of Arrays

The amount of memory used by an array depends on the array's data type and the number of elements. The `age` array, defined here, is an array that holds six `short int` values.

```
short age[6];
```

On a typical PC, a `short int` uses 2 bytes of memory, so the `age` array would occupy 12 bytes. This is shown in Figure 8-3.

Figure 8-3



The size of an array can be calculated by multiplying the number of bytes needed to store an individual element by the number of elements in the array. Table 8-1 shows the sizes of various arrays on a typical system.

Table 8-1 Example Array Size Declarators

Array declaration	Number of elements	Size of each element	Size of the array
`char letter[26];`	26	1 byte	26 bytes
`short ring[100];`	100	2 bytes	200 bytes
`int mile[84];`	84	4 bytes	336 bytes
`float temp[12];`	12	4 bytes	48 bytes
`double distance[1000];`	1000	8 bytes	8,000 bytes

8.2 Accessing Array Elements

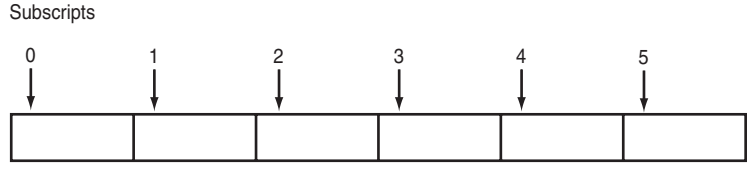
CONCEPT: **The individual elements of an array are assigned unique subscripts. These subscripts are used to access the elements.**



VideoNote
Accessing Array Elements

Even though an entire array has only one name, the elements may be accessed and used as individual variables. This is possible because each element is assigned a number known as a *subscript*. A subscript is used as an index to pinpoint a specific element within an array. The first element is assigned the subscript 0, the second element is assigned 1, and so forth. The six elements in the `hours` array we defined in the previous section would have the subscripts 0 through 5. This is shown in Figure 8-4.

Figure 8-4





NOTE: Subscript numbering in C++ always starts at zero. The subscript of the last element in an array is one less than the total number of elements in the array. This means that in the array shown in Figure 8-4, the element `hours[6]` does not exist. The last element in the array is `hours[5]`.

Each element in the `hours` array, when accessed by its subscript, can be used as an `int` variable. Here is an example of a statement that stores the number 20 in the first element of the array:

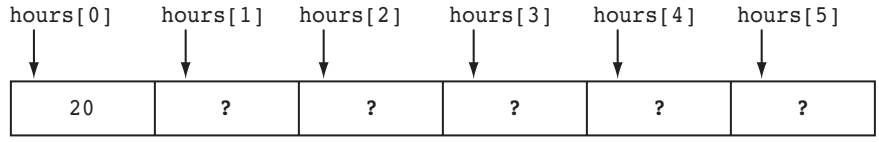
```
hours[0] = 20;
```



NOTE: The expression `hours[0]` is pronounced "hours sub zero." You would read this assignment statement as "hours sub zero is assigned twenty."

Figure 8-5 shows the contents of the `hours` array after the statement assigns 20 to `hours[0]`.

Figure 8-5





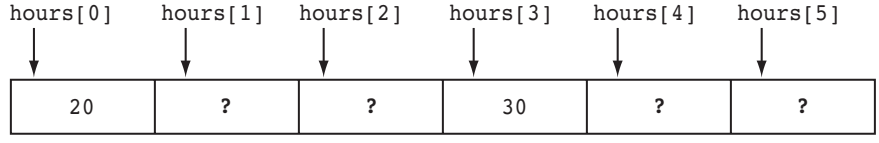
NOTE: Because values have not been assigned to the other elements of the array, question marks are used to indicate that the contents of those elements are unknown. If an array holding numeric values is defined globally, all of its elements are initialized to zero by default. Local arrays, however, have no default initialization value.

The following statement stores the integer 30 in `hours[3]`. Note that this is the fourth array element.

```
hours[3] = 30;
```

Figure 8-6 shows the contents of the array after this statement executes.

Figure 8-6





NOTE: It is important to understand the difference between the array size declarator and a subscript. The number inside the brackets in an array definition is the size declarator. It specifies how many elements the array holds. The number inside the brackets in an assignment statement or any statement that works with the contents of an array is a subscript. It specifies which element is being accessed.

Array elements may receive values with assignment statements just like other variables. However, entire arrays may not receive values for all their elements at once. Assume the following two arrays have been defined.

```
int doctorA[5];      // Holds the number of patients seen by Dr. A
                     // on each of 5 days.
int doctorB[5];      // Holds the number of patients seen by Dr. B
                     // on each of 5 days.
```

The following are all legal assignment statements.

```
doctorA[0] = 31;           // doctorA[0] now holds 31.
doctorA[1] = 40;           // doctorA[1] now holds 40.
doctorA[2] = doctorA[0];   // doctorA[2] now also holds 31.
doctorB[0] = doctorA[1];   // doctorB[0] now holds 40.
```

However, the following statements are not legal.

```
doctorA = 152;        // Illegal! An array as a whole may not
doctorB = doctorA;    // be assigned a value. This must be done
                      // one element at a time, using a subscript.
```

8.3 Inputting and Displaying Array Contents

Array elements may also have information read into them using the `cin` object and have their values displayed with the `cout` object, just like regular variables, as long as it is done one element at a time. Program 8-1 shows the `hours` array, discussed in the last section, being used to store and display values entered by the user.

Program 8-1

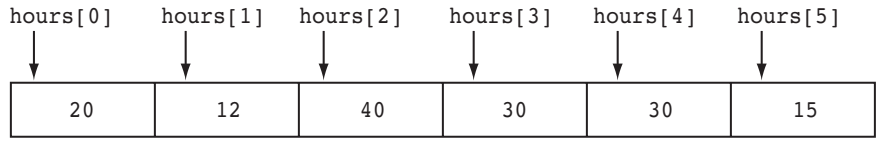
```
1  // This program stores employee work hours in an int array.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     const int NUM_EMPLOYEES = 6;
8     int hours[NUM_EMPLOYEES];    // Holds hours worked for 6 employees
9
10    // Input hours worked by each employee
11    cout << "Enter the hours worked by " << NUM_EMPLOYEES
12         << " employees: ";
13    cin  >> hours[0];
14    cin  >> hours[1];
15    cin  >> hours[2];
16    cin  >> hours[3];
17    cin  >> hours[4];
18    cin  >> hours[5];
19
20    // Display the contents of the array
21    cout << "The hours you entered are:";
22    cout << " " << hours[0];
23    cout << " " << hours[1];
24    cout << " " << hours[2];
25    cout << " " << hours[3];
26    cout << " " << hours[4];
27    cout << " " << hours[5] << endl;
28    return 0;
29 }
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked by 6 employees: 20 12 40 30 30 15[Enter]
The hours you entered are: 20 12 40 30 30 15
```

Figure 8-7 shows the contents of the hours array with the example values entered by the user for Program 8-1.

Figure 8-7



Even though most C++ compilers require the size declarator of an array definition to be a constant or a literal, subscript numbers can be stored in variables. This makes it possible to use a loop to "cycle through" an entire array, performing the same operation on each element. For example, Program 8-1 could be simplified by using two for loops: one for inputting the values into the array and another for displaying the contents of the array. This is shown in Program 8-2.

Program 8-2

```
1  // This program stores employee work hours in an int array. It uses
2  // one for loop to input the hours and another for loop to display them.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int NUM_EMPLOYEES = 6;
9     int hours[NUM_EMPLOYEES];      // Holds hours worked for 6 employees
10    int count;                     // Loop counter
11
12    // Input hours worked by each employee
13    cout << "Enter the hours worked by " << NUM_EMPLOYEES
14         << " employees: ";
15
16    for (count = 0; count < NUM_EMPLOYEES; count++)
17       cin >> hours[count];
18
19    // Display the contents of the array
20    cout << "The hours you entered are:";
21
22    for (count = 0; count < NUM_EMPLOYEES; count++)
23       cout << " " << hours[count];
24
25    cout << endl;
26    return 0;
27 }
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked by 6 employees: 20 12 40 30 30 15[Enter]
The hours you entered are: 20 12 40 30 30 15
```

Let's look at Program 8-2 more carefully. In line 9, the hours array is defined using the named constant NUM_EMPLOYEES as the size declarator. This creates the hours array with

six elements, `hours[0]` through `hours[5]`. In lines 16 and 17 a `for` loop is used to input a value into each array location. Notice that `count`, the loop control variable, is also used as the subscript for the `hours` array. Each time the loop iterates, `count` will have a different value, so a different array element will be accessed.

Because the `for` loop initializes `count` to 0, the first time the loop iterates, the user input value is read into `hours[0]`. The next time the loop iterates, `count` equals 1, so this time the user input value is read into `hours[1]`. This continues until, on the last iteration, `count` equals 5, and the final user input value is read into `hours[5]`. The `for` loop test condition is written so that when `count` reaches `NUM_EMPLOYEES`, which equals 6, the loop will stop.

The program's second `for` loop appears in lines 22 and 23. It works in a similar fashion, except that this loop is using `cout` to display each array element's value, rather than `cin` to read a value into each array element. In line 22 the `count` variable is re-initialized to 0, so the first time the loop iterates, the value stored in `hours[0]` is displayed. The next time the loop iterates, `count` equals 1, so this time the value stored in `hours[1]` is displayed. This continues until, on the final iteration, `count` equals 5 and the value stored in `hours[5]` is displayed.

Reading Data from a File into an Array

In many circumstances you will need to read data from a file and store it in an array. The process is straightforward, and in most cases is best done with a loop. Each iteration of the loop reads an item from the file and stores it in an array element. Program 8-3 modifies Program 8-2 to read the data from a file.

Program 8-3

```
1  // This program reads employee work hours from a file and stores them
2  // in an int array. It uses one for loop to input the hours and another
3  // for loop to display them.
4  #include <iostream>
5  #include <fstream>
6  using namespace std;
7
8  int main()
9  {
10    const int NUM_EMPLOYEES = 6;  // Sets number of employees
11    int hours[NUM_EMPLOYEES];     // Holds each employee's hours
12    int count;                    // Loop counter
13    ifstream datafile;            // Used to read data from a file
14
15    // Open the data file
16    datafile.open("work.dat");
17    if (!datafile)
18       cout << "Error opening data file\n";
19    else
20    {  // Input hours worked by each employee
21       for (count = 0; count < NUM_EMPLOYEES; count++)
22          datafile >> hours[count];
23       datafile.close();
24
```

(program continues)

Program 8-3 *(continued)*

```
25       // Display the contents of the array
26       cout << "The hours worked by each employee are\n";
27       for (count = 0; count < NUM_EMPLOYEES; count++)
28       {  cout << "Employee " << count+1 << ": ";
29          cout << hours[count] << endl;
30       }
31    }
32    return 0;
33 }
```

Program Output

```
The hours worked by each employee are
Employee 1: 20
Employee 2: 12
Employee 3: 40
Employee 4: 30
Employee 5: 30
Employee 6: 15
```

Notice in Program 8-3 that when we displayed a worker's data in line 29 we printed `hours[count]`, using this line:

```
cout << hours[count] << endl;
```

However, when we displayed that same worker's number in line 28 we added 1 to the count, using this line:

```
cout << "Employee " << count+1 << ": ";
```

This is because the data for employee 1 is stored in `hours[0]`, the data for employee 2 is stored in `hours[1]`, and so forth. Because array subscripts may be any integer expression, we could also have written the loop that displays the output like this:

```
for (count = 1; count <= NUM_EMPLOYEES; count++)
{  cout << "Employee " << count << ": ";
   cout << hours[count-1] << endl;
}
```

Notice in all these examples that the contents of the `hours` array were input and displayed one element at a time. The following statements would have been incorrect.

```
cin  >> hours;        // Incorrect!
cout << hours;        // Incorrect!
datafile >> hours;    // Incorrect!
```

The only time an entire array can be input or displayed as if it were a single value is when it is a string. C-strings are discussed in Chapter 3 and in Chapter 12. String objects are used throughout this text and will be dealt with more in Section 8.5.

Writing the Contents of an Array to a File

Writing the contents of an array to a file is also a straightforward matter. First open an output file pointed to by an `ofstream` object, as you learned to do in Chapter 3. Then simply use a loop to step through each element of the array, as we did in lines 27 through 30 of Program 8-3, and direct the output to the file instead of to the computer screen.

No Bounds Checking in C++

Historically, one of the reasons for C++'s popularity has been the freedom it gives programmers to work with the computer's memory. However, this means that many of the safeguards provided by other languages to prevent programs from unsafely accessing memory are absent in C++. For example, C++ does not perform array bounds checking. This means you could write a program that accidentally allows an array's subscript to go beyond its boundaries, causing a serious problem. What exactly occurs depends on how your system manages memory. On many systems it causes other nearby variables to have their contents overwritten, losing their correct value. On some systems it can even cause the computer to crash.

Program 8-4 demonstrates what occurs on the authors' computer when an array subscript goes out of bounds. It shows that data stored into one array overwrites the data in another array. It also shows, in line 10, how to initialize an array with data when it is defined. This technique is discussed further in the following section.

Program 8-4

```
1  // This program unsafely stores values beyond an array's boundary.
2  // What happens depends on how your computer manages memory.
3  // It MAY overwrite other memory variables. It MAY crash your computer.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     const int SIZE = 3;
10    int A[SIZE] = {1, 1, 1}; // Define A as a 3-element int array
11                             // holding the values 1, 1, 1
12    int B[SIZE];             // Define B as another 3-element int array
13
14    // Here is what is stored in array A
15    cout << "Here are the original numbers in 3-element array A:  ";
16    for (int count = 0; count < 3; count++)
17       cout << A[count] << "   ";
18
19    // Attempt to store seven numbers in the 3-element array
20    cout << "\n\nNow I'm storing 7 numbers in 3-element array B.";
21    for (int count = 0; count < 7; count++)
22       B[count] = 5;
23
24    // If the program is still running, display the numbers
25    cout << "\nIf you see this message, the computer did not crash.";
26    cout << "\n\nHere are the 7 numbers in array B  : ";
27    for (int count = 0; count < 7; count++)
28       cout << B[count] << "  ";
29
```

(program continues)

Program 8-4 *(continued)*

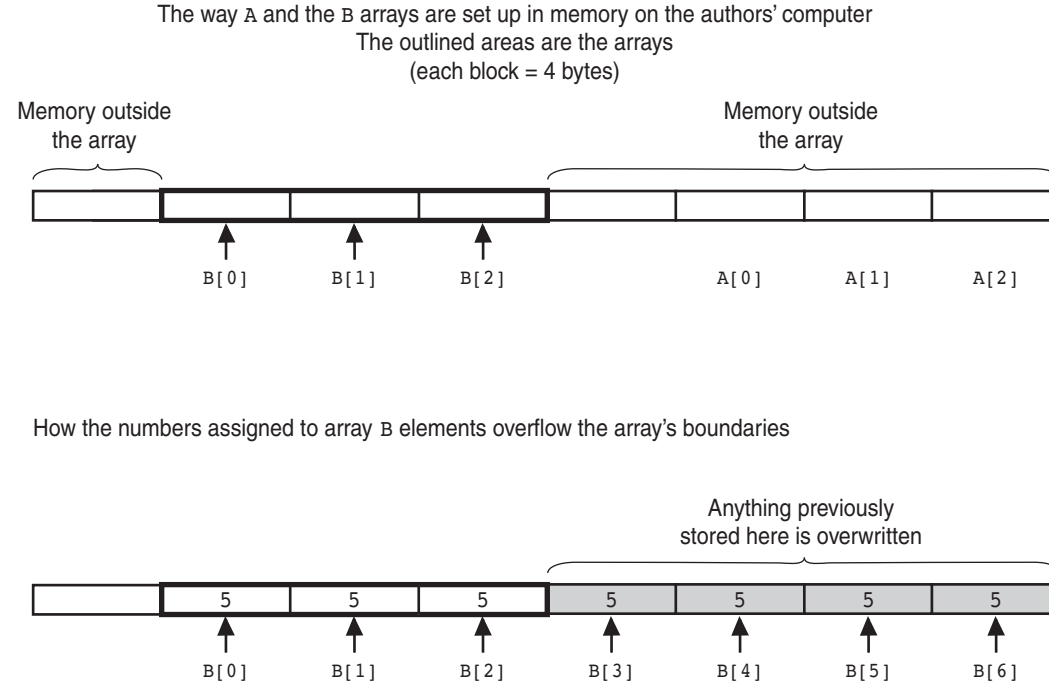
```
30    cout << "\nHere are the numbers now in array A: ";
31    for (int count = 0; count < 3; count++)
32       cout << A[count] << "  ";
33
34    return 0;
35 }
```

Program Output

```
Here are the original numbers in 3-element array A:  1   1   1

Now I'm storing 7 numbers in 3-element array B.
If you see this message, the computer did not crash.

Here are the 7 numbers in array B  : 5  5  5  5  5  5  5
Here are the numbers now in array A: 5  5  5
```

Let's look more closely at what occurred. Notice that array `A` started out with the values 1, 1, 1, but ended up with the values 5, 5, 5. This occurred because the loop in lines 21 and 22 of the program stored the value 5 in seven array `B` elements, even though array `B` only had enough memory assigned to it to store three values. The rest of the values were stored in adjacent memory locations that did not belong to array `B`. In this case, some of them belonged to array `A`, so its contents were overwritten and destroyed. Figure 8-8 illustrates this.

Figure 8-8



You can see why it's important to make sure that any time you assign values to array elements, the values are written within the array's boundaries.

Watch for Off-By-One Errors

When working with arrays, a common type of mistake is the *off-by-one error*. This is an easy mistake to make because array subscripts start at 0 rather than 1. For example, look at the following code:

```
// This code has an off-by-one error
const int SIZE = 100;
int numbers[SIZE];
for (int count = 1; count <= SIZE; count++)
   numbers[count] = 0;
```

The intent of this code is to create an array of integers with 100 elements, and store the value 0 in each element. However, this code has an off-by-one error. The loop uses its counter variable, `count`, as a subscript with the `numbers` array. During the loop's execution, the variable `count` takes on the values 1 through 100, when it should take on the values 0 through 99. As a result, the first element, which is at subscript 0, is skipped. In addition, the loop attempts to use 100 as a subscript during the last iteration. Because 100 is an invalid subscript, the program will write data beyond the array's boundaries.



Checkpoint

8.1 Define the following arrays:
- A) `empNum`, a 100-element array of `int`s
- B) `payRate`, a 25-element array of `double`s
- C) `miles`, a 14-element array of `long`s
- D) `letter`, a 26-element array of `char`s
- E) `lightYears`, a 1,000-element array of `double`s

8.2 What's wrong with the following array definitions?

```
int readings[-1];
double measurements[4.5];
int size;
char name[size];
```

8.3 What would the valid subscript values be in a four-element array of `double`s?

8.4 What is the difference between an array's size declarator and a subscript?

8.5 What is "array bounds checking"? Does C++ perform it?

8.6 What is the output of the following code?

```
int values[5], count;

for (count = 0; count < 5; count++)
   values[count] = count + 1;

for (count = 0; count < 5; count++)
   cout << values[count] << endl;
```

8.7 Complete the following program skeleton so it will have a 10-element array of `int` values called `fish`. When completed, the program should ask how many fish were caught by fishermen 1 through 10, and store this information in the array. Then it should display the data.

```
#include <iostream>
using namespace std;

int main ()
{
   const int NUM_MEN = 10;
   // Define an array named fish that can hold 10 int values.

   // You must finish this program so it works as
   // described above.
   return 0;
}
```

8.4 Array Initialization

CONCEPT: Arrays may be initialized when they are defined.

Sometimes it is more appropriate to set variable values within a program than to input them. However, writing separate assignment statements for the individual elements of an array can mean a lot of typing, especially for large arrays. For example, consider Program 8-5.

Program 8-5

```
1  // This program displays the number of days in each month.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  int main()
7  {
8     const int NUM_MONTHS = 12;
9     int days[NUM_MONTHS];
10
11    days[0]  = 31;  // January
12    days[1]  = 28;  // February
13    days[2]  = 31;  // March
14    days[3]  = 30;  // April
15    days[4]  = 31;  // May
16    days[5]  = 30;  // June
17    days[6]  = 31;  // July
18    days[7]  = 31;  // August
19    days[8]  = 30;  // September
20    days[9]  = 31;  // October
21    days[10] = 30;  // November
22    days[11] = 31;  // December
23
24    for (int month = 0; month < NUM_MONTHS; month++)
25    {
26       cout << "Month " << setw(2) << (month + 1) << " has ";
27       cout << days[month] << " days.\n";
28    }
29    return 0;
30 }
```

(program continues)

Program 8-5 *(continued)*

Program Output

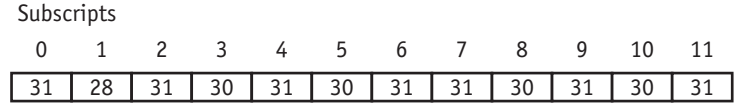
```
Month  1 has 31 days.
Month  2 has 28 days.
Month  3 has 31 days.
Month  4 has 30 days.
Month  5 has 31 days.
Month  6 has 30 days.
Month  7 has 31 days.
Month  8 has 31 days.
Month  9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

Fortunately, there is an alternative. As you saw briefly in Program 8-4, C++ allows you to initialize arrays when you define them. By using an initialization list, all the elements of the array can be easily initialized when the array is created. The following statement defines the `days` array and initializes it with the same values established by the set of assignment statements in Program 8-5:

```
int days[NUM_MONTHS] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

These values are stored in the array elements in the order they appear in the list. (The first value, 31, is stored in `days[0]`, the second value, 28, is stored in `days[1]`, and so forth). Figure 8-9 shows the contents of the array after the initialization.

Figure 8-9



Program 8-6 is a modification of Program 8-5. It initializes the `days` array at the time it is created rather than by using separate assignment statements. Notice that the initialization list is spread across multiple lines. The program also adds an array of `string` objects to hold the month names.

Program 8-6

```
1 // This program displays the number of days in each month. It uses an
2 // array of string objects to hold the month names and an int array
3 // to hold the number of days in each month. Both are initialized with
4 // initialization lists at the time they are created.
5 #include <iostream>
6 #include <iomanip>
7 #include <string>
8 using namespace std;
9
```

(program continues)

Program 8-6 *(continued)*

```
10 int main()
11 {
12    Const int NUM_MONTHS = 12;
13    string name[NUM_MONTHS] = {"January", "February", "March",
14                               "April",   "May",      "June",
15                               "July",    "August",   "September",
16                               "October", "November", "December"  };
17
18    int days[NUM_MONTHS] = {31, 28, 31, 30,
19                            31, 30, 31, 31,
20                            30, 31, 30, 31};
21
22    for (int month = 0; month < NUM_MONTHS; month++)
23    {
24       cout << setw(9) << left << name[month] << " has ";
25       cout << days[month] << " days.\n";
26    }
27    return 0;
28 }
```

Program Output

```
January   has 31 days.
February  has 28 days.
March     has 31 days.
April     has 30 days.
May       has 31 days.
June      has 30 days.
July      has 31 days.
August    has 31 days.
September has 30 days.
October   has 31 days.
November  has 30 days.
December  has 31 days.
```

So far we have demonstrated how to fill an array with values and then display all the values. Sometimes, however, we want to retrieve one specific value from the array. Program 8-7 is a variation of Program 8-6 that displays how many days are in the month the user selects.

Program 8-7

```
1 // This program allows the user to select a month and then
2 // displays how many days are in that month. It does this
3 // by "looking up" information it has stored in arrays.
4 #include <iostream>
5 #include <iomanip>
6 #include <string>
7 using namespace std;
8
```

(program continues)

Program 8-7 *(continued)*

```
 9 int main()
10 {
11    const int NUM_MONTHS = 12;
12    int choice;
13    string name[NUM_MONTHS] = {"January", "February", "March",
14                               "April",   "May",      "June",
15                               "July",    "August",   "September",
16                               "October", "November", "December"  };
17
18    int days[NUM_MONTHS] = {31, 28, 31, 30,
19                            31, 30, 31, 31,
20                            30, 31, 30, 31};
21
22    cout << "This program will tell you how many days are "
23         << "in any month.\n\n";
24
25    // Display the months
26    for (int month = 1; month <= NUM_MONTHS; month++)
27       cout << setw(2) << month << "   " << name[month-1] << endl;
28
29    cout << "\nEnter the number of the month you want: ";
30    cin  >> choice;
31
32    // Use the choice the user entered to get the name of
33    // the month and its number of days from the arrays.
34    cout << "The month of " << name[choice-1] << " has "
35         << days[choice-1]  << " days.\n";
36    return 0;
37 }
```

Program Output with Example Input Shown in Bold

```
This program will tell you how many days are in any month.

 1  January
 2  February
 3  March
 4  April
 5  May
 6  June
 7  July
 8  August
 9  September
10  October
11  November
12  December

Enter the number of the month you want: 4[Enter]
The month of April has 30 days.
```

Starting with Array Element 1

Some instructors prefer that you not use array element 0 and, instead, begin storing the actual data in element 1 when you are modeling something in the real world that logically begins with 1. The months of the year are a good example. In this case you would declare the `name` and `days` arrays to each have 13 elements and would initialize them like this:

```
string name[NUM_MONTHS+1] =
                     {" ", "January",  "February","March",   "April",
                           "May",      "June",    "July",    "August",
                           "September","October", "November","December"};

int days[NUM_MONTHS+1] = {0, 31, 28, 31, 30,
                             31, 30, 31, 31,
                             30, 31, 30, 31};
```

Notice that array element 0 is not used. It just holds a dummy value. This allows the name of the first month, January, to be stored in `name[1]`, the name of the second month, February, to be stored in `name[2]`, and so on. Likewise, the number of days in January is found in `days[1]`, the number of days in February in `days[2]`, and so on.

Here is what the loop found in lines 22 through 26 of Program 8-6 would look like if the arrays were defined and initialized as we have done here. It displays the contents of array elements 1 through 12, instead of elements 0 through 11 as before.

```
for (int month = 1; month <= NUM_MONTHS; month++)
{
   cout << setw(9) << left << name[month] << " has ";
   cout << days[month] << " days.\n";
}
```

If the actual data is stored beginning with element 1, it is also not necessary to offset array subscripts by 1 to locate a particular piece of data. Here is what the loop in lines 26 and 27 of Program 8-7 that lists each month number with its name would look like:

```
for (int month = 1; month <= NUM_MONTHS; month++)
   cout << setw(2) << month << "  " << name[month] << endl;
```

And lines 34 and 35 of Program 8-7 that display the number of days in a month selected by the user would look like this:

```
cout << "The month of " << name[choice] << " has "
     << days[choice]  << " days.\n";
```

Versions of Programs 8-5, 8-6, and 8-7 that store data values beginning with element 1 can be found on the student CD in files `pr8-05B.cpp`, `pr8-06B.cpp`, and `pr8-07B.cpp`, respectively.

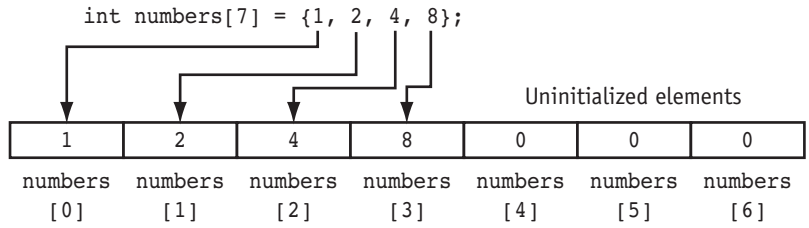
Partial Array Initialization

An array's initialization list cannot have more values than the array has elements, but it may have fewer values than there are elements. That is, C++ does not require a value for every element. It's possible to only initialize part of an array, like this:

```
int numbers[7] = {1, 2, 4, 8};
```

This definition only initializes the first four elements of a seven-element array, as illustrated in Figure 8-10.

Figure 8-10



If an array is partially initialized, the uninitialized elements will be set to zero for numeric arrays or to the null character for character arrays. This is true even if the array is defined locally. (If a local array is completely uninitialized, its elements will contain "garbage," like other local variables.) Program 8-8 shows the contents of the `numbers` array after it is partially initialized.

Program 8-8

```
1  // This program has a partially initialized array.
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7     const int SIZE = 7;
8     int numbers[SIZE] = {1, 2, 4, 8}; // Initialize the first 4 elements
9
10    cout << "Here are the contents of the array:\n";
11    for (int index = 0; index < SIZE; index++)
12       cout << numbers[index] << " ";
13    cout << endl;
14    return 0;
15 }
```

Program Output

```
Here are the contents of the array:
1  2  4  8  0  0  0
```

If you leave an element uninitialized, you must leave all the elements that follow it uninitialized as well. C++ does not provide a way to skip elements in the initialization list. For example, the following is *not* legal:

```
int array[6] = {2, 4, , 8, , 12};     // NOT legal!
```

Implicit Array Sizing

It's possible to define an array without specifying its size, as long as you provide an initialization list that includes a value for every element. C++ automatically makes the array large enough to hold all the initialization values. For example, the following definition creates an array with five elements:

```
double ratings[] = {1.0, 1.5, 2.0, 2.5, 3.0};
```

Because the size declarator is omitted, C++ counts the number of items in the initialization list and gives the array that many elements.



NOTE: You *must* specify an initialization list if you leave out the size declarator. Otherwise, C++ doesn't know how large to make the array.

8.5 Processing Array Contents

CONCEPT: Individual array elements are processed like any other type of variable.

Processing array elements is no different than processing other variables. For example, the following statement multiplies `hours[3]` by the variable `rate`:

```
pay = hours[3] * rate;
```

And the following are examples of pre-increment and post-increment operations on array elements:

```
int score[5] = {7, 8, 9, 10, 11};
++score[2];      // Pre-increment operation on the value in score[2]
score[4]++;      // Post-increment operation on the value in score[4]
```



NOTE: When using increment and decrement operators, be careful not to confuse the subscript with the array element. The following example illustrates the difference.

```
amount[count]--;  // This decrements the value stored in amount[count]
amount[count--];  // This decrements the variable count, but does
                  // nothing to the value stored in amount[count]
```

Program 8-9 demonstrates the use of array elements in a simple mathematical statement. A loop steps through each element of the array, using the elements to calculate the gross pay of five employees.

Program 8-9

```
1  // This program uses an array to store the hours worked by
2  // a set of employees who all make the same hourly wage.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     const int NUM_WORKERS = 5;  // Set the number of employees
10    int hours[NUM_WORKERS];     // Array to hold each employee's hours
11    double payRate;             // Hourly pay rate for all employees
12
```

(program continues)

Program 8-9 *(continued)*

```
13    // Input hours worked by each employee
14    cout << "Enter the hours worked by \n";
15    for (int worker = 0; worker < NUM_WORKERS; worker++)
16    {
17       cout << "Employee #" << (worker+1) << ": ";
18       cin  >> hours[worker];
19    }
20
21    // Input the hourly pay rate for all employees
22    cout << "\nEnter the hourly pay rate for all the employees: ";
23    cin  >> payRate;
24
25    // Display each employee's gross pay
26    cout << "\nHere is the gross pay for each employee:\n";
27    cout << fixed << showpoint << setprecision(2);
28    for (int worker = 0; worker < NUM_WORKERS; worker++)
29    {
30       double grossPay = hours[worker] * payRate;
31       cout << "Employee #" << (worker + 1);
32       cout << ": $" << setw(7) << grossPay << endl;
33    }
34    return 0;
35 }
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked by
Employee #1: 5[Enter]
Employee #2: 10[Enter]
Employee #3: 15[Enter]
Employee #4: 20[Enter]
Employee #5: 40[Enter]

Enter the hourly pay rate for all the employees: 12.75[Enter]

Here is the gross pay for each employee:
Employee #1: $  63.75
Employee #2: $ 127.50
Employee #3: $ 191.25
Employee #4: $ 255.00
Employee #5: $ 510.00
```

Array elements can be used in all the same ways regular variables can. You have seen how to read in a value and store it in an array element, how to assign a value to an array element, and how to display an element's value.

Array elements can also be used in relational expressions. For example, the following `if` statement tests `cost[20]` to determine whether it is less than `cost[0]`:

```
if (cost[20] < cost[0])
```

And this line begins a `while` loop that iterates as long as `value[place]` does not equal 0:

```
while (value[place] != 0)
```

Copying One Array to Another

We have already discussed that you cannot simply assign one array to another array. To copy the contents of one array to another, you must assign each element of the first array, one at a time, to the corresponding element of the second array. The following code segment uses a `for` loop to do this.

```
const int SIZE = 6;
int arrayA[SIZE] = {10, 20, 30, 40, 50, 60};
int arrayB[SIZE] = { 2,  4,  6,  8, 10, 12};
for (int index = 0; index < SIZE; index++)
   arrayA[index] = arrayB[index];
```

On the first iteration of the loop, `index = 0`, so `arrayA[0]` is assigned the value stored in `arrayB[0]`. On the second iteration, `index = 1`, so `arrayA[1]` is assigned the value stored in `arrayB[1]`. This continues until, one by one, all the elements of `arrayB` are copied to `arrayA`. When the loop is finished executing, both arrays will contain the values 2, 4, 6, 8, 10, 12.

This code can be found in the program `ArrayCopy.cpp` on the Student CD.

Comparing Two Arrays

Just as you cannot copy one array to another with a single statement, you also cannot compare the contents of two arrays with a single statement. That is, you cannot use the == operator with the names of two arrays to determine whether the arrays are equal. The following code appears to compare the contents of two arrays, but in reality does not.

```
int arrayA[] = { 5, 10, 15, 20, 25 };
int arrayB[] = { 5, 10, 15, 20, 25 };

if (arrayA == arrayB)       // This is a mistake
   cout << "The arrays are the same.\n";
else
   cout << "The arrays are not the same.\n";
```

When you use the == operator with array names, the operator compares the beginning memory addresses of the arrays, not the contents of the arrays. The two arrays in this code will obviously have different memory addresses. Therefore, the result of the expression `arrayA == arrayB` is false and the code reports that the arrays are not the same.

To compare the contents of two arrays, you must compare their individual elements. For example, look at the following code.

```
const int SIZE = 5;
int arrayA[SIZE] = { 5, 10, 15, 20, 25 };
int arrayB[SIZE] = { 5, 10, 15, 20, 25 };
bool arraysEqual = true;  // Flag variable
int count = 0;            // Loop counter variable

// Determine whether the elements contain the same data
while (arraysEqual && count < SIZE)
{
      if (arrayA[count] != arrayB[count])
         arraysEqual = false;
      count++;
}
```

```
// Display the appropriate message
if (arraysEqual)
   cout << "The arrays are equal.\n";
else
   cout << "The arrays are not equal.\n";
```

This code determines whether `arrayA` and `arrayB` contain the same values. A `bool` variable `arraysEqual`, which is initialized to `true`, is used to signal whether the arrays are equal. Another variable `count`, which is initialized to 0, is used as a loop counter variable.

Then a `while` loop begins. The loop executes as long as `arraysEqual` is `true` and the counter variable `count` is less than `SIZE`. During each iteration, it compares a different pair of corresponding elements in the arrays. If it finds two corresponding elements that have different values, the `arraysEqual` variable is set to `false`, which allows the loop to be exited without examining any more values. After the loop finishes, an `if` statement tests the `arraysEqual` variable. If the variable is still `true`, then no differences were found. The arrays are equal and a message indicating this is displayed. Otherwise, they are not equal, so a different message is displayed. This code can be found in the program `ArrayCompare.cpp` on the Student CD.

Summing the Values in a Numeric Array

To sum the values in an array, you must use a loop with an accumulator variable. The loop adds the value in each array element to the accumulator. For example, assume that the following statements appear in a program.

```
const int NUM_UNITS = 6;
int units[NUM_UNITS] = {16, 20, 14, 8, 6, 10};
int total = 0;              // Initialize accumulator
```

The following loop adds the values of each element in the array to the `total` variable. When the code is finished, `total` will contain the sum of the `units` array's elements.

```
for (int count = 0; count < NUM_UNITS; count++)
   total += units[count];
```



NOTE: Notice that `total` is initialized to 0. Recall from Chapter 5 that an accumulator variable must be set to 0 before it is used to keep a running total or the sum will not be correct.

Finding the Average of the Values in a Numeric Array

The first step in calculating the average of all the values in an array is to sum the values. The second step is to divide the sum by the number of elements in the array. Assume that the following statements appear in a program.

```
const int NUM_SCORES = 5;
double scores[NUM_SCORES] = {90, 88, 91, 82, 95};
```

The following code calculates the average of the values in the `scores` array and stores the result in the `average` variable.

```
double total = 0;           // Initialize accumulator
double average;             // Will hold the average

for (int count = 0; count < NUM_SCORES; count++)
   total += scores[count];
average = total / NUM_SCORES;
```

Notice that the last statement, which divides `total` by `NUM_SCORES`, is not inside the loop. This statement should only execute once, after the loop has finished all its iterations.

Finding the Highest and Lowest Values in a Numeric Array

The algorithms for finding the highest and lowest values in an array are very similar. First, let's look at code for finding the highest value in an array. Assume that the following statements appear in a program.

```
const int SIZE = 10;
int numbers[SIZE] = {15, 6, 3, 11, 22, 4, 0, 1, 9, 12};
```

The code to find the highest value in the array is as follows.

```
int count;
int highest;

highest = numbers[0];
for (count = 1; count < SIZE; count++)
{
   if (numbers[count] > highest)
      highest = numbers[count];
}
```

First we copy the value in the first array element to the variable named `highest`. Then the loop compares all of the remaining array elements, beginning at subscript 1, to the value in `highest`. Each time it finds a value in the array that is greater than `highest`, it copies that value to `highest`. When the loop has finished, `highest` will contain the highest value in the array.

The following code finds the lowest value in the array. As you can see, it is nearly identical to the code for finding the highest value.

```
int count;
int lowest;

lowest = numbers[0];
for (count = 1; count < SIZE; count++)
{
   if (numbers[count] < lowest)
      lowest = numbers[count];
}
```

When the loop has finished, `lowest` will contain the lowest value in the array.

Program 8-10, which creates a monthly sales report, demonstrates the algorithms for finding the sum, average, highest, and lowest values in an array. It combines the algorithms to find the highest and the lowest value into a single loop. The sales data used to fill the array is read in from the `sales.dat` file, which contains the following values

62458 81598 98745 53460 35678 86322
89920 78960 124569 43550 45679 98750

Program 8-10

```
// This program uses an array to store monthly sales figures
// for a company's regional offices. It then finds and displays
// the total, average, highest, and lowest sales amounts.
// The data to fill the array is read in from a file.
#include <iostream>
#include <fstream>             // Needed to use files
#include <iomanip>
using namespace std;

int main()
{
   const int NUM_OFFICES = 12;
   ifstream dataIn;
   int office;                 // Loop counter
   double sales[NUM_OFFICES],  // Array to hold the sales data
          totalSales = 0.0,    // Accumulator initialized to zero
          averageSales,
          highestSales,
          lowestSales;

   // Open the data file
   dataIn.open("sales.dat");
   if (!dataIn)
      cout << "Error opening data file.\n";
   else
   {  // Fill the array with data from the file
      for (office = 0; office < NUM_OFFICES; office++)
         dataIn >> sales[office];
      dataIn.close();

      // Sum all the array elements
      for (office = 0; office < NUM_OFFICES; office++)
         totalSales += sales[office];

      // Calculate average sales
      averageSales = totalSales / NUM_OFFICES;

      // Find highest and lowest sales amounts
      highestSales = lowestSales = sales[0];
      for (office = 1; office < NUM_OFFICES; office++)
      {
         if (sales[office] > highestSales)
            highestSales = sales[office];
         else if (sales[office] < lowestSales)
            lowestSales = sales[office];
      }
      // Display results
      cout << fixed << showpoint << setprecision(2);
      cout << "Total sales   $" << setw(9) << totalSales   << endl;
      cout << "Average sales $" << setw(9) << averageSales << endl;
      cout << "Highest sales $" << setw(9) << highestSales << endl;
      cout << "Lowest sales  $" << setw(9) << lowestSales  << endl;
   }
   return 0;
}
```

(program continues)

Program 8-10 *(continued)*

Program Output

```
Total sales   $899689.00
Average sales $ 74974.08
Highest sales $124569.00
Lowest sales  $ 35678.00
```

Partially-Filled Arrays

Sometimes you need to store a series of items in an array, but you do not know the number of items that there are. As a result, you do not know the exact number of elements needed for the array. One solution is to make the array large enough to hold the largest possible number of items. This can lead to another problem, however. If the actual number of items stored in the array is less than the number of elements, the array will be only partially filled. When you process a partially-filled array, you must only process the elements that contain valid data items.

A partially-filled array is normally used with an accompanying integer variable that holds the number of items stored in the array. For example, suppose a program uses the following code to create an array with 100 elements, and an `int` variable named `numValues` which will hold the number of items stored in the array. Notice that `numValues` is initialized to zero because no values have been stored in the array yet.

```
const int SIZE = 100;
int array[SIZE];
int numValues = 0;
```

Each time we add an item to the array, we must increment `numValues`. The following code demonstrates.

```
int number;

cout << "Enter a number or -1 to quit: ";
cin  >> number;
while (number != -1 && numValues < SIZE)
{
   array[numValues] = number;
   numValues++;
   cout << "Enter a number or -1 to quit: ";
   cin  >> number;
}
```

Each iteration of this sentinel-controlled loop allows the user to enter a number to be stored in the array, or -1 to quit. After each value is stored in the array, the `numValues` variable is incremented to hold the subscript of the next available element in the array. When the user enters -1, or when `numValues` exceeds 99, the loop stops. The following code displays all of the valid items in the array.

```
for (int index = 0; index < numValues; index++)
{
   cout << array[index] << endl;
}
```

Why Use an Array?

Program 8-10 stored a set of numbers in an array in order to sum the numbers and find the average, largest, and smallest values. However, this could have been done without using an array at all. The sales figures could have just been placed one at a time into a simple variable, added to a sum, and compared to the largest and smallest values as they were read in. This is illustrated by the following code segment.

```
dataIn >> salesAmt;     // Input the data from the first office
totalSales = highestSales = lowestSales = salesAmt;
for (office = 2; office <= numOffices; office++)
{  dataIn >> salesAmt;
   totalSales += salesAmt;
   if (salesAmt > highestSales)
      highestSales = salesAmt;
   else if (salesAmt < lowestSales)
      lowestSales = salesAmt;

}
averageSales = totalSales / numOffices;
```

Then why use an array at all? There are many reasons. One of the most important is that once the data is in the array it can be used more than once without having to be input again. For example, supposed that instead of finding the highest and lowest sales figures we want to create a report that tells which offices have below-average sales figures. Program 8-11 modifies Program 8-10 to do this. Note that it requires looking at each piece of data twice. First each value must be input and summed to find and display the average. Then each data value must be examined again, so it can be compared to the average, and the below-average values displayed. Program 8-11 also illustrates the use of a partially-filled array. It allows the `sales` array to hold up to 20 values, then uses the loop control variable of a `while` loop to count the actual number of values stored in it as they are read in from the file. The data is read in from the same `sales.dat` file used by Program 8-10.

Program 8-11

```
1  // This program uses a partially-filled array to store monthly sales
2  // figures for a set of offices. It then finds and displays the total
3  // sales amount, the average sales amount, and a listing of the offices
4  // with sales below the average. The data to fill the array is read
5  // in from a file and the number of data values are counted.
6  #include <iostream>
7  #include <fstream>           // Needed to use files
8  #include <iomanip>
9  using namespace std;
10
11 int main()
12 {
13    const int SIZE = 20;
14    ifstream dataIn;         // Object to read file input
15    int numOffices,          // Number of data values read in
16        count;               // Loop counter
```

(program continues)

Program 8-11 *(continued)*

```
17    double sales[SIZE],       // Array to hold the sales data
18           totalSales = 0.0,  // Accumulator initialized to zero
19           averageSales;      // Average sales for all offices
20
21    // Open the data file
22    dataIn.open("sales.dat");
23    if (!dataIn)
24       cout << "Error opening the data file.\n";
25    else
26    {  // Read values from the file and store them in the array,
27       // counting them and summing them as they are read in
28       count = 0;
29       while (count < SIZE && dataIn >> sales[count])
30       {  totalSales += sales[count];
31          count++;
32       }
33       numOffices = count;
34       dataIn.close();
35
36       // Calculate average sales
37       averageSales = totalSales / numOffices;
38
39       // Display total and average
40       cout << fixed << showpoint << setprecision(2);
41       cout << "The total sales are   $"
42            << setw(9) << totalSales << endl;
43       cout << "The average sales are $"
44            << setw(9) << averageSales << endl;
45
46       // Display figures for offices performing below the average
47       cout << "\nThe following offices have below-average "
48            << "sales figures.\n";
49       for (int office = 0; office < numOffices; office++)
50       {  if (sales[office] < averageSales)
51             cout << "Office "     << setw(2) << (office + 1)
52                  << " $" << sales[office] << endl;
53       }
54    }
55    return 0;
56 }
```

Program Output

```
The total sales are   $899689.00
The average sales are $ 74974.08

The following offices have below-average sales figures.
Office  1 $62458.00
Office  4 $53460.00
Office  5 $35678.00
Office 10 $43550.00
Office 11 $45679.00
```

Let's look at a couple of key points in Program 8-11. First, look at line 29. This line controls the `while` loop and reads in the data.

```
while (count < SIZE && dataIn >> sales[count])
```

The loop repeats as long as `count` is less than the size of the array *and* a data value is successfully read in from the file (i.e., the end of the file has not been encountered). The first part of the `while` loop's test expression, `count < SIZE`, prevents the loop from writing outside the array boundaries. The second part of the test expression stops the loop if there is no more data in the file to read. Recall from Chapter 4 that the `&&` operator performs short-circuit evaluation, so the second part of the `while` loop's test expression, `dataIn >> sales[count]`, will be executed only if `count` is less than `SIZE`. The `sales` array defined in line 17 has room to store up to 20 values, but because the data file contains only 12 values, the `while` loop terminates after reading in these 12 items.

Notice how `count`, the `while` loop control variable, serves two purposes in addition to controlling execution of the loop. Because it is initialized to zero and is incremented on line 31 once each time the loop iterates, it keeps count of which array position the next item read should be stored in, correctly allowing the 12 values from the `sales.dat` file to be stored in array positions 0 through 11. It also keeps count of how many values are read in. When the loop terminated in our sample run, `count` was 12, which equaled the number of items read in.

We said that using an array is particularly helpful when data values need to be looked at more than once. That is exactly what happens in Program 8-11. The statement in line 30 adds each piece of stored data to a total it is accumulating of all the values. This total is later used in line 37 to compute an average. Then, inside the `for` loop on lines 49 through 53, each stored data item is again examined to compare it to the average and to display it if it is below the average

As you continue to program you will encounter many additional algorithms that require examining data values more than once and you will discover many cases where arrays are a particularly useful way to organize and store data.

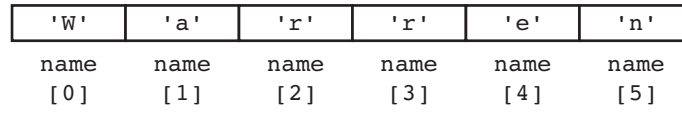
Processing Strings

Strings are internally stored as arrays of characters. They are different from other arrays in that the elements can either be treated as a set of individual characters or can be used as a single entity. The following sample code defines a string object and treats it as a single entity, inputting it and displaying it as a single unit.

```
string name;
cout << "Enter your name: ";
cin  >> name;
cout << "Hello, " << name << endl;
```

This is, in fact, how strings are normally treated and processed—as single entities. However, C++ provides the ability to index them with a subscript, like any other array, so they can be processed character by character. If `"Warren"` were entered for the name in the previous code segment, it would be stored in the `name` string object as shown in Figure 8-11.

Figure 8-11





NOTE: Both string objects and C-strings are stored as characters in contiguous bytes of memory, as shown in Figure 8-11. String constants and C-strings are terminated by placing a `'\0'`, which represents the null terminator, in the byte of memory following the last character of the string. There is no guarantee, however, how string objects will be implemented. Many versions of C++ do terminate string objects with the null terminator, but it is never safe to assume they will be terminated this way.

If we wanted to process the string character by character, like a regular array, we could do so. For example the statement

```
cout << name[0];     would print the letter W,
cout << name[1];     would print the letter a, and so forth
```

Program 8-12 illustrates character by character string processing. It reads in a string and then counts the number of vowels in the string. The `string` class member function `length` is used to determine how many characters are in the string.

Program 8-12

```
1  // This program illustrates how a string can be processed as an array
2  // of individual characters. It reads in a string, then counts the
3  // number of vowels in the string. It uses the toupper function to
4  // uppercase each letter in the string and the string class member
5  // function length() to determine how many characters are in the string.
6  #include <iostream>
7  #include <string>          // Needed to use string objects
8  #include <cctype>          // Needed for the toupper function
9  using namespace std;
10
11 int main()
12 {
13    char ch;
14    int vowelCount = 0;
15    string sentence;
16
17    cout << "Enter any sentence you wish and I will \n"
18         << "tell you how many vowels are in it.\n";
19    getline(cin, sentence);
20
21    for (int pos = 0; pos < sentence.length(); pos++)
22    {
23       // Uppercase a copy of the next character and assign it to ch
24       ch = toupper(sentence[pos]);
25
```

(program continues)

Program 8-12 *(continued)*

```
26       // If the character is a vowel, increment vowelCount
27       switch(ch)
28       {  case 'A':
29          case 'E':
30          case 'I':
31          case 'O':
32          case 'U': vowelCount++;
33       }
34    }
35    cout << "There are " << vowelCount << " vowels in the sentence.\n";
36    return 0;
37 }
```

Program Output with Example Input Shown in Bold

```
Enter any sentence you wish and I will
tell you how many vowels are in it.
The quick brown fox jumped over the lazy dog.[Enter]
There are 12 vowels in the sentence.
```

Additional examples of string processing are introduced in Chapter 12.

8.6 Using Parallel Arrays

CONCEPT: **By using the same subscript, you can build relationships between data stored in two or more arrays.**

Sometimes it is useful to store related data in two or more arrays. It's especially useful when the related data is of different data types. We did this in Programs 8-6 and 8-7, where the `name` array stored the names of the 12 months and the `days` array stored the number of days in a given month. A month name and its number of days were related by having the same subscript. For example, `days[3]` stored the number of days in the month whose name was stored in `month[3]`. When data items stored in two or more arrays are related in this fashion, the arrays are called *parallel arrays*. Program 8-13, which is a variation of the payroll program, uses parallel arrays. An `int` array stores the hours worked by each employee and a `double` array stores each employee's hourly pay rate.

Program 8-13

```
1  // This program stores employee hours worked and hourly pay rates
2  // in two parallel arrays.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     const int NUM_EMPS = 5;
10    int index;
```

(program continues)

Program 8-13 *(continued)*

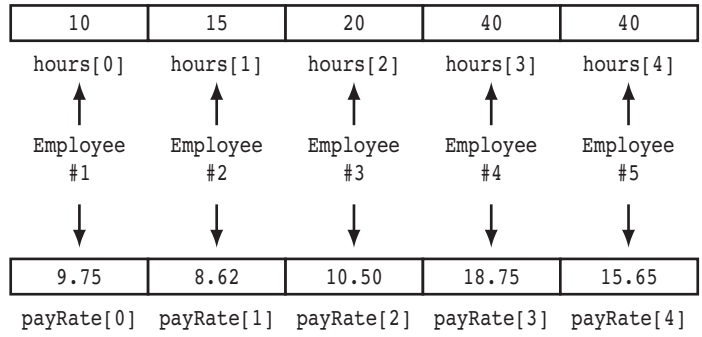
```
11    int hours[NUM_EMPS];       // Define 2 parallel arrays
12    double payRate[NUM_EMPS];
13    double grossPay;
14
15    // Get employee work data
16    cout << "Enter the hours worked and hourly pay rates of "
17         << NUM_EMPS << " employees. \n";
18
19    for (index = 0; index < NUM_EMPS; index++)
20    {
21       cout << "Hours worked by employee #" << (index + 1) << ": ";
22       cin  >> hours[index];
23       cout << "Hourly pay rate for employee #" << (index + 1) << ": ";
24       cin  >> payRate[index];
25    }
26    // Display the data
27    cout << "\nHere is the gross pay for each employee:\n";
28    cout << fixed << showpoint << setprecision(2);
29    for (index = 0; index < NUM_EMPS; index++)
30    {
31       grossPay = hours[index] * payRate[index];
32       cout << "Employee #" << (index + 1);
33       cout << ": $" << setw(7) << grossPay << endl;
34    }
35    return 0;
36 }
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked and hourly pay rates of 5 employees.
Hours worked by employee #1: 10[Enter]
Hourly pay rate for employee #1: 9.75[Enter]
Hours worked by employee #2: 15[Enter]
Hourly pay rate for employee #2: 8.62[Enter]
Hours worked by employee #3: 20[Enter]
Hourly pay rate for employee #3: 10.50[Enter]
Hours worked by employee #4: 40[Enter]
Hourly pay rate for employee #4: 18.75[Enter]
Hours worked by employee #5: 40[Enter]
Hourly pay rate for employee #5: 15.65[Enter]

Here is the gross pay for each employee:
Employee #1: $  97.50
Employee #2: $ 129.30
Employee #3: $ 210.00
Employee #4: $ 750.00
Employee #5: $ 626.00
```

Notice in the loops that the same subscript is used to access both arrays. That's because the data for a particular employee is stored in the same relative position in each array. For example, the hours worked by employee #1 are stored in `hours[0]`, and the same employee's pay rate is stored in `payRate[0]`. The subscript relates the data in both arrays. This concept is illustrated in Figure 8-12.

Figure 8-12



Checkpoint

8.8 Define the following arrays:

A) `ages`, a 10-element array of `int`s initialized with the values 5, 7, 9, 14, 15, 17, 18, 19, 21, and 23

B) `temps`, a 7-element array of `double`s initialized with the values 14.7, 16.3, 18.43, 21.09, 17.9, 18.76, and 26.7

C) `alpha`, an 8-element array of `char`s initialized with the values 'J', 'B', 'L', 'A', '*', '$', 'H', and 'M'

8.9 Indicate if each of the following array definitions is valid or invalid (If a definition is invalid, explain why.)

A) `int numbers[10] = {0, 0, 1, 0, 0, 1, 0, 0, 1, 1};`
B) `int matrix[5] = {1, 2, 3, 4, 5, 6, 7};`
C) `double radii[10] = {3.2, 4.7};`
D) `int table[7] = {2, , , 27, , 45, 39};`
E) `char codes[] = {'A', 'X', '1', '2', 's'};`
F) `int blanks[];`
G) `string suit[4] = {"Clubs", "Diamonds", "Hearts", "Spades"};`

8.10 Given the following array definitions

```
double array1[4] = {1.2, 3.2, 4.2, 5.2};
double array2[4];
```

will the following statement work? If not, why?

```
array2 = array1;
```

8.11 Given the following array definition:

```
int values[] = {2, 6, 10, 14};
```

what do each of the following display?

A) `cout << values[2];`
B) `cout << ++values[0];`
C) `cout << values[1]++;`
D)
```
x = 2;
cout << values[++x];
```

8.12 Given the following array definition

```
int nums[5] = {1, 2, 3};
```

what will the following statement display?

```
cout << nums[3];
```

8.13 What is the output of the following code?

```
double balance[5] = {100.0, 250.0, 325.0, 500.0, 1100.0};
const double INT_RATE = 0.1;

cout << fixed << showpoint << setprecision(2);
for (int count = 0; count < 5; count++)
   cout << (balance[count] * INT_RATE) << endl;
```

8.14 What is the output of the following code?

```
const int SIZE 5;
int count;
int time[SIZE] =  {1, 2, 3, 4, 5},
    speed[SIZE] = {18, 4, 27, 52, 100},
    dist[SIZE];

for (count = 0; count < SIZE; count++)
   dist[count] = time[count] * speed[count];

for (count = 0; count < SIZE; count++)
{
   cout << time[count] << " ";
   cout << speed[count] << " ";
   cout << dist[count] << endl;
}
```

8.7 The `typedef` Statement

CONCEPT: The `typedef` statement allows an alias to be associated with a simple or structured data type.

The `typedef` statement allows the programmer to create an alias, or synonym, for an existing data type. This can be a simple data type, like an `int`, or a more complicated data type such as an array. The simplest form of the statement is

```
typedef  <existing data type> <alias>;
```

For example, the following statements declare `examScore` to be another name for an `int` and then define two variables of type `examScore`.

```
typedef int examScore;
examScore score1, score2;   // score1 and score2 are of type examScore
```

The declaration emphasizes that variables of type `examScore` are integers that will hold exam scores.

One of the most common uses of the `typedef` statement is to provide a descriptive alias for an array of a specific purpose. When used with arrays, the `[]` holding the array size is written next to the alias name, not next to the data type name. The following statement creates an alias named `score` for a `double` array of size 100.

```
typedef double score[100];
```

This means that anything defined to be a `score` is an array of 100 `double` elements intended to hold scores. The following two statements now do the same thing.

```
double finalExam[100];
score finalExam;
```

Sometimes it is desirable to create an alias for an array of a specific data type without specifying its size. The following statement creates an alias, named `arrayType` for an `int` array of unspecified size.

```
typedef int arrayType[];
```

In the next section, when you learn how to pass arrays as function arguments, it will become apparent why it is convenient to set up a `typedef` for an array type.

8.8 Arrays as Function Arguments

CONCEPT: **To pass an array as an argument to a function, simply pass the name of the array.**



VideoNote
Passing an Array to a Function

Quite often you'll want to write functions that process the data in arrays. For example, functions can be written to put values in an array, display an array's contents on the screen, total all of an array's elements, or calculate their average. Usually, such functions accept an array as an argument.

When a single element of an array is passed to a function, it is handled like any other variable. For example, Program 8-14 shows a loop that passes one element of the `collection` array to the `showValue` function each time the loop is executed. Because the elements of the `collection` array are `int`s, a single `int` value is passed to the `showValue` function each time it is called. Notice how this is specified in the `showValue` function prototype and function header. All `showValue` knows is that it is receiving an `int`. It does not matter that it happens to be coming from an array.

Program 8-14

```
1  // This program demonstrates that an array element can be passed
2  // to a function like any other variable.
3  #include <iostream>
4  using namespace std;
5
6  void showValue(int);    // Function prototype
7
8  int main()
9  {
10    const int ARRAY_SIZE = 8;
11    int collection[ARRAY_SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
```

(program continues)

Program 8-14 *(continued)*

```
12
13    for (int index = 0; index < ARRAY_SIZE; index++)
14       showValue(collection[index]);
15    cout << endl;
16    return 0;
17 }
18
19 /*************************************
20  *               showValue           *
21  * This function displays the integer *
22  * value passed to its num parameter. *
23  *************************************/
24 void showValue(int num)
25 {
26    cout << num << " ";
27 }
```

Program Output

```
5 10 15 20 25 30 35 40
```

Because the `showValue` function simply displays the contents of `num` and doesn't need to work directly with the array elements themselves, the array elements are passed to it by value. If the function needed to access the original array elements, they would be passed by reference.

If the function were written to accept the entire array as an argument, the parameter would be set up differently. In the following function definition, the parameter `nums` is followed by an empty set of brackets. This indicates that the argument will be an entire array, not a single value.

```
void showValues (int nums[], int size)
{
   for (int index = 0; index < size; index++)
      cout << nums[index] << " ";
   cout << endl;
}
```

Notice that along with the array containing the values, the size of the array is also passed to `showValues`. This is so it will know how many values there are to process.

Notice also that there is no size declarator inside the brackets of `nums`. This is because `nums` is not actually an array—it's a special variable that accepts the *address* of an array. When an entire array is passed to a function, it is not passed by value. Imagine the CPU time and memory that would be necessary if a copy of a 10,000-element array were created each time it was passed to a function! Instead, only the starting memory address of the array is passed. This is similar to passing a variable to a function by reference, except that in this case no `&` is used. Program 8-15 illustrates how function `showValues` receives the address of an entire array so it can access and display the contents of all its elements.

Program 8-15

```
1  // This program shows how to pass an entire array to a function.
2  #include <iostream>
3  using namespace std;
4
5  void showValues(int intArray[], int size); // Function prototype
6
7  int main()
8  {
9     const int ARRAY_SIZE = 8;
10    int collection[ARRAY_SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
11
12    cout << "The array contains the values\n";
13    showValues(collection, ARRAY_SIZE);
14    return 0;
15 }
16
17 /**********************************************************
18  *                        showValues                      *
19  * This function displays the contents of an integer array   *
20  * when passed the array's address and its size as arguments.*
21  **********************************************************/
22 void showValues (int nums[], int size)
23 {
24    for (int index = 0; index < size; index++)
25       cout << nums[index] << "  ";
26    cout << endl;
27 }
```

Program Output

```
The array contains the values
5  10  15  20  25  30  35  40
```

Look closely at the `showValues` prototype in line 5 and function header in line 22. In both cases a pair of braces follows the first parameter name. This lets the program know that this parameter accepts the address of an array. If the function prototype had not used parameter names, it would have looked like this:

```
void showValues(int [], int);
```

This would still have indicated that the first `showValues` parameter receives the address of an integer array and the second parameter receives a single integer value.

Look also at how the `showValues` function is called in line 13 of the program with the following statement:

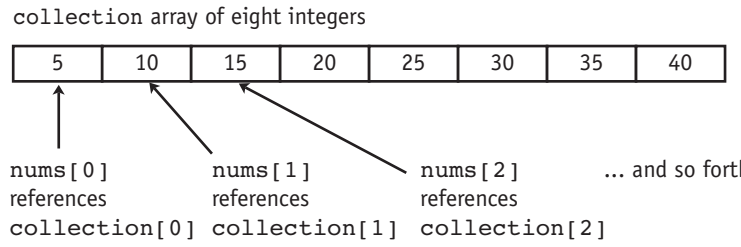
```
showValues(collection, ARRAY_SIZE);
```

The first argument is the name of the array being passed to the function. Remember, in C++ the name of an array without brackets and a subscript is actually the beginning address of the array. In this function call, the address of the `collection` array is being passed to the function. The second argument is the size of the array.

In the `showValues` function, the beginning address of the `collection` array is copied into the `nums` parameter variable. The `nums` variable is then used to reference the `collection`

array. Figure 8-13 illustrates the relationship between the `collection` array and the `nums` parameter variable. When `nums[0]` is displayed, it is actually the contents of `collection[0]` that appears on the screen.

Figure 8-13





NOTE: Although `nums` is not a reference variable, it works like one.

The `nums` parameter variable in the `showValues` function can accept the address of any integer array and can use it to reference that array. So, we can use the `showValues` function to display the contents of any integer array by passing the name of the array and its size as arguments. Program 8-16 uses this function to display the contents of two different arrays. Notice that they do not have to be the same size. Notice also the use of the `typedef` statement in this program. It makes the name `arrayType` an alias for an array of integers. This name is then used in the `showValues` prototype and function header, instead of using `int[]`, to indicate that the first parameter receives the starting address of an `int` array.

Program 8-16

```
1  // This program demonstrates passing different arrays to a function.
2  #include <iostream>
3  using namespace std;
4
5  // Declare arrayType to be an alias for an array of ints
6  typedef int arrayType[];
7
8  void showValues(arrayType, int);_// Function prototype
9
10 int main()
11 {
12    const int SIZE1 = 8;
13    const int SIZE2 = 5;
14    int set1[] = {5, 10, 15, 20, 25, 30, 35, 40};
15    int set2[] = {2, 4, 6, 8, 10};
16
17    cout << "Here are the values stored in array set1: ";
18    showValues(set1, SIZE1);    // Pass set 1 to showValues
19
20    cout << "Here are the values stored in array set2: ";
21    showValues(set2, SIZE2);    // Pass set 2 to showValues
22    return 0;
23 }
24
```

(program continues)

Program 8-16 *(continued)*

```
25 /*********************************************************
26  *                          showValues                   *
27  * This function displays the contents of an integer array   *
28  * when passed the array's address and its size as arguments.*
29  *********************************************************/
30 void showValues (arrayType nums, int size)
31 {
32    for (int index = 0; index < size; index++)
33       cout << nums[index] << " ";
34    cout << endl;
35 }
```

Program Output

```
Here are the values stored in array set1: 5 10 15 20 25 30 35 40
Here are the values stored in array set2: 2 4 6 8 10
```

Notice that when `set1` and `set2` are declared in lines 14 and 15, no size declarator is used. We could have used one, but recall that a size declarator is not required when an initialization list is used.

Recall also, from Chapter 6, that when a reference variable is used as a parameter, it gives the function access to the original argument. Any changes made to the reference variable are actually performed on the argument referenced by the variable. Array parameters work very much like reference variables. They give the function direct access to the original array. Any changes made to the array parameter are actually made to the original array used as the argument. The function `doubleArray` in Program 8-17 uses this capability to double the contents of each element in the array.

Program 8-17

```
1  // This program uses a function to double the value of
2  // each element of an array.
3  #include <iostream>
4  using namespace std;
5
6  // Declare arrayType to be an alias for an array of ints
7  typedef int arrayType[];
8
9  // Function prototypes
10 void doubleArray(arrayType, int);
11 void showValues (arrayType, int);
12
13 int main()
14 {
15    const int ARRAY_SIZE = 7;
16    arrayType set = {1, 2, 3, 4, 5, 6, 7};
17
18    // Display the original values
19    cout << "The arrays values are:\n";
20    showValues(set, ARRAY_SIZE);
21
```

(program continues)

Program 8-17 *(continued)*

```
22    // Double the values in the array
23    doubleArray(set, ARRAY_SIZE);
24
25    // Display the new values
26    cout << "\nAfter calling doubleArray, the values are:\n";
27    showValues(set, ARRAY_SIZE);
28    cout << endl;
29    return 0;
30 }
31
32 /************************************************
33  *                    doubleArray                *
34  * This function doubles the value of each element *
35  * in the array whose address is passed to it.     *
36  ************************************************/
37 void doubleArray(arrayType nums, int size)
38 {
39    for (int index = 0; index < size; index++)
40       nums[index] *= 2;
41 }
42
43 /********************************************************
44  *                          showValues                  *
45  * This function displays the contents of an int array  *
46  * when passed the array's address and size as arguments.*
47  ********************************************************/
48 void showValues (arrayType nums, int size)
49 {
50    for (int index = 0; index < size; index++)
51       cout << nums[index] << " ";
52    cout << endl;
53 }
```

Program Output

```
The array values are:
1 2 3 4 5 6 7

After calling doubleArray, the values are:
2 4 6 8 10 12 14
```

Notice that in line 16 of Program 8-17 the `set` array is defined to be type `arrayType` rather than `int set[]` or `int set[ARRAY_SIZE]`, although it could be defined either of these ways also. As in Program 8-16, it is not necessary to indicate the size of the array because it is initialized with an initialization list at the time it is created.

Notice also that in the `typedef` statement in line 7, the `showValues` prototype in line 11, and the `showValues` function header in line 48, there is no `&`. Remember, when you pass an array to a function you do not use an `&`.



NOTE: In C++ when a regular variable is passed to a function and an `&` precedes its name, it means that the function is receiving a reference to the memory address where a variable is stored. An array name, however, is already a memory address. That is, instead of holding a value, it holds the starting address of where the array is located in memory. Therefore, an `&` should *not* be used with it.

Some Useful Array Functions

Section 8.5 introduced you to algorithms such as summing an array and finding the highest and lowest values in an array. Now we can write general-purpose functions that perform these operations. Program 8-18, which is a modification of Program 8-10, uses the functions `sumArray`, `getHighest`, and `getLowest`.

Program 8-18

```
1  // This program gets a set of sales data from the user
2  // and displays the total, average, highest, and lowest amounts.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  // Function prototypes
8  double sumArray  (double[], int);
9  double getHighest(double[], int);
10 double getLowest (double[], int);
11
12 int main()
13 {
14    const int NUM_DAYS = 5;   // Number of days
15    double sales[NUM_DAYS],   // Holds the daily sales amounts
16           total,             // Holds the week's total sales
17           average,           // Holds the average daily sales
18           highest,           // Holds the highest daily sales
19           lowest;            // Holds the lowest daily sales
20
21    // Get the sales data
22    cout << "Enter the sales for this week.\n";
23    for (int day = 0; day < NUM_DAYS; day++)
24    {  cout << "Day " << (day + 1) <<": ";
25       cin  >> sales[day];
26    }
27
28    // Get total sales and compute average sales
29    total = sumArray(sales, NUM_DAYS);
30    average = total / NUM_DAYS;
31
32    // Get highest and lowest sales amounts
33    highest = getHighest(sales, NUM_DAYS);
34    lowest = getLowest(sales, NUM_DAYS);
35
```

(program continues)

Program 8-18 *(continued)*

```
36    // Display results
37    cout  << fixed << showpoint << setprecision(2) << endl;
38    cout  << "The total sales are         $"
39          << setw(9) << total   << endl;
40    cout  << "The average sales amount is $"
41          << setw(9) << average << endl;
42    cout  << "The highest sales amount is $"
43          << setw(9) << highest << endl;
44    cout  << "The lowest sales amount  is $"
45          << setw(9) << lowest  << endl;
46    return 0;
47 }
48
49 /**************************************************
50  *                      sumArray                  *
51  * This function computes and returns the sum of the  *
52  * values in the array whose address is passed to it. *
53  **************************************************/
54 double sumArray(double array[], int size)
55 {
56    double total = 0;  // Accumulator
57
58    for (int count = 0; count < size; count++)
59       total += array[count];
60    return total;
61 }
62
63 /**************************************************
64  *                    getHighest                  *
65  * This function finds and returns the largest value  *
66  * in the array whose address is passed to it.        *
67  **************************************************/
68 double getHighest(double array[], int size)
69 {
70    double highest = array[0];
71
72    for (int count = 1; count < size; count++)
73    {  if (array[count] > highest)
74          highest = array[count];
75    }
76    return highest;
77 }
78
79 /**************************************************
80  *                    getLowest                   *
81  * This function finds and returns the smallest value *
82  * in the array whose address is passed to it.        *
83  **************************************************/
```

(program continues)

Program 8-18 *(continued)*

```
84 double getLowest(double array[], int size)
85 {
86    double lowest = array[0];
87
88    for (int count = 1; count < size; count++)
89    {  if (array[count] < lowest)
90          lowest = array[count];
91    }
92    return lowest;
93 }
```

Program Output with Example Input Shown in Bold

```
Enter the sales for this week.
Day 1: 2698.72[Enter]
Day 2: 3757.29[Enter]
Day 3: 1109.67[Enter]
Day 4: 2498.65[Enter]
Day 5: 1489.87[Enter]
The total sales are          $ 11554.20
The average sales amount is $  2310.84
The highest sales amount is $  3757.29
The lowest sales amount  is $  1109.67
```



Checkpoint

8.15 Write a `typedef` statement that makes the name `TenInts` an alias for an array that holds 10 integers.

8.16 When an array name is passed to a function, what is actually being passed?

8.17 What is the output of the following program? (You may need to consult the ASCII table in Appendix A.)

```
#include <iostream>
using namespace std;

// Function prototypes
void fillArray(char [], int)
void showArray(char [], int)

int main ()
{  char prodCode[8] = {'0', '0', '0', '0', '0', '0', '0', '0'};

   fillArray(prodCode,8);
   showArray(prodCode,8);
   return 0;
}

// Definition of function fillArray
// (Hint: 65 is the ASCII code for 'A'.)
void fillArray(char arr[], int size)
{  char code = 65;
   for (int k = 0; k < size; code++, k++)
      arr[k] = code;
}
```

```
// Definition of function showArray
void showArray(char codes[], int size)
{  for (int k = 0; k < size; k++)
      cout << codes[k];
}
```

8.18 The following program skeleton, when completed, will ask the user to enter 10 integers which are stored in an array. The function `avgArray`, which you must write, should calculate and return the average of the numbers entered.

```
#include <iostream>
using namespace std;

// Write the avgArray function prototype here.

int main ()
{
   const int SIZE = 10;
   int userNums[SIZE];

   cout << "Enter 10 numbers: ";
   for (int count = 0; count < SIZE; count++)
   {
      cout << "#" << (count + 1) << " ";
      cin  >> userNums[count];
   }
   cout << "The average of those numbers is ";
   cout << avgArray(userNums, SIZE) << endl;
   return 0;
}

// Write the avgArray function here.
```

8.9 Two-Dimensional Arrays

CONCEPT: A two-dimensional array is like several identical arrays put together. It is useful for storing multiple sets of data.



VideoNote
Two-Dimensional Arrays

An array is useful for storing and working with a set of data. Sometimes, though, it's necessary to work with multiple sets of data. For example, in a grade-averaging program a teacher might record all of one student's test scores in an array of `doubles`. If the teacher has 30 students, that means 30 arrays of `doubles` will be needed to record the scores for the entire class. Instead of defining 30 individual arrays, however, it would be better to define a two-dimensional array.

The arrays that you have studied so far are called *one-dimensional arrays* because they can only hold one set of data. *Two-dimensional arrays,* which are also called *2D arrays*, can hold multiple sets of data. It's best to think of a two-dimensional array as a table having rows and columns of elements, as shown in Figure 8-14. This figure shows an array of test scores that has three rows and four columns.

Notice that the three rows are numbered 0 through 2 and the four columns are numbered 0 through 3. There are a total of 12 elements in the array.

Figure 8-14

	Column 0	Column 1	Column 2	Column 3
Row 0	score[0][0]	score[0][1]	score[0][2]	score[0][3]
Row 1	score[1][0]	score[1][1]	score[1][2]	score[1][3]
Row 2	score[2][0]	score[2][1]	score[2][2]	score[2][3]

To define a two-dimensional array, two size declarators are required: the first one is for the number of rows and the second one is for the number of columns. Here is an example definition of a two-dimensional array with three rows and four columns:

```
double score[3][4];
```

Rows Columns

Notice that each number is enclosed in its own set of brackets.

For processing the information in a two-dimensional array, each element has two subscripts: one for its row and another for its column. In the `score` array, the elements in row 0 are referenced as

```
score[0][0]
score[0][1]
score[0][2]
score[0][3]
```

The elements in row 1 are

```
score[1][0]
score[1][1]
score[1][2]
score[1][3]
```

And the elements in row 2 are

```
score[2][0]
score[2][1]
score[2][2]
score[2][3]
```

The subscripted references are used in a program just like the references to elements in a one-dimensional array. For example, the following statement assigns the value 92.25 to the element at row 2, column 1 of the `score` array:

```
score[2][1] = 92.25;
```

And the following statement displays the element at row 0, column 2:

```
cout << score[0][2];
```

Programs that cycle through each element of a two-dimensional array usually do so with nested loops. Program 8-19 shows an example.

Program 8-19

```
// This program uses a two-dimensional array. The
// data stored in the array is read in from a file.
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;

int main()
{
   const int NUM_DIVS = 3;          // Number of divisions
   const int NUM_QTRS = 4;          // Number of quarters
   double sales[NUM_DIVS][NUM_QTRS];// 2D array with 3 rows & 4 columns
   double totalSales = 0;           // Accumulates total sales
   int div, qtr;                    // Loop counters
   ifstream datafile;               // Used to read data from a file

   datafile.open("sales2.dat");
   if (!datafile)
      cout << "Error opening data file.\n";
   else
   {
      cout << fixed << showpoint << setprecision(2);
      cout << "Quarterly Sales by Division\n\n";

      // Nested loops are used to fill the array with quarterly
      // sales figures for each division and to display the data
      for (div = 0; div < NUM_DIVS; div++)
      {  for (qtr = 0; qtr < NUM_QTRS; qtr++)
         {
            cout << "Division "  << (div + 1)
                 << ", Quarter " << (qtr + 1) << ": $";
            datafile >> sales[div][qtr];
            cout << sales[div][qtr] << endl;
         }
         cout << endl;              // Print blank line
      }
      datafile.close();

      // Nested loops are used to add all the elements
      for (div = 0; div < NUM_DIVS; div++)
      {  for (qtr = 0; qtr < NUM_QTRS; qtr++)
            totalSales += sales[div][qtr];
      }
      // Display the total
      cout << "The total sales for the company are: $";
      cout << totalSales << endl;
   }
   return 0;
}
```

(program continues)

Program 8-19 *(continued)*

Program Output

```
Quarterly Sales by Division

Division 1, Quarter 1: $31569.45
Division 1, Quarter 2: $29654.23
Division 1, Quarter 3: $32982.54
Division 1, Quarter 4: $39651.21

Division 2, Quarter 1: $56321.02
Division 2, Quarter 2: $54128.63
Division 2, Quarter 3: $41235.85
Division 2, Quarter 4: $54652.33

Division 3, Quarter 1: $29654.35
Division 3, Quarter 2: $28963.32
Division 3, Quarter 3: $25353.55
Division 3, Quarter 4: $32615.88

The total sales for the company are: $456782.34
```

As with one-dimensional arrays, two-dimensional arrays can be initialized when they are created. When initializing a two-dimensional array, it helps to enclose each row's initialization list in a set of braces. Here is an example:

```
int hours[3][2] = {{8, 5}, {7, 9}, {6, 3}};
```

The same statement could also be written as

```
int hours[3][2] = {{8, 5},
                   {7, 9},
                   {6, 3}};
```

In either case, the values are assigned to hours in the following manner:

```
hours[0][0] is set to 8
hours[0][1] is set to 5
hours[1][0] is set to 7
hours[1][1] is set to 9
hours[2][0] is set to 6
hours[2][1] is set to 3
```

Figure 8-15 illustrates the initialization.

Figure 8-15

	Column 0	Column 1
Row 0	8	5
Row 1	7	9
Row 2	6	3

The extra braces that enclose each row's initialization list are optional. Both of the following statements perform the same initialization:

```
int hours[3][2] = {{8, 5}, {7, 9}, {6, 3}};
int hours[3][2] = {8, 5, 7, 9, 6, 3};
```

Because the extra braces visually separate each row, however, it's a good idea to use them. In addition, the braces give you the ability to leave out initializers within a row without omitting the initializers for the rows that follow it. For instance, look at the following array definition:

```
int table[3][2] = {{1}, {3, 4}, {5}};
```

`table[0][0]` is initialized to 1, `table[1][0]` is initialized to 3, `table[1][1]` is initialized to 4, and `table[2][0]` is initialized to 5. The uninitialized elements (in this case `table[0][1]` and `table[2][1]`) are automatically set to zero.

Passing Two-Dimensional Arrays to Functions

Program 8-20 illustrates how to pass a two-dimensional array to a function. When a two-dimensional array is passed to a function, the parameter type must contain a size declarator for the number of columns. C++ needs this information to correctly translate a subscripted array reference, such as `table[2][1]`, to the address in memory where that element is stored. Here is the header for the function `showArray`, from Program 8-20:

```
void showArray(int array[][NUM_COLS], int numRows)
```

The `showArray` function can accept any two-dimensional integer array, as long as it has four columns. In Program 8-20, the contents of two separate arrays are displayed by this function.

Program 8-20

```
1  // This program demonstrates how to pass a two-dimensional array
2  // to a function.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  const int NUM_COLS  = 4;               // Number of columns in each array
8  const int TBL1_ROWS = 3;               // Number of rows in table1
9  const int TBL2_ROWS = 4;               // Number of rows in table2
10
11 void showArray(int [][NUM_COLS], int); // Function prototype
12
13 int main()
14 {
15    int table1[TBL1_ROWS][NUM_COLS] = {{1,  2,  3,  4},
16                                       {5,  6,  7,  8},
17                                       {9, 10, 11, 12}};
18
```

(program continues)

Program 8-20 *(continued)*

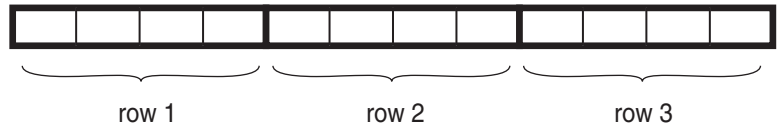
```
19    int table2[TBL2_ROWS][NUM_COLS] = {{ 10,  20,  30,  40},
20                                       { 50,  60,  70,  80},
21                                       { 90, 100, 110, 120},
22                                       {130, 140, 150, 160}};
23
24    cout << "The contents of table1 are:\n";
25    showArray(table1, TBL1_Rows);
26    cout << "\nThe contents of table2 are:\n";
27    showArray(table2, TBL2_Rows);
28    return 0;
29 }
30 /*************************************************************
31  *                          showArray                        *
32  * This function displays the contents of a 2-D integer array. *
33  * The first parameter is passed the address of the array,     *
34  * which has NUM_COLS columns. The second parameter, numRows,  *
35  * is passed the number of rows in the array.                  *
36  *************************************************************/
37 void showArray(int array[][NUM_COLS], int numRows)
38 {
39    for (int row = 0; row < numRows; row++)
40    {  for (int col = 0; col < NUM_COLS; col++)
41       {
42          cout << setw(5) << array[row][col] << " ";
43       }
44       cout << endl;
45    }
46 }
```

Program Output

```
The contents of table1 are:
    1    2    3    4
    5    6    7    8
    9   10   11   12

The contents of table2 are:
   10   20   30   40
   50   60   70   80
   90  100  110  120
  130  140  150  160
```

C++ requires the columns to be specified in the function prototype and header because of the way two-dimensional arrays are stored in memory. One row actually follows another, as shown in Figure 8-16.

Figure 8-16



When the compiler generates code for accessing the elements of a two-dimensional array, it needs to know how many bytes separate the rows in memory. The number of columns is a critical factor in this calculation.

This required column information can also be provided with a `typedef` statement. Here is how a `typedef` declaration for a two-dimensional array might look:

```
typedef int intTable[][4];
```

This statement makes `intTable` an alias for a two-dimensional array with any number of rows and four columns. If this `typedef` statement had been included in Program 8-20, the prototype for the `showArray` function could then have been written as

```
void showArray(intTable, int);
```

and its function header could have been written as

```
void showArray(intTable array, int numRows)
```

Summing All the Elements of a Two-Dimensional Array

To sum all the elements of a two-dimensional array, you can use a pair of nested loops to add the contents of each element to an accumulator. The following code shows an example.

```
const int NUM_ROWS = 3;          // Number of rows
const int NUM_COLS = 5;          // Number of columns
int total = 0;                   // Accumulator
int numbers[NUM_ROWS][NUM_COLS] =  {{2, 7, 9, 6, 4},
                                    {6, 1, 8, 9, 4},
                                    {4, 3, 7, 2, 9}};
// Sum the array elements
for (int row = 0; row < NUM_ROWS; row++)
{   for (int col = 0; col < NUM_COLS; col++)
       total += numbers[row][col];
}
// Display the sum
cout << "The total is " << total << endl;
```

Summing the Rows of a Two-Dimensional Array

Sometimes you need to calculate the sum of each row in a two-dimensional array. For example, suppose a two-dimensional array is used to hold a set of test scores for a group of students. Each row in the array is a set of scores for one student. To sum the scores for each student, you again use a pair of nested loops. The inner loop is used to add all the scores in a row, that is, all the scores for one student. The outer loop is executed once for each student. The following code shows an example.

```
const int NUM_STUDENTS = 3;      // Number of students
const int NUM_SCORES = 5;        // Number of test scores
double total;                    // Accumulator
double average;                  // Holds a given student's average
double scores[NUM_STUDENTS][NUM_SCORES] = {{88, 97, 79, 86, 94},
                                           {86, 91, 78, 79, 84},
                                           {82, 73, 77, 82, 89}};
```

```
// Sum each student's test scores so his or her
// average can be calculated and displayed
for (int row = 0; row < NUM_STUDENTS; row ++)
{
   // Reset accumulator to 0 for this student
   total = 0;

   // Sum a row
   for (int col = 0; col < NUM_SCORES; col++)
      total += scores[row][col];

   // Compute and display the average for this student
   average = total / NUM_SCORES;
   cout << "Score average for student "
        << (row + 1) << " is " << average << endl;
}
```

Summing the Columns of a Two-Dimensional Array

Sometimes you may need to calculate the sum of each column in a two-dimensional array. Using the array of test scores from the previous example, suppose you wish to calculate the class average for each of the tests. To do this, you must calculate the average of each column in the array. As in the previous example, this is accomplished with a set of nested loops. However, now the order of the two loops is reversed. The inner loop is used to add all the scores in a column, that is, all the scores for one test. The outer loop is executed once for each test. The following code illustrates this.

```
const int NUM_STUDENTS = 3;    // Number of students
const int NUM_SCORES = 5;      // Number of test scores
double total;                  // Accumulator
double average;                // Holds average score on a given test
double scores[NUM_STUDENTS][NUM_SCORES] = {{88, 97, 79, 86, 94},
                                           {86, 91, 78, 79, 84},
                                           {82, 73, 77, 82, 89}};
// Calculate and display the class
// average for each test
for (int col = 0; col < NUM_SCORES; col++)
{
   // Reset accumulator to 0 for this test
   total = 0;

   // Sum a column
   for (int row = 0; row < NUM_STUDENTS; row++)
       total += scores[row][col];
   // Compute and display the class average for this test
   average = total / NUM_STUDENTS;
   cout << "Class average for test " << (col + 1)
        << " is " << average << endl;
}
```

8.10 Arrays with Three or More Dimensions

CONCEPT: C++ permits arrays to have multiple dimensions.

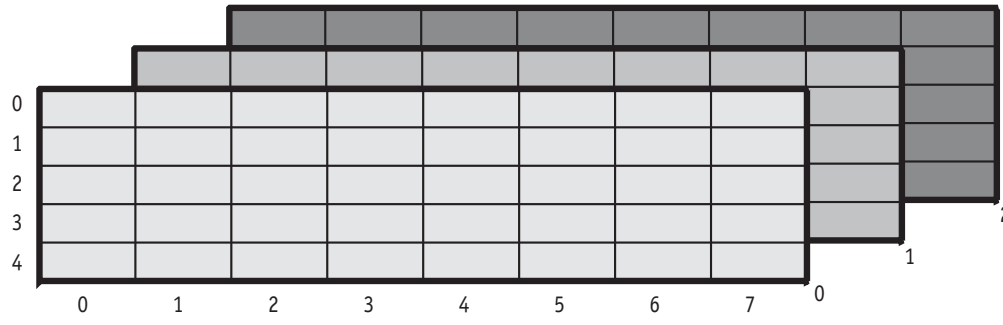
C++ allows you to create arrays with virtually any number of dimensions. Here is an example of a three-dimensional (3D) array definition:

```
double seat[3][5][8];
```

This array can be thought of as three sets of five rows, with each row containing eight elements. The array might be used, for example, to store the price of seats in an auditorium that has three sections of seats, with five rows of eight seats in each section.

Figure 8-17 illustrates the concept of a three-dimensional array as "pages" of two-dimensional arrays.

Figure 8-17



Arrays with more than three dimensions are difficult to visualize but can be useful in some programming problems.

When writing functions that accept multidimensional arrays as arguments, you must explicitely state all but the first dimension in the parameter list. If the `seat` array, defined here, were passed to a `displaySeats` function, its prototype and function header might look like the following:

```
// Function prototype
void displaySeats(double [][5][8], int);

// Function header
void displaySeats(double array[][5][8], int numGroups);
```

As with one-dimensional and two-dimensional arrays, the parameter lists can be simplified if a `typedef` statement is used to create an alias for the array type. This is demonstrated in Program 8-21, which uses the `seat` array to store theater seat prices. The information to populate the array is read in from a file. The information on number of sections, number of rows in a section, and number of seats in a row is stored in global constants, rather than being passed to the functions.

Program 8-21

```
// This program stores and displays theater seat prices.
// It demonstrates how to pass a 3-dimensional array to a function.
// The data is read in from a file.
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;

const int NUM_SECTIONS = 3,
          ROWS_IN_SECTION = 5,
          SEATS_IN_ROW = 8;

typedef double seatTable[][ROWS_IN_SECTION][SEATS_IN_ROW];

// Function prototypes
void fillArray(seatTable);
void showArray(seatTable);

int main()
{
   // Define 3-D array to hold seat prices
   double seats[NUM_SECTIONS][ROWS_IN_SECTION][SEATS_IN_ROW];

   fillArray(seats);
   showArray(seats);
   return 0;
}

/**************************************************
 *                    fillArray                   *
 * This function receives the address of a 3-D array *
 * and fills it with data read in from a file.    *
 **************************************************/
void fillArray(seatTable array)
{
   ifstream dataIn;
   dataIn.open("seats.dat");

   if (!dataIn)
      cout << "Error opening file.\n";
   else
   {  for (int section = 0; section < NUM_SECTIONS; section++)
         for (int row = 0; row < ROWS_IN_SECTION; row++)
            for (int seat = 0; seat < SEATS_IN_ROW; seat++)
               dataIn >> array[section][row][seat];

      dataIn.close();
   }
}

```

(program continues)

Program 8-21 *(continued)*

```
51 /*************************************************
52  *                    showArray                  *
53  * This function displays the contents of the 3-D *
54  * array of doubles whose address is passed to it.*
55  *************************************************/
56 void showArray(seatTable array)
57 {
58    cout << fixed << showpoint << setprecision(2);
59
60    for (int section = 0; section < NUM_SECTIONS; section++)
61    {
62       cout << "\n\nSection" << (section+1);
63       for (int row = 0; row < ROWS_IN_SECTION; row++)
64       {
65          cout << "\nRow " << (row+1) << ": ";
66          for (int seat = 0; seat < SEATS_IN_ROW; seat++)
67             cout << setw(7) << array[section][row][seat];
68       }
69    }
70    cout << endl;
71 }
```

Program Output

```
Section1
Row 1:   18.00  18.00  18.00  18.00  18.00  18.00  18.00  18.00
Row 2:   15.00  15.00  15.00  15.00  15.00  15.00  15.00  15.00
Row 3:   15.00  15.00  15.00  15.00  15.00  15.00  15.00  15.00
Row 4:   15.00  15.00  15.00  15.00  15.00  15.00  15.00  15.00
Row 5:   12.00  12.00  12.00  12.00  12.00  12.00  12.00  12.00

Section2
Row 1:   12.00  12.00  12.00  12.00  12.00  12.00  12.00  12.00
Row 2:   12.00  12.00  12.00  12.00  12.00  12.00  12.00  12.00
Row 3:   12.00  12.00  12.00  12.00  12.00  12.00  12.00  12.00
Row 4:   10.00  10.00  10.00  10.00  10.00  10.00  10.00  10.00
Row 5:   10.00  10.00  10.00  10.00  10.00  10.00  10.00  10.00

Section3
Row 1:    8.00   8.00  10.00  10.00  10.00  10.00   8.00   8.00
Row 2:    8.00   8.00  10.00  10.00  10.00  10.00   8.00   8.00
Row 3:    5.00   5.00   8.00   8.00   8.00   8.00   5.00   5.00
Row 4:    5.00   5.00   8.00   8.00   8.00   8.00   5.00   5.00
Row 5:    5.00   5.00   8.00   8.00   8.00   8.00   5.00   5.00
```



Checkpoint

8.19 Define a two-dimensional array of `int`s named `grades`. It should have 30 rows and 10 columns.

8.20 How many elements are in the following array?

```
double sales[6][4];
```

8.21 Write a statement that assigns the value 56893.12 to the first column of the first row of the `sales` array defined in question 8.20.

8.22 Write a statement that displays the contents of the last column of the last row of the `sales` array defined in question 8.20.

8.23 Define a two-dimensional array named `settings` large enough to hold the table of information below. Initialize the array with the values in the table.

12	24	32	21	42
14	67	87	65	90
19	1	24	12	8

8.24 Fill in the empty table below so it shows the contents of the following array:

```
int table[3][4] = {{2, 3}, {7, 9, 2}, {1}};
```


8.25 Write a function called `displayArray7`. The function should accept a two-dimensional array as an argument and display its contents on the screen. The function should work with any of the following arrays:

```
int hours[5][7];
int stamps[8][7];
int autos[12][7];
```

8.26 A DVD rental store keeps DVDs on 50 racks with 10 shelves each. Each shelf holds 25 DVDs. Define a 3D array to represent this storage system.

8.11 Vectors

CONCEPT: The Standard Template Library offers a `vector` data type, which in many ways is superior to standard arrays.

The *Standard Template Library* (STL) is a collection of data types and algorithms that you may use in your C++ programs. These data types and algorithms are *programmer-defined*. They are not part of the C++ language but were created in addition to the built-in data types. If you plan to continue your studies in the field of computer science, you should become familiar with the STL. This section introduces one of the STL data types.



NOTE: To use vectors your program header must indicate that you are using `namespace std`, since vectors are contained within that namespace. Many older compilers do not allow namespaces or support the STL.

The data types that are defined in the STL are commonly called *containers*. They are called containers because they store and organize data. There are two types of containers in the STL: sequence containers and associative containers. A *sequence container* organizes data in a sequential fashion, similar to an array. *Associative containers* organize data with keys, which allow rapid, random access to elements stored in the container.

In this section you will learn to use the `vector` data type, which is a sequence container. A vector is like a one-dimensional array in the following ways:

- A vector holds a sequence of values, or elements.
- A vector stores its elements in contiguous memory locations.
- You can use the array subscript operator `[]` to access individual elements in the vector.

However, a vector offers several advantages over arrays. Here are just a few:

- You do not have to declare the number of elements that the vector will have.
- If you add a value to a vector that is already full, the vector will automatically increase its size to accommodate the new value.
- Vectors can report the number of elements they contain.

Defining a Vector

To use vectors in your program, you must include the `vector` header file with the following statement:

```
#include <vector>
```

To create a vector object you must use a statement whose syntax is somewhat different from the syntax used in defining a regular variable or array. Here is an example:

```
vector<int> numbers;
```

This statement defines `numbers` as a vector of `int`s. Notice that the data type is enclosed in angled brackets, immediately after the word `vector`. Because a vector expands in size as you add values to it, there is no need to declare a size. However, you can declare a starting size, if you prefer. Here is an example:

```
vector<int> numbers(10);
```

This statement defines `numbers` as a vector of 10 `int`s, but this is only a starting size. Its size will expand if you add more than 10 values to it.



NOTE: Notice that if you specify a starting size for a vector, the size declarator is enclosed in parentheses, not square brackets.

When you specify a starting size for a vector, you may also specify an initialization value. The initialization value is copied to each element. Here is an example:

```
vector<int> numbers(10, 2);
```

In this statement, `numbers` is defined as a vector of 10 `int`s. Each element in `numbers` is initialized to the value 2.

You may also initialize a vector with the values in another vector. For example, if `set1` is a vector of `int`s that already has values in it, the following statement will create a new vector, named `set2`, which is an exact copy of `set1`.

```
vector<int> set2(set1);
```

After this statement executes, the vector `set2` will have the same number of elements and hold the same set of values as `set1`.

Table 8-2 summarizes the vector definition procedures we have discussed.

Table 8-2 Example Vector Definitions

Definition Format	Description
`vector<float> amounts;`	Defines `amounts` as an empty vector of `float`s.
`vector<int> scores(15);`	Defines `scores` as a vector of 15 `int`s.
`vector<char> letters(25, 'A');`	Defines `letters` as a vector of 25 characters. Each element is initialized with 'A'.
`vector<double> values2(values1);`	Defines `values2` as a vector of `double`s. All the elements of `values1`, which is also a vector of `double`s, are copied to `value2`.

Storing and Retrieving Values in a Vector

To store a value in an element that already exists in a vector, you may use the array subscript operator `[]`. Program 8-22, which is a modification of Program 8-13, illustrates this.

Program 8-22

```
1  // This program stores employee hours worked and hourly pay rates
2  // in vectors.
3  #include <iostream>
4  #include <iomanip>
5  #include <vector>                       // Needed to use vectors
6  using namespace std;
7
8  int main()
9  {
10    const int NUM_EMPS = 5;              // Number of employees
11    vector <int> hours(NUM_EMPS);        // Define a vector of integers
12    vector <double> payRate(NUM_EMPS);   // Define a vector of doubles
13    double grossPay;
14    int index;                           // Loop counter
15
16    // Get employee work data
17    cout << "Enter the hours worked and hourly pay rates of "
18         << NUM_EMPS << " employees. \n";
19
```

(program continues)

Program 8-22 *(continued)*

```
20    for (index = 0; index < NUM_EMPS; index++)
21    {
22       cout << "Hours worked by employee #" << (index + 1) << ": ";
23       cin  >> hours[index];
24       cout << "Hourly pay rate for employee #" << (index + 1) << ": ";
25       cin  >> payRate[index];
26    }
27    // Display each employee's gross pay
28    cout << "\nHere is the gross pay for each employee:\n";
29    cout << fixed << showpoint << setprecision(2);
30    for (index = 0; index < NUM_EMPS; index++)
31    {
32       grossPay = hours[index] * payRate[index];
33       cout << "Employee #" << (index + 1);
34       cout << ": $" << setw(7) << grossPay << endl;
35    }
36    return 0;
37 }
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked and hourly pay rates of 5 employees.
Hours worked by employee #1: 10[Enter]
Hourly pay rate for employee #1: 9.75[Enter]
Hours worked by employee #2: 15[Enter]
Hourly pay rate for employee #2: 8.62[Enter]
Hours worked by employee #3: 20[Enter]

Hourly pay rate for employee #3: 10.50[Enter]
Hours worked by employee #4: 40[Enter]
Hourly pay rate for employee #4: 18.75[Enter]
Hours worked by employee #5: 40[Enter]
Hourly pay rate for employee #5: 15.65[Enter]

Here is the gross pay for each employee:
Employee #1: $  97.50
Employee #2: $ 129.30
Employee #3: $ 210.00
Employee #4: $ 750.00
Employee #5: $ 626.00
```

Notice that Program 8-22 uses the following statements in lines 11 and 12 to define two vectors:

```
vector<int> hours(NUM_EMPS);       // Define a vector of integers
vector<double> payRate(NUM_EMPS);  // Define a vector of doubles
```

Because the named constant NUM_EMPS equals 5, both vectors are defined with the starting size 5. The program uses the following loop in lines 20 through 26 to store a value in each element of both vectors:

```
for (index = 0; index < NUM_EMPS; index++)
{
   cout << "Hours worked by employee #" << (index + 1) << ": ";
   cin  >> hours[index];
   cout << "Hourly pay rate for employee #" << (index + 1) << ": ";
   cin  >> payRate[index];
}
```

Because the values entered by the user are being stored in vector elements that already exist, the program uses the array subscript operator `[]`, as shown in the following statements which appear in lines 23 and 25:

```
cin >> hours[index];
cin >> payRate[index];
```

Using the `push_back` Member Function

You cannot, however, use the `[]` operator to access a vector element that does not yet exist. To store a value in a vector that does not have a starting size, or that is already full, you should use the `push_back` member function. This function accepts a value as an argument, and stores it in a new element placed at the end of the vector. (It "pushes" the value at the "back" of the vector.) Here is an example that uses the `push_back` function to add an element to a vector of `int`s named `numbers`.

```
numbers.push_back(25);
```

This statement creates a new element holding 25 and places it at the end of `numbers`. If `numbers` previously had no elements, the new element becomes its single element.

Program 8-23 is a modification of Program 8-22. This version, however, allows the user to specify the number of employees. The two vectors, `hours` and `payRate`, are defined without starting sizes. Because these vectors have no starting elements, the `push_back` member function is used to store values in the vectors.

Program 8-23

```
1  // This program stores employee hours worked and hourly pay rates
2  // in two vectors. It demonstrates the use of the push_back member
3  // function to add new elements to the vectors.
4  #include <iostream>
5  #include <iomanip>
6  #include <vector>                // Needed to use vectors
7  using namespace std;
8
9  int main()
10 {
11    vector<int> hours;            // hours is an empty integer vector
12    vector<double> payRate;       // payRate is an empty double vector
13    double grossPay;
14    int numEmployees;             // Number of employees
15    int index;                    // Loop counter
16
17    // Get the number of employees
18    cout << "How many employees do you have? ";
19    cin  >> numEmployees;
20
```

(program continues)

Program 8-23 *(continued)*

```
21    // Input the payroll data
22    cout << "Enter the hours worked and hourly pay rates of the "
23         << numEmployees << " employees. \n";
24
25    for (index = 0; index < numEmployees; index++)
26    {
27       int tempHours;                    // Number of hours entered
28       double tempRate;                  // Pay rate entered
29
30       cout << "Hours worked by employee #" << (index + 1) << ": ";
31       cin  >> tempHours;
32       hours.push_back(tempHours);       // Add an element to hours
33       cout << "Hourly pay rate for employee #" << (index + 1) << ": ";
34       cin  >> tempRate;
35       payRate.push_back(tempRate);      // Add an element to payRate
36    }
37    // Display each employee's gross pay
38    cout << "\nHere is the gross pay for each employee:\n";
39    cout << fixed << showpoint << setprecision(2);
40    for (index = 0; index < numEmployees; index++)
41    {
42       grossPay = hours[index] * payRate[index];
43       cout << "Employee #" << (index + 1);
44       cout << ": $" << setw(7) << grossPay << endl;
45    }
46    return 0;
47 }
```

Program Output with Example Input Shown in Bold

```
How many employees do you have? 3[Enter]
Enter the hours worked by 3 employees and their hourly rates.
Hours worked by employee #1: 40[Enter]
Hourly pay rate for employee #1: 12.63[Enter]
Hours worked by employee #2: 25[Enter]
Hourly pay rate for employee #2: 10.35[Enter]
Hours worked by employee #3: 45[Enter]
Hourly pay rate for employee #3: 22.65[Enter]

Here is the gross pay for each employee:
Employee #1: $ 505.20
Employee #2: $ 258.75
Employee #3: $1019.25
```

Notice that the Program 8-23 loop in lines 40 through 45 which calculates and displays each employee's gross pay uses the `[]` operator to access the elements of the `hours` and `payRate` vectors. This is possible because the first loop in lines 25 through 36 already used the `push_back` member function to create the elements in the two vectors.

Determining the Size of a Vector

Unlike arrays, vectors can report the number of elements they contain. This is accomplished with the `size` member function. Here is an example of a statement that uses the `size` member function:

```
numValues = set.size();
```

In this statement, assume that `numValues` is an `int` and `set` is a vector. After the statement executes, `numValues` will contain the number of elements in `set`.

The `size` member function is especially useful when you are writing functions that accept vectors as arguments. For example, look at the following code for the `showValues` function:

```
void showValues(vector<int> vect)
{
   for (int count = 0; count < vect.size(); count++)
      cout << vect[count] << endl;
}
```

Because the vector can report its size, this function does not need a second argument indicating the number of elements in the vector. Program 8-24 demonstrates this function.

Program 8-24

```
1  // This program demonstrates the vector size member function.
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  // Function prototype
7  void showValues(vector<int>);
8
9  int main()
10 {
11    vector<int> values;
12
13    // Store a series of numbers in the vector
14    for (int count = 0; count < 7; count++)
15       values.push_back(count * 2);
16
17    // Display the numbers
18    showValues(values);
19
20    return 0;
21 }
22
```

(program continues)

Program 8-24 *(continued)*

```
23 /***************************************************************
24  *                        showValues                           *
25  * This function accepts an int vector as its sole argument, and *
26  * displays the value stored in each of the vector's elements.   *
27  ***************************************************************/
28 void showValues(vector<int> vect)
29 {
30    for (int count = 0; count < vect.size(); count++)
31       cout << vect[count] << "  ";
32    cout << endl;
33 }
```

Program Output

```
0  2  4  6  8  10  12
```

Removing Elements from a Vector

To remove the last element from a vector you can use the `pop_back` member function. The following statement removes the last element from a vector named `collection`.

```
collection.pop_back();
```

Program 8-25 demonstrates the `pop_back` function.

Program 8-25

```
1  // This program demonstrates the vector size, push_back,
2  // and pop_back member functions.
3  #include <iostream>
4  #include <vector>
5  using namespace std;
6
7  int main()
8  {
9     vector<int> values;
10
11    // Store values in the vector
12    values.push_back(1);
13    values.push_back(2);
14    values.push_back(3);
15    cout << "The size of values is " << values.size() << endl;
16
17    // Remove a value from the vector
18    cout << "Popping a value from the vector...\n";
19    values.pop_back();
20    cout << "The size of values is now " << values.size() << endl;
21
```

(program continues)

Program 8-25 *(continued)*

```
22    // Now remove another value from the vector
23    cout << "Popping a value from the vector...\n";
24    values.pop_back();
25    cout << "The size of values is now " << values.size() << endl;
26
27    // Remove the last value from the vector
28    cout << "Popping a value from the vector...\n";
29    values.pop_back();
30    cout << "The size of values is now " << values.size() << endl;
31    return 0;
32 }
```

Program Output

```
The size of values is 3
Popping a value from the vector...
The size of values is now 2
Popping a value from the vector...
The size of values is now 1
Popping a value from the vector...
The size of values is now 0
```



NOTE: The `pop_back` function is a `void` function that does not return the value being removed from the vector. The following line of code will *not* work:

```
cout << "The value being removed from the vector is "
     << values.pop_back() << endl;     // Error!
```

Clearing a Vector

To completely clear the contents of a vector, use the `clear` member function, as shown in the following example:

```
numbers.clear();
```

After this statement executes, `numbers` will be cleared of all its elements. Program 8-26 demonstrates the function.

Program 8-26

```
1 // This program demonstrates the vector clear member function.
2 #include <iostream>
3 #include <vector>
4 using namespace std;
5
```

(program continues)

Program 8-26 *(continued)*

```
6  int main()
7  {
8     vector<int> values(100);
9
10    cout << "The values vector has "
11         << values.size() << " elements.\n";
12    cout << "I will call the clear member function...\n";
13    values.clear();
14    cout << "Now the values vector has "
15         << values.size() << " elements.\n";
16    return 0;
17 }
```

Program Output

```
The values vector has 100 elements.
I will call the clear member function...
Now the values vector has 0 elements.
```

Detecting an Empty Vector

To determine if a vector is empty, use the `empty` member function. The function returns `true` if the vector is empty, and `false` if the vector has elements stored in it. Assuming `set` is a vector, here is an example of its use:

```
if (set.empty())
   cout << "No values in set.\n";
```

Program 8-27 uses a function named `avgVector`, which demonstrates the `empty` member function.

Program 8-27

```
1  // This program demonstrates the vector empty member function.
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  // Function prototype
7  double avgVector(vector<int>);
8
9  int main()
10 {
11    vector<int> values;     // Define a vector to hold int values
12    int numValues;          // Number of values to be averaged
13    double average;         // Average of the stored values
14
15    // Get the number of values to average
16    cout << "How many values do you wish to average? ";
17    cin  >> numValues;
18
```

(program continues)

Program 8-27 *(continued)*

```
19    // Get the values and store them in a vector
20    for (int count = 0; count < numValues; count++)
21    {  int tempValue;
22
23       cout << "Enter an integer value: ";
24       cin  >> tempValue;
25       values.push_back(tempValue);
26    }
27    // Get the average of the values and display it
28    average = avgVector(values);
29    cout << "Average: " << average << endl;
30    return 0;
31 }
32
33 /*************************************************************
34  *                        avgVector                          *
35  * This function accepts an int vector as its argument. If    *
36  * the vector contains values, the function returns the       *
37  * average of those values. Otherwise, an error message is    *
38  * displayed and the function returns 0.0.                    *
39  *************************************************************/
40 double avgVector(vector<int> vect)
41 {
42    int total = 0;          // Accumulator
43    double avg = 0.0;
44
45    if (vect.empty())       // Determine if the vector is empty
46         cout << "No values to average.\n";
47    else
48    {  for (int count = 0; count < vect.size(); count++)
49          total += vect[count];
50       avg = static_cast<double>(total)/vect.size();
51    }
52    return avg;
53 }
```

Program Output with Example Input Shown in Bold

```
How many values do you wish to average? 4[Enter]
Enter an integer value: 12[Enter]
Enter an integer value: 3[Enter]
Enter an integer value: 7[Enter]
Enter an integer value: 9[Enter]
Average: 7.75
```

Program Output with Different Example Input Shown in Bold

```
How many values do you wish to average? 0[Enter]
No values to average.
Average: 0
```

Summary of Vector Member Functions

Table 8-3 provides a summary of the `vector` member functions we have discussed, as well as some additional ones.

Table 8-3 Vector Member Functions

Member Function	Description
`at(`*`position`*`)`	Returns the value of the element located at *position* in the vector. *Example:* `x = vect.at(5); // Assigns the value of vect[5] to x.`
`capacity()`	Returns the maximum number of elements that may be stored in the vector without additional memory being allocated. (This is not the same value as returned by the `size` member function). *Example*: `x = vect.capacity(); // Assigns the capacity of vect to x.`
`clear()`	Clears a vector of all its elements. *Example*: `vect.clear(); // Removes all the elements from vect.`

`empty()`	Returns `true` if the vector is empty. Otherwise, it returns `false`.
	Example:
	`if (vect.empty(); // If the vector is empty`
	`  cout << "The vector is empty."; // the message is displayed.`

`pop_back()`	Removes the last element from the vector.
	Example:
	`vect.pop_back(); // Removes the last element of vect, thus`
	`// reducing its size by 1.`

`push_back(`*`value`*`)`	Stores a value in the last element of the vector. If the vector is full or empty, a new element is created. *Example*: `vect.push_back(7); // Stores 7 in the last element of vect.`
`reverse()`	Reverses the order of the elements in the vector (the last element becomes the first element, and the first element becomes the last element). *Example*: `vect.reverse(); // Reverses the order of the element in vect.`

`resize(n)`
`resize(n, value)`
	Resizes a vector to have `n` elements, where `n` is greater than the vector's current size. If the optional `value` argument is included, each of the new elements will be initialized with that value.
	Example where `vect` *currently has 4 elements:*
	`vect.resize(6,99); // Adds two elements to the end of the vector,`
	`// each initialized to 99.`

`size()`	Returns the number of elements in the vector. *Example:* `numElements = vect.size();`
`swap(`*`vector2`*`)`	Swaps the contents of the vector with the contents of *vector2*. *Example*: `vect1.swap(vect2); // Swaps the contents of vect1 and vect2.`



Checkpoint

8.27 What header file must you `#include` in order to define vector objects?

8.28 Write definition statements for the following three vector objects: `frogs` (an empty vector of `int`s), `lizards` (a vector of 20 `double`s), and `toads` (a vector of 100 `char`s, with each element initialized to `'Z'`).

8.29 Define `gators` to be an empty vector of `int`s and `snakes` to be a 10-element vector of `double`s. Then write a statement that stores the value 27 in `gators` and a statement that stores the value 12.897 in element 4 of `snakes`.

8.12 Arrays of Class Objects*

CONCEPT: Elements of arrays can be class objects.

Earlier in this chapter you learned that all the elements in an array must be of the same data type. So far we have only used arrays of simple data types, like `int` arrays and `string` arrays. However, arrays can also hold more complex data types, such as structures or objects. All that is required is that each element hold a structure of the same type or an object of the same class.

Let's look at arrays of objects. You define an array of objects the same way you define any array. If, for example, a class named `Circle` has been defined, here is how you would create an array that can hold four `Circle` objects:

```
Circle circle[4];
```

The four objects are `circle[0]`, `circle[1]`, `circle[2]`, and `circle[3]`.

Notice that the name of the class is `Circle`, with a capital C. The name of the array is `circle`, with a lowercase c. You will recall from Chapter 7, that the convention is to begin the name of a class with a capital letter and the name of a variable or object with a lowercase letter. Calling a class function for one of these objects is just like calling a class function for any other object, except that a subscript must be included to identify which of the objects in the array is being referenced. For example, the following statement would call the `findArea` function of `circle[2]`.

```
circle[2].findArea();
```

Program 8-28 illustrates these ideas by creating and using an array of `Circle` class objects. Here is the definition of the `Circle` class it uses. It is a variation of the `Circle` class introduced in Chapter 7.

* This section should be skipped if Chapter 7 has not yet been covered.

Circle.h

```
1  // This header file contains the Circle class declaration.
2  #ifndef CIRCLE_H
3  #define CIRCLE_H
4  #include <cmath>
5
6  class Circle
7  {  private:
8        double radius;                    // Circle radius
9        int centerX, centerY;             // Center coordinates
10
11    public:
12       Circle()                          // Default constructor
13       {  radius = 1.0;                  // accepts no arguments
14          centerX = centerY = 0;
15       }
16
17       Circle(double r)                  // Constructor 2
18       {  radius = r;                    // accepts 1 argument
19          centerX = centerY = 0;
20       }
21
22       Circle(double r, int x, int y)    // Constructor 3
23       {  radius = r;                    // accepts 3 arguments
24          centerX = x;
25          centerY = y;
26       }
27
28       void setRadius(double r)
29       {  radius = r;
30       }
31
32       int getXcoord()
33       {  return centerX;
34       }
35
36       int getYcoord()
37       {  return centerY;
38       }
39
40       double findArea()
41       {  return 3.14 * pow(radius, 2);
42       }
43 }; // End Circle class declaration
44 #endif
```

Program 8-28 creates an array of four `Circle` objects in line 12, then uses a loop in lines 15 through 20 to call the `setRadius` method for each object. A second loop is used in lines 26 through 29 to call the `findArea` method for each object and display the result.

Program 8-28

```
// This program uses an array of objects.
// The objects are instances of the Circle class.
#include <iostream>
#include <iomanip>
#include "Circle.h"    // Needed to create Circle objects
using namespace std;

const int NUM_CIRCLES = 4;

int main()
{
   Circle circle[NUM_CIRCLES];  // Define an array of Circle objects

   // Use a loop to initialize the radius of each object
   for (int index = 0; index < NUM_CIRCLES; index++)
   {  double r;
      cout << "Enter the radius for circle " << (index+1) << ": ";
      cin  >> r;
      circle[index].setRadius(r);
   }

   // Use a loop to get and print out the area of each object
   cout  << fixed << showpoint << setprecision(2);
   cout  << "\nHere are the areas of the " << NUM_CIRCLES
         << " circles.\n";
   for (int index = 0; index < NUM_CIRCLES; index++)
   {  cout  << "circle " << (index+1) << setw(8)
            << circle[index].findArea() << endl;
   }
   return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the radius for circle 1: 0[Enter]
Enter the radius for circle 2: 2[Enter]
Enter the radius for circle 3: 2.5[Enter]
Enter the radius for circle 4: 10[Enter]

Here are the areas of the 4 circles.
circle 1    0.00
circle 2   12.56
circle 3   19.63
circle 4  314.00
```



NOTE: Whenever an array of objects is created with no constructor arguments, the default constructor, if one exists, runs for every object in the array. This occurred in Program 8-28.

When the array of `Circle` objects was first created, the default constructor executed for each object in the array and assigned its radius the value 1.0. We never saw this because the call made to the `setRadius` member function of each object replaced its 1.0 with the

new value passed to `setRadius`. If we commented out lines 15 through 20 of Program 8-30, no calls would be made to `setRadius`. So every object in the array would still have a radius of 1.0 when the loop on lines 26 through 29 gets and prints the area. The output would look like this:

```
Here are the areas of the 4 circles.
circle 1    3.14
circle 2    3.14
circle 3    3.14
circle 4    3.14
```

This version of Program 8-28 can be found on the student CD with the name `pr8-28B.cpp`.

It is also possible to create an array of objects and have another constructor called for each object. To do this you must use an initialization list. The following array definition and initialization list creates four `Circle` objects and initializes them to the same four values that were input in the original Program 8-28 sample run.

```
Circle circle[NUM_CIRCLES] = {0.0, 2.0, 2.5, 10.0};
```

This invokes the constructor that accepts one `double` argument and sets the radii shown here.

Object	radius
circle[0]	0.0
circle[1]	2.0
circle[2]	2.5
circle[3]	10.0

If the initialization list had been shorter than the number of objects, any remaining objects would have been initialized by the default constructor. For example, the following statement invokes the constructor that accepts one `double` argument for the first three objects and causes the default constructor to run for the fourth object. The fourth object is assigned a default radius of 1.0.

```
Circle circle[NUM_CIRCLES] = {0.0, 2.0, 2.5};
```

This is illustrated in Program 8-29.

Program 8-29

```
1  // This program demonstrates how an overloaded constructor
2  // that accepts an argument can be invoked for multiple objects
3  // when an array of objects is created.
4  #include <iostream>
5  #include <iomanip>
6  #include "Circle.h"    // Needed to create Circle objects
7  using namespace std;
8
9  const int NUM_CIRCLES = 4;
10
```

(program continues)

Program 8-29 *(continued)*

```
11 int main()
12 {
13    // Define an array of 4 Circle objects. Use an initialization list
14    // to call the 1-argument constructor for the first 3 objects.
15    // The default constructor will be called for the final object.
16    Circle circle[NUM_CIRCLES] = {0.0, 2.0, 2.5};
17
18    // Display the area of each object
19    cout  << fixed << showpoint << setprecision(2);
20    cout  << "\nHere are the areas of the " << NUM_CIRCLES
21          << " circles.\n";
22
23    for (int index = 0; index < NUM_CIRCLES; index++)
24    {  cout  << "circle " << (index+1) << setw(8)
25             << circle[index].findArea() << endl;
26    }
27    return 0;
28 }
```

Program Output

```
Here are the areas of the 4 circles.
circle 1    0.00
circle 2   12.56
circle 3   19.63
circle 4    3.14
```

To use a constructor that requires more than one argument, the initializer must take the form of a function call. For example, look at the following definition statement. It invokes the 3-argument constructor for each of three `Circle` objects.

```
Circle circle[3] = { Circle(4.0, 2,  1),
                     Circle(2.0, 1,  3),
                     Circle(2.5, 5, -1) };
```

`circle[0]` will have its `radius` variable set to 4.0, its `centerX` variable set to 2, and its `centerY` variable set to 1. `circle[1]` will have its `radius` variable set to 2.0, its `centerX` variable set to 1, and its `centerY` variable set to 3. `circle[2]` will have its `radius` variable set to 2.5, its `centerX` variable set to 5, and its `centerY` variable set to -1.

It isn't necessary to call the same constructor for each object in an array. For example, look at the following statement:

```
Circle circle[3] = { 4.0,
                     Circle(2.0, 1,  3),
                     2.5 };
```

This statement invokes the 1-argument constructor for `circle[0]` and `circle[2]` and the 3-argument constructor for `circle[1]`.

In summary, there are seven key points to remember about arrays of objects.

1. The elements of arrays can be objects.
2. If you do not use an initialization list when an array of objects is created, the default constructor will be invoked for each object in the array.
3. It is not necessary that all objects in the array use the same constructor.
4. If you do use an initialization list when an array of objects is created, the correct constructor will be called for each object, depending on the number and type of arguments used.
5. If a constructor requires more than one argument, the initializer must take the form of a constructor function call.
6. If there are fewer initializer calls in the list than there are objects in the array, the default constructor will be called for all the remaining objects.
7. It is best to always provide a default constructor; but if there is none you must be sure to furnish an initializer for every object in the array.

These seven statements also apply to arrays of structures, which we will look at more closely in the next few pages.



Checkpoint

8.30 True or false: The default constructor is the only constructor that may be called for objects in an array of objects.

8.31 True or false: All elements in an array of objects must use the same constructor.

8.32 What will the following program display on the screen?

```
#include <iostream>
using namespace std;

class Tank
{
private:
   int gallons;
public:
   Tank()
     { gallons = 50; }
   Tank(int gal)
     { gallons = gal; }
   int getGallons()
     { return gallons; }
};

int main ()
{
   Tank storage[3] = { 10, 20 };

   for (int index = 0; index < 3; index++)
      cout << storage[index].getGallons() << endl;
   return 0;
}
```

8.33 Complete the following program so it defines an array of 10 `Yard` objects. The program should use a loop to ask the user for the length and width of each yard. Then it should loop and display the length and width of each yard. To do this you will need to add two member functions to the `Yard` class.

```
#include <iostream>
using namespace std;

class Yard
{
private:
   int length, width;
public:
   Yard()
      { length = 0; width = 0; }
   void  setLength(int l)
     { length = l; }
   void  setWidth(int w)
     { width = w; }
};

int main ()
{
   // Finish this program.
}
```

Arrays of Structures

As mentioned earlier in this section, array elements can also be structures. This is useful when you want to store a collection of records that hold multiple data fields, but you aren't using objects. Program 8-13, which we saw earlier in this chapter, showed how related information of different data types can be stored in parallel arrays. These are two or more arrays with a relationship established between them through their subscripts. Because structures can hold multiple items of varying data types, a single array of structures can be used in place of several arrays of regular variables.

An array of structures is defined like any other array. Assume the following structure declaration exists in a program:

```
struct BookInfo
{
   string title;
   string author;
   string publisher;
   double price;
};
```

The following statement defines an array, `bookList`, which has 20 elements. Each element is a `BookInfo` structure.

```
BookInfo bookList[20];
```

Each element of the array may be accessed through a subscript. For example, `bookList[0]` is the first structure in the array, `bookList[1]` is the second, and so forth. Because members of structures are public by default, you do not need to use a function, as you do with class objects, to access them. You can access a member of any element by simply placing the dot operator and member name after the subscript. For example, the following expression refers to the `title` member of `bookList[5]`:

```
bookList[5].title
```

The following loop steps through the array, displaying the information stored in each element:

```
for (int index = 0; index < 20; index++)
{
   cout << bookList[index].title << endl;
   cout << bookList[index].author << endl;
   cout << bookList[index].publisher << endl;
   cout << bookList[index].price << endl << endl;
}
```

Because the members `title`, `author`, and `publisher` are string objects their individual elements can be accessed as well. The following statement displays the first character of the `title` member of `bookList[10]`:

```
cout << bookList[10].title[0];
```

And the following statement stores the character 't' in the fourth position of the `publisher` member of `bookList[2]`:

```
bookList[2].publisher[3] = 't';
```

Program 8-30 is a modification of Program 8-13 which calculates and displays payroll information for a set of employees. The original program used two parallel arrays to hold the hours and pay rates of the employees. This modified version uses a single array of structures.

Program 8-30

```
1  // This program uses an array of structures to hold payroll data.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  struct PayInfo
7  {
8     int hours;                  // Hours worked
9     double payRate;             // Hourly pay rate
10 };
11
12 int main ()
13 {
14    const int NUM_EMPS = 3;     // Number of employees
15    int index;
16    PayInfo workers[NUM_EMPS];  // Define an array of structures
17    double grossPay;
18
19    // Get payroll data
20    cout  << "Enter the hours worked and hourly pay rates of "
21          << NUM_EMPS << " employees.";
```

(program continues)

Program 8-30 *(continued)*

```
22    for (index = 0; index < NUM_EMPS; index++)
23    {
24       cout << "\n Hours worked by employee #" << (index + 1);
25       cout << ": ";
26       cin  >> workers[index].hours;
27       cout << "Hourly pay rate for employee #";
28       cout << (index + 1) << ": ";
29       cin  >> workers[index].payRate;
30    }
31
32    // Display each employee's gross pay
33    cout << "\nHere is the gross pay for each employee:\n";
34    cout << fixed << showpoint << setprecision(2);
35    for (index = 0; index < NUM_EMPS; index++)
36    {
37       grossPay = workers[index].hours * workers[index].payRate;
38       cout << "Employee #" << (index + 1);
39       cout << ": $" << setw(7) << grossPay << endl;
40    }
41    return 0;
42 }
```

Program Output with Example Input Shown in Bold

```
Enter the hours worked and hourly pay rates of 3 employees.
Hours worked by employee #1: 10[Enter]
Hourly pay rate for employee #1: 9.75[Enter]

Hours worked by employee #2: 20[Enter]
Hourly pay rate for employee #2: 10.00[Enter]

Hours worked by employee #3: 40[Enter]
Hourly pay rate for employee #3: 20.00[Enter]

Here is the gross pay for each employee:
Employee #1: $  97.50
Employee #2: $ 200.00
Employee #3: $ 800.00
```

You can initialize an array of structures the same way you initialize an array of class objects, with a constructor. Here is the structure declaration from Program 8-30 modified to include a constructor. It accepts two arguments, but also has default values in case a structure variable is created without passing any values to the constructor.

```
struct PayInfo
{
   int hours;                          // Hours worked
   double payRate;                     // Hourly pay rate

   PayInfo(int h = 0, double p = 0.0)  // Constructor
   {  hours = h;
      payRate = p;
   }
};
```

Using this structure declaration, the array in Program 8-30 could now be initialized as follows:

```
PayInfo workers[NUM_EMPS] = { PayInfo(10,  9.75),
                              PayInfo(20, 10.00),
                              PayInfo(40, 20.00) };
```

Notice that the syntax for initializing members in an array of structures is the same as for initializing members in an array of objects. It is different from the syntax presented in Chapter 7 for initializing a single structure.



Checkpoint

For questions 8.34–8.38, assume the `Product` structure is declared as follows:

```
struct Product
{
   string description;  // Product description
   int partNum;         // Part number
   double cost;         // Product cost
};
```

8.34 Add two constructors to the `Product` structure declaration. The first should be a default constructor that sets the `description` member to the null string and the `partNum` and `cost` members to zero. The second constructor should have three parameters: a `string`, an `int`, and a `double`. It should copy the values of the arguments into the `description`, `partNum`, and `cost` members.

8.35 Write a definition for an array named `items` that can hold 100 `Product` structures.

8.36 Write statements that store the following information in the first element of the `items` array you defined in question 8.35.

Description: Claw Hammer
Part Number: 547
Part Cost: $8.29

8.37 Write a loop that displays the contents of the entire `items` array you created in question 8.35.

8.38 Write the definition for an array of five `Product` structures, initializing the first three elements with the following information:

Description	Part Number	Cost
Screwdriver	621	$ 1.72
Socket set	892	18.97
Claw hammer	547	8.29

8.39 Write a structure declaration called `Measurement`, with the following members:

`miles`, an integer
`meters`, a long integer

8.40 Write a structure declaration called `Destination`, with the following members:

`city`, a string object
`distance`, a `Measurement` structure (declared in Checkpoint 8.39)

8.41 Define an array of 20 `Destination` structures (see Checkpoint 8.40). Write statements that store the following information in the fifth array element:

City: Tupelo
Miles: 375
Meters: 603,375

8.13 National Commerce Bank Case Study

The National Commerce Bank has hired you as a contract programmer. Your first assignment is to write a function that will be used by the bank's automated teller machines (ATMs) to validate a customer's personal identification number (PIN).

Your function will be incorporated into a larger program that asks the customer to input his or her PIN on the ATM's numeric keypad. (PINs are four-digit numbers. The program stores each digit in an element of an `int` array.) The program also retrieves a copy of the customer's actual PIN from a database. (The PINs are also stored in the database as four element arrays.) If these two numbers match, then the customer's identity is validated. Your function should compare the two arrays and determine whether they contain the same numbers.

Here are the specifications your function must meet.

Parameters The function should accept three arguments. The first is an array holding the digits entered by the customer. The second is an array holding the digits of the customer's correct PIN, retrieved from the bank's database. The final argument indicates the number of digits in a PIN. This is set in the program to 4. However, by passing this argument to the function it makes the program easier to update in the future if the bank decides to change the PIN size.

Return value The function should return a Boolean `true` value if the two arrays are identical. Otherwise, it should return `false`.

Here is the pseudocode for the function:

```
For each element in the first array
    Compare the element with the corresponding one in the 2nd array
    If the two elements contain different values
        Return false
    End If
End For   // If we made it this far the values are the same
Return true
```

You have only been asked to write a function that performs the comparison between the customer's input and the PIN that was retrieved from the database, however, code must also be written to test it. Program 8-31 is a complete program that includes both the function and a test driver.

Program 8-31

```
// This program tests a function that compares the contents of two arrays.
#include <iostream>
using namespace std;

// Function prototype
bool testPIN(int set1[], int set2[], int size);

int main()
{
   const int NUM_DIGITS = 4;
   int pin1[NUM_DIGITS] = {2, 4, 1, 8};  // Base set of values

   int pin2[NUM_DIGITS] = {2, 4, 6, 8};  // One element is
                                         // different from PIN1.
   int pin3[NUM_DIGITS] = {1, 2, 3, 4};  // All elements are
                                         // different from PIN1.
   if (testPIN(pin1, pin2, NUM_DIGITS))
      cout << "ERROR: pin1 and pin2 are reported to be the same.\n";
   else
      cout << "SUCCESS: pin1 and pin2 are correctly identified "
           << "as different.\n";

   if (testPIN(pin1, pin3, NUM_DIGITS))
      cout << "ERROR: pin1 and pin3 are reported to be the same.\n";
   else
      cout << "SUCCESS: pin1 and pin3 are correctly identified "
           << "as different.\n";

   if (testPIN(pin1, pin1, NUM_DIGITS))
      cout << "SUCCESS: pin1 and pin1 are correctly reported "
           << "to be the same.\n";
   else
      cout << "ERROR: pin1 and pin1 are erroneously identified "
           << "as different.\n";
   return 0;
}

/*****************************************************************
 *                          testPIN                              *
 * This Boolean function accepts and compares the values stored in   *
 * two int arrays. If they both have exactly the same set of values, *
 * true is returned. If there are any differences, false is returned.*
 *****************************************************************/
bool testPIN(int custPIN[], int databasePIN[], int size)
{
   for (int index = 0; index < size; index++)
   {
      if (custPIN[index] != databasePIN[index])
         return false;        // We've found two different values
   }
   return true;                 // If we make it this far,
                                // all values are the same
}
```

(program continues)

Program 8-31 *(continued)*

Program Output

```
SUCCESS: pin1 and pin2 are correctly identified as different.
SUCCESS: pin1 and pin3 are correctly identified as different.
SUCCESS: pin1 and pin1 are correctly reported to be the same.
```

Additional Case Studies

The following additional case studies, which contain applications of material introduced in Chapter 8, can be found on the student CD.

Set Intersection Case Study

In algebra, the intersection of two sets is defined as a new set that contains those values common to the two original sets. This case study, which utilizes three one-dimensional arrays, finds and displays the intersection of two sets.

Creating an Abstract Array Data Type—Part 1

The absence of bounds checking in C++ is a source of potential hazard. This object-oriented case study develops a simple integer list class with array-like characteristics that provides bounds checking.

8.14 Tying It All Together: *Rock, Paper, Scissors*

Now that you have learned to use arrays, you can create more advanced computer games, like Rock, Paper, Scissors. You have probably played this game before. Here is how it works. Simultaneously, two players form their hand to represent one of three objects. A fist represents a rock. A flat palm represents a sheet of paper. Two extended fingers represent a pair of scissors. If the two players choose the same object, the round is a tie. Otherwise, someone wins the round. Rock beats scissors because it can break a pair of scissors. Scissors beats paper because it can cut a sheet of paper. Paper beats rock because it can wrap itself around the rock.

In this section we will create a program that lets a user play a game of Rock, Paper, Scissors with the computer. Notice how in line 9 of the Rock, Paper, Scissors program shown here, the strings holding the names of the choices are stored in an array. In line 28 the program randomly generates a 1, 2, or 3 for the computer's choice. Then, in lines 31 and 32, the human player's choice is entered:

```
cout << "Pick 1 (rock), 2 (paper), or 3 (scissors): ";
cin  >> playerChoice;
```

Notice how, for both the computer and the player, the choice number matches the array element holding the name of the object they chose. Therefore, the choice number can be used as a subscript to get the string to be displayed. With this ability, the program can easily display information for each round of the game about who chose what, what beats what, and who wins that round.

Try running the program to see if you can beat the computer.

Program 8-32

```
// This program lets the user play a game of rock, paper, scissors
// with the computer. The computer's choices are randomly generated.
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <string>
using namespace std;

const string name[4] = {" ", "rock", "paper", "scissors"};

int main()
{
   int computerChoice,
       playerChoice,
       computerPoints = 0,   // Point accumulators
       playerPoints = 0;

   srand(time(NULL));        // Give the random generator
                             // a seed to start with
   playerPoints = 0;
   computerPoints = 0;

   cout << "Let's play Rock-Paper-Scissors!\n";
   cout << "The first player to score 5 points wins.\n\n";

   do
   {  // Generate a random number 1 to 3 to simulate computer choice
      computerChoice = 1 + rand() % 3;

      // Get player's choice
      cout << "Pick 1 (rock), 2 (paper), or 3 (scissors): ";
      cin  >> playerChoice;

      if (computerChoice == playerChoice)                    // Tie
      {   cout << "I chose " << name[computerChoice]
               << " too, so we tied.\n\n";
      }
      else if ((playerChoice == 1 && computerChoice == 2) ||  // Computer
               (playerChoice == 2 && computerChoice == 3) ||  // wins
               (playerChoice == 3 && computerChoice == 1) )
      {   cout << "I chose " << name[computerChoice] << ", so I win! "
               << name[computerChoice] << " beats "
               << name[playerChoice] << ".\n\n";
          computerPoints++;
      }                                                      // Player
      else                                                   // wins
      {   cout << "I chose " << name[computerChoice] << ", so you win! "
               << name[playerChoice] << " beats "
               << name[computerChoice] << ".\n\n";
          playerPoints++;
      }
   } while (playerPoints < 5 && computerPoints < 5);

```

(program continues)

Program 8-32 *(continued)*

```
54    cout << "Let's see how you did :\n"
55         << "You won " << playerPoints << " points and I won "
56         << computerPoints << " points.\n";
57
58    if (playerPoints == 5)
59       cout << "Congratulations! You're the champ!\n";
60    else
61       cout << "Hurray for me! I'm the champ!\n";
62
63    return 0;
64 }
```

Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. The _____ indicates the number of elements, or values, an array can hold.
2. The size declarator must be a(n) _____ with a value greater than _____.
3. Each element of an array is accessed and indexed by a number known as a(n) _____.
4. Subscript numbering in C++ always starts at _____.
5. The number inside the brackets of an array definition is the _____, but the number inside an array's brackets in an assignment statement, or any other statement that works with the contents of the array, is the _____.
6. C++ has no array _____ checking, which means you can inadvertently store data past the end of an array.
7. Starting values for the elements of an array may be specified with a(n) _____ list.
8. If a numeric array is partially initialized, the uninitialized elements will bet set to _____.
9. If the size declarator of an array definition is omitted, C++ counts the number of items in the _____ to determine how large the array should be.
10. To allow an array of structures or an array of objects to be initialized, the `struct` or `class` declaration should include a(n) _____.
11. By using the same _____ for multiple arrays, you can build relationships between the data stored in the arrays. These arrays are referred to as parallel arrays.
12. You cannot use the _____ operator to copy data from one array to another in a single statement.
13. Arrays are never passed to functions by _____ because there would be too much overhead in copying all the elements.
14. To pass an array to a function, pass the _____ of the array.
15. A(n) _____ array is like several arrays of the same type put together.

16. It's best to think of a two-dimensional array as having _____ and _____.
17. To define a two-dimensional array, _____ size declarators are required.
18. When initializing a two-dimensional array, it helps to enclose each row's initialization list in _____.
19. When a two-dimensional array is passed to a function, the number of _____ must be specified.
20. To print out all elements of a two-dimensional array you would normally use a _____ loop.
21. Look at the following array definition.

```
int values[10];
```

A) How many elements does the array have?
B) What is the subscript of the first element in the array?
C) What is the subscript of the last element in the array?
D) If an `int` uses four bytes of memory, how much memory does the array use?

22. Given the following array definition:

```
int values[5] = { 4, 7, 6, 8, 2 };
```

What does the following statement display?

```
cout << values[4] << "  " << (values[2] + values[3])
     << "  "  << ++values[1] << endl;
```

23. Look at the following array definition.

```
int numbers[5] = { 1, 2, 3 };
```

A) What value is stored in `numbers[2]`?
B) What value is stored in `numbers[4]`?

24. Assume that `array1` and `array2` are both 25-element integer arrays. Indicate whether each of the following statements is legal or illegal.

A) `array1 = array2;`
B) `cout << array1;`
C) `cin  >> array2;`

25. When you pass an array name as an argument to a function, what is actually being passed?
26. How do you establish a parallel relationship between two or more arrays?
27. Look at the following array definition.

```
double sales[8][10];
```

A) How many rows does the array have?
B) How many columns does the array have?
C) How many elements does the array have?
D) Write a statement that stores 3.52 in the last column of the last row in the array.

Use the following `Car` structure declaration to answer questions 28–30.

```
struct Car
{
   string make,
          model;
   int    year;
   double cost;

   // Constructors
   Car()
   { make = model = ""; year = cost = 0; }

   Car(string mk, string md, int yr, double c)
   {  make = mk;  model = md;  year = yr;  cost = c; }
};
```

28. Define an array named `collection` that holds 25 `Car` structures.
29. Define an array named `forSale` that holds 35 `Car` structures. Initialize the first three elements with the following data:

Make	Model	Year	Cost
Ford	Taurus	2006	$21,000
Honda	Accord	2004	$11,000
Jeep	Wrangler	2007	$24,000

30. Write a loop that will step through the array you defined in question 29, displaying the contents of each element.

Algorithm Workbench

31. The arrays `array1` and `array2` each hold 25 integer elements. Write code that copies the values in `array1` to `array2`.
32. The following code totals the values in each of two arrays described in question 31. Will the code print the correct total for both arrays? Why or why not?

```
int total = 0;     // Accumulator
int count;         // Loop counter

// Calculate and display the total of the first array.
for (count = 0; count <= 25; count++)
   total += array1[count];

cout << "The total for array1 is " << total << endl;

// Calculate and display the total of the second array.
for (count = 0; count <= 25; count++)
   total += array2[count];

cout << "The total for array2 is " << total << endl;
```

33. In a program you need to store the identification numbers of 10 employees (as `int`s) and their weekly gross pay (as `double`s).
    A) Define two arrays that may be used in parallel to store the 10 employee identification numbers and 10 weekly gross pay amounts.
    B) Write a loop that uses these arrays to print each employee's identification number and weekly gross pay.

34. Revise your answer for question 33 to define and use an array of `Payroll` structures, instead of two parallel arrays. A `Payroll` structure should hold an employee identification number and weekly gross pay amount.
35. In a program you need to store the names and populations of 12 countries. Create an appropriate array to store this information and then write the code needed to read the information into the array from a file named `pop.dat`.
36. A weather analysis program uses the following array to store the temperature for each hour of the day on each day of a week.

    ```
    int temp[7][24];
    ```

    Each row represents a day (0 = Sunday, 1 = Monday, etc.) and each column represents a time (0 = midnight, 1 = 1 a.m., … , 12 = noon, 13 = 1 p.m., etc.).

    A) Write code to find Tuesday's average temperature.
    B) Write code to find the average weekly noon temperature.

Find the Errors

37. Each of the following definitions has errors. Locate as many as you can.

    A)
    ```
    int size;
    double values[size];
    ```
    B)
    ```
    int collection[-20];
    ```
    C)
    ```
    int hours[3] = 8, 12, 16;
    ```

38. Each of the following definitions has errors. Locate as many as you can.

    A)
    ```
    int numbers[8] = {1, 2, , 4, , 5};
    ```
    B)
    ```
    double ratings[];
    ```
    C)
    ```
    values[3] = {6, 8.2, 'A'};
    ```

39. Each of the following functions contains errors. Locate as many as you can.

    A)
    ```
    void showValues(int nums)
    {
       for (int count = 0; count < 8; count++)
          cout << nums[count];
    }
    ```
    B)
    ```
    void showValues(int nums[4][])
    {
       for (rows = 0; rows < 4; rows++)
          for (cols = 0; cols < 5; cols++)
             cout << nums[rows][cols];
    }
    ```

Soft Skills

Diagrams are an important means of clarifying many programming concepts. You have seen them used throughout this book to illustrate such things as how the flow of control works for various programming constructs, how a program is broken into modules and those modules related, how data is stored in memory, and how data is organized.

40. Here is a set of declarations that define how the data for a set of poker hands is organized. Create a neat diagram that illustrates this organization. The diagram in Section 7.4 of Chapter 7 on nested structures might give you an idea of how to begin.

```
struct CardStruct
{  int  face;
   char suit;      // 's', 'h', 'd', or 'c'
};

struct PlayerStruct
{  int playerNum;
   CardStruct card[5];
}

PlayerStruct player[4];
```

Programming Challenges

Programming Challenges 1–7 allow you to practice working with arrays without using classes or structures. Most of the problems beginning with Programming Challenge 8 use arrays with classes or structures.

1. Perfect Scores

Write a modular program that accepts at least 10 integer test scores from the user and stores them in an array. Then `main` should display how many perfect scores were entered (i.e., scores of 100), using a value-returning `countPerfect` function to help it.

Input validation: Do not accept scores less than 0 or greater than 100.

2. Roman Numeral Converter

Write a program that displays the roman numeral equivalent of any decimal number between 1 and 20 that the user enters. The roman numerals should be stored in an array of strings and the decimal number that the user enters should be used to locate the array element holding the roman numeral equivalent. The program should have a loop that allows the user to continue entering numbers until an end sentinel of 0 is entered.

Input validation: Do not accept scores less than 0 or greater than 20.



VideoNote
Solving the Chips and Salsa Problem

3. Chips and Salsa

Write a program that lets a maker of chips and salsa keep track of their sales for five different types of salsa they produce: mild, medium, sweet, hot, and zesty. It should use two parallel five-element arrays: an array of strings that holds the five salsa names and an array of integers that holds the number of jars sold during the past month for each salsa type. The salsa names should be stored using an initialization list at the time the `name` array is created. The program should prompt the user to enter the number of jars sold for each type. Once this sales data has been entered, the program should produce a report that displays sales for each salsa type, total sales, and the names of the highest selling and lowest selling products.

Input Validation: Do not accept negative values for number of jars sold.

4. Monkey Business

A local zoo wants to keep track of how many pounds of food each of its three monkeys eats each day during a typical week. Write a program that stores this information in a two-dimensional 3 × 7 array, where each row represents a different monkey and each column represents a different day of the week. The program should first have the user input the data for each monkey. Then it should create a report that includes the following information:

- Average amount of food eaten per day by the whole family of monkeys.
- The least amount of food eaten during the week by any one monkey.
- The greatest amount of food eaten during the week by any one monkey.

Input Validation: Do not accept negative numbers for pounds of food eaten.

5. Rain or Shine

An amateur meteorologist wants to keep track of weather conditions during the past year's three month summer season and has designated each day as either rainy ('R'), cloudy ('C'), or sunny ('S'). Write a program that stores this information in a 3 × 30 array of characters, where the row indicates the month (0 = June, 1 = July, 2 = August) and the column indicates the day of the month. Note that data is not being collected for the 31st of any month. The program should begin by reading the weather data in from a file. Then it should create a report that displays for each month and for the whole three-month period, how many days were rainy, how many were cloudy, and how many were sunny. It should also report which of the three months had the largest number of rainy days. Data for the program can be found in the `RainOrShine.dat` file.

6. Lottery

Write a program that simulates a lottery. The program should have an array of 5 integers named `winningDigits`, with a randomly generated number in the range of 0 through 9 for each element in the array. The program should ask the user to enter 5 digits and should store them in a second integer array named `player`. The program must compare the corresponding elements in the two arrays and count how many digits match. For example, the following shows the `winningDigits` array and the `Player` array with sample numbers stored in each. There are two matching digits, elements 2 and 4.

`WinningDigits`	7	4	9	1	3
`player`	4	2	9	7	3

Once the user has entered a set of numbers, the program should display the winning digits and the player's digits and tell how many digits matched.

Input validation: Do not accept player inputs less than 0 or greater than 9.

7. Rainfall Statistics

Write a modular program that analyzes a year's worth of rainfall data. In addition to `main`, the program should have a `getData` function that accepts the total rainfall for each of 12 months from the user, and stores it in a `double` array. It should also have four value-returning functions that compute and return to `main` the `totalRainfall`, `averageRainfall`, `driestMonth`, and `wettestMonth`. These last two functions return the *number* of the

month with the lowest and highest rainfall amounts, not the amount of rain that fell those months. Notice that this month number can be used to obtain the amount of rain that fell those months. This information should be used either by `main` or by a `displayReport` function called by `main` to print a summary rainfall report similar to the following:

```
        2010 Rain Report for Neversnows County

Total rainfall: 23.19 inches
Average monthly rainfall: 1.93 inches
The least rain fell in January with 0.24 inches.
The most rain fell in April with 4.29 inches.
```

Input validation: Do not accept rainfall amounts less than 0.

8. `Stats` Class and Rainfall Statistics

Create a `stats` class whose member data includes an array capable of storing 30 `double` data values, and whose member functions include `total`, `average`, `lowest`, and `highest` functions for returning information about the data to the client program. These are general versions of the same functions you created for Programming Challenge 7, but now they belong to the `stats` class, not the application program. In addition to these functions, the `stats` class should have a Boolean `storeValue` function that accepts a `double` value from the client program and stores it in the array. It is the job of this function to keep track of how many values are currently in the array, so it will know where to put the next value it receives and will know how many values there are to process when it is carrying out its other functions. It is also the job of this function to make sure that no more than 30 values are accepted. If the `storeValue` function is able to successfully store the value sent to it, it should return `true` to the client program. However, if the client program tries to store a thirty-first value, the function should *not* store the value and should return `false` to the client program.

The client program should create and use a `stats` object to carry out the same rainfall analysis requested by Programming Challenge 7. Notice that the `stats` object does no I/O. All input and output is done by the client program.

9. `Stats` Class and Track Statistics

Write a client program that uses the `stats` class you created for Programming Challenge 8 to store and analyze "best" 100-yard dash times for each of the 15 runners on a track team. As in Programming Challenge 8, all I/O is done by the client program. In addition to `main`, the client program should have two other functions: a `getData` function to accept input from the user and send it to the `stats` object and a `createReport` function that creates and displays a report similar to the one shown here,

```
          Tulsa Tigers Track Team

Average 100 yard-dash time: 11.16 seconds
Slowest runner: Jack        13.09 seconds
Fastest runner: Will        10.82 seconds
```

Input validation: Do not accept times less than 0.

10. Character Converter Class

Create a `CharConverter` class that performs various operations on strings. It should have the following two public member functions:

- The `uppercase` member function accepts a string object and returns the string with all lowercase letters converted to uppercase. If a character is already uppercase, or is not a letter, it should be left alone.
- The `properWords` member function accepts a string object consisting of words separated by spaces and returns it with the first letter of each word converted to uppercase.

Write a simple program that uses the class. It should prompt the user to input a string, display it, call one or the other of the class functions to convert it, then display the converted string. The program should loop to allow additional strings to be converted and displayed until the user chooses to quit.

11. Driver's License Exam

The State Department of Motor Vehicles (DMV) has asked you to write a program that grades the written portion of the driver's license exam, which has 20 multiple choice questions. Here are the correct answers:

1. B	5. C	9. C	13. D	17. C
2. D	6. A	10. D	14. A	18. B
3. A	7. B	11. B	15. D	19. D
4. A	8. A	12. C	16. C	20. A

To do this you should create a `TestGrader` class. The class will have an `answers` array of 20 characters, which holds the correct test answers. It will have two public member functions that enable user programs to interact with the class: `setKey` and `grade`. The `setKey` function receives a 20-character string holding the correct answers, and copies this information into its `answers` array. The `grade` function receives a 20-character array holding the test taker's answers and compares each of their answers to the correct one. After "grading" the exam, the function should display a message indicating whether the applicant passed or failed the exam. An applicant must correctly answer 15 or more of the 20 questions to pass the exam. The function should then display the total number of correctly answered questions, the total number of incorrectly answered questions, and a list of the question numbers for all incorrectly answered questions.

The client program that creates and uses a `TestGrader` object should first make a single call to `setKey`, passing it a string containing the 20 correct answers. Once this is done it should allow a test taker's 20 answers to be entered, store them in a 20-character array, and then call the `grade` function to grade the exam. The program should loop to allow additional tests to be entered and graded until the user indicates a desire to quit.

Input Validation: Only accept the letters A,B,C, and D for the test answers.

12. Array of `Payroll` Objects

Design a `PayRoll` class that has data members for an employee's hourly pay rate and number of hours worked. Write a program with an array of seven `PayRoll` objects. The program should read the number of hours each employee worked and their hourly pay rate from a file and call class functions to store this information in the appropriate objects. It should then call a class function, once for each object, to return the employee's gross pay, so this information can be displayed. Sample data to test this program can be found in the `payroll.dat` file.



13. Drink Machine Simulator

Create a class that simulates and manages a soft drink machine. Information on each drink type should be stored in a structure that has data members to hold the drink name, the drink price, and the number of drinks of that type currently in the machine.

The class should have an array of five of these structures, initialized with the following data.

Drink Name	Cost	Number in Machine
Cola	.75	20
Root beer	.75	20
Orange Soda	.75	20
Grape Soda	.75	20
Bottled water	1.00	20

The class should have two public member functions, `displayChoices` (which displays a menu of drink names and prices) and `buyDrink` (which handles a sale). The class should also have at least two private member functions, `inputMoney`, which is called by `buyDrink` to accept, validate, and return (to `buyDrink`) the amount of money input, and `dailyReport`, which is called by the destructor to report how many of each drink type remain in the machine at the end of the day and how much money was collected. You may want to use additional functions to make the program more modular.

The client program that uses the class should have a main processing loop which calls the `displayChoices` class member function and allows the patron to either pick a drink or quit the program. If the patron selects a drink, the `buyDrink` class member function is called to handle the actual sale. This function should be passed the patron's drink choice. Here is what the `buyDrink` function should do:

- Call the `inputMoney` function, passing it the patron's drink choice.
- If the patron no longer wishes to make the purchase, return all input money.
- If the machine is out of the requested soda, display an appropriate "sold out" message and return all input money.
- If the machine has the soda and enough money was entered, complete the sale by updating the quantity on hand and money collected information, calculating any change due to be returned to the patron, and delivering the soda. This last action can be simulated by printing an appropriate "here is your beverage" message.

Input Validation: Only accept valid menu choices. Do not deliver a beverage if the money inserted is less than the price of the selected drink.

14. Bin Manager Class

Design and write an object-oriented program for managing inventory bins in a warehouse. To do this you will use two classes: `InvBin` and `BinManager`. The `InvBin` class holds information about a single bin. The `BinManager` class will own and manage an array of `InvBin` objects. Here is a skeleton of what the `InvBin` and `BinManager` class declarations should look like:

```
class InvBin
{
private:
   string description;                    // Item name
   int qty;                               // Quantity of items
                                          // in this bin

public:
   InvBin (string d = "empty", int q = 0)  // 2-parameter constructor
   {   description = d;  qty = q; }        // with default values

   // It will also have the following public member functions. They will
   // be used by the BinManager class, not the client program.
   void setDescription(string d)
   string getDescription()
   void setQty(int q)
   int getQty( )
};

class BinManager
{
private:
   InvBin bin[30];                        // Array of InvBin objects
   int numBins;                           // Number of bins
                                          // currently in use

public:
   BinManager()                           // Default constructor
   {   numBins = 0; }

   BinManager(int size, string d[], int q[])   // 3-parameter constructor
   {   // Receives number of bins in use and parallel arrays of item names
       // and quantities. Uses this info. to store values in the elements
       // of the bin array. Remember, these elements are InvBin objects.
   }

   // The class will also have the following public member functions:
   string getDescription(int index)     // Returns name of one item
   int getQuantity(int index)           // Returns qty of one item
   string displayAllBins()               // Returns string having one line
                                        // for each item
   bool addParts(int binIndex, int q)   // These return true if the
   bool removeParts(int binIndex, int q) // action was done and false
                                        // if it could not be done—
                                        // see validation information
};
```

Client Program

Once you have created these two classes, write a menu-driven client program that uses a `BinManager` object to manage its warehouse bins. It should initialize it to use 9 of the bins, holding the following data. The bin index where the item will be stored is also show here.

1. Regular pliers 25	2. N. nose pliers 5	3. screwdriver 25
4. P. head screw driver 6	5. Wrench-large 7	6. Wrench-small 18
7. Drill 51	8. Cordless drill 16	9. Hand saw 12

Using the `BinManager` object to help it, the modular client program should display the current bin data, then call functions to display a menu, get and validate the user's choice, and carry out the necessary activities to handle that choice. From looking at the public member functions that the `BinManager` class provides, it should be clear what options to include in the menu. When the user chooses the "Quit" option from the menu, the program should display the day's final bin information. All I/O should be done in the client class. The `BinManager` class only accepts information, keeps the array of `InvBin` objects up to date, and returns information to the client program.

Input Validation in the client class: The entered menu choice must be valid.

Input Validation in the `BinManager` class: Do not accept numbers less than 1 for the number of parts being added or removed from a bin. Do not allow the user to remove more items from a bin than it currently holds.

Group Projects

15. Tic-Tac-Toe Game

Write a modular program that allows two players to play a game of tic-tac-toe. Use a two-dimensional `char` array with 3 rows and 3 columns as the game board. Each element of the array should be initialized with an asterisk (*). The program should display the initial board configuration and then start a loop that does the following:

- Allow player 1 to select a location on the board for an X by entering a row and column number. Then redisplay the board with an X replacing the * in the chosen location.
- If there is no winner yet and the board is not yet full, allow player 2 to select a location on the board for an O by entering a row and column number. Then redisplay the board with an O replacing the * in the chosen location.

The loop should continue until a player has won or a tie has occurred, then display a message indicating who won, or reporting that a tie occurred.

- Player 1 wins when there are three Xs in a row, a column, or a diagonal on the game board.
- Player 2 wins when there are three Os in a row, a column, or a diagonal on the game board.
- A tie occurs when all of the locations on the board are full, but there is no winner.

Input Validation: Only allow legal moves to be entered. The row must be 1, 2, or 3. The column must be 1, 2, or 3. The (row, column) position entered must currently be empty (i.e., still have an asterisk in it).

16. Theater Ticket Sales

Create a `TicketManager` class and a program that uses it to sell tickets for a single performance theater production. This project is intended to be designed and written by a team of 2-4 students. Here are some suggestions:

- One student might design and write the client program that uses the class. The `TicketManager` class and all of its functions can be designed and written by other team members.
- The program requirements should be analyzed and each student given about the same work load.
- The class design and the names, parameters, and return types of each function should be decided in advance.
- The project can be implemented either as a multi-file program, or all the functions can be cut and pasted into a single file.

Here are the specifications:

- The theater's auditorium has 15 rows, with 30 seats in each row. To represent the seats, the `TicketManager` class should have a two-dimensional array of `SeatStructures`. Each of these structures should have data members to keep track of the seat's price and whether or not it is available or already sold.
- The seat prices should be read in from the `SeatPrices.dat` file. This file contains 15 pieces of data representing the price for each row. All seats in a given row are the same price, but different rows have different prices. The seat availability information should be read in from the `SeatAvailability.dat` file. This file contains 450 characters (15 rows with 30 characters each), indicating which seats have been sold (`'*'`) and which are available (`'#'`). Initially all seats are available. However, once the program runs and the file is updated, some of the seats will have been sold. The obvious function to read in the data from these files and set up the array is the constructor that runs when a `TicketManager` object is created at the beginning of the program.
- The client program should be a menu-driven program that provides the user with a menu of box office options, accepts and validates user inputs, and calls appropriate class functions to carry out desired tasks. The menu should include options to display the seating chart, to request tickets, to print a theater sales report, and to exit the program.
- When the user selects the *display seats* menu option, a `TicketManager` function should be called that displays a chart, similar to the one shown on the following page. It should indicate which seats are already sold (`*`) and which are still available for purchase (`#`).

```
                    Seats
          123456789012345678901234567890
Row 1     ##############################
Row 2     ##############################
Row 3     ##############################
Row 4     ##############################
Row 5     ##############################
Row 6     ##############################
Row 7     ##############################
Row 8     ##############################
Row 9     ##############################
Row 10    ##############################
Row 11    ##############################
Row 12    ##############################
Row 13    ##############################
Row 14    ##############################
Row 15    ##############################
```

- When the user selects the *request tickets* menu option, the program should prompt for the number of seats the patron wants, the desired row number, and the desired starting seat number. A `TicketManager` ticket request function should then be called and passed this information so that it can handle the ticket request. If any of the requested seats do not exist, or are not available, an appropriate message should be displayed. If the seats exist and are available, a bill should be printed that lists the number of requested seats, the price per seat in the requested row, and the total price for the seats. Then the user program should ask if the patron wishes to purchase these seats.
- If the patron indicates they do want to buy the requested seats, a `TicketManager` purchase tickets module should be called to handle the actual sale. This module must be able to accept money, ensure that it is sufficient to continue with the sale, print a ticket for each seat sold (with the correct row, seat number, and price on it), mark the seat as sold, and give the patron any change due.
- When the user selects the *sales report* menu option, a `TicketManager` report module should be called. This module must display a report that tells how many seats have been sold, how many are still available, and how much money has been collected so far for the sold seats. Think about how your team will either calculate or collect and store this information so that it will be available when it is needed for the report.
- When the day of ticket sales is over and the *quit* menu choice is selected, the program needs to be able to write the updated seat availability data back out to the file. The obvious place to do this is in the `TicketManager` destructor.

This page intentionally left blank

CHAPTER 9

Searching, Sorting, and Algorithm Analysis

TOPICS

9.1 Introduction to Search Algorithms
9.2 Searching an Array of Objects
9.3 Introduction to Sorting Algorithms
9.4 Sorting an Array of Objects
9.5 Sorting and Searching Vectors
9.6 Introduction to Analysis of Algorithms
9.7 Case Studies
9.8 Tying It All Together: *Secret Messages*

9.1 Introduction to Search Algorithms

CONCEPT: A search algorithm is a method of locating a specific item in a collection of data.

It's very common for programs not only to store and process data stored in arrays, but to search arrays for specific items. This section will show you two methods of searching an array: the linear search and the binary search. Each has its advantages and disadvantages.

The Linear Search

The *linear search* is a very simple algorithm. Sometimes called a *sequential search*, it uses a loop to sequentially step through an array, starting with the first element. It compares each element with the value being searched for, and stops when either the value is found or the end of the array is encountered. If the value being searched for is not in the array, the algorithm will search to the end of the array.

Here is the pseudocode for a function that performs the linear search:

```
Set found to false
Set position to -1
Set index to 0
While index < number of elements and found is false
   If list[index] is equal to search value
      found = true
      position = index
   End If
   Add 1 to index
End While
Return position
```

The function `searchList`, which follows, is an example of C++ code used to perform a linear search on an integer array. The array `list`, which has a maximum of `size` elements, is searched for an occurrence of the number stored in `value`. If the number is found, its array subscript is returned. Otherwise, –1 is returned, indicating the value did not appear in the array.

```
int searchList(int list[], int size, int value)
{
   int index = 0;          // Used as a subscript to search array
   int position = -1;      // Used to record position of search value
   bool found = false;     // Flag to indicate if the value was found

   while (index < size && !found)
   {
      if (list[index] == value)     // If the value is found
      {
          found = true;             // Set the flag
          position = index;         // Record the value's subscript
      }
      index++;                      // Go to the next element
   }
   return position;                 // Return the position, or -1
}
```



NOTE: The reason –1 is chosen to indicate that the search value was not found in the array is that –1 is not a valid subscript. Any other nonvalid subscript value could also have been used to signal this.

Program 9-1 is a complete program that uses the `searchList` function. It searches the five-element `tests` array to find a score of 100.

Program 9-1

```
1 // This program demonstrates the searchList function,
2 // which performs a linear search on an integer array.
3 #include <iostream>
4 using namespace std;
5
```

(program continues)

Program 9-1 *(continued)*

```
// Function prototype
int searchList(int [], int, int);

const int SIZE = 5;

int main()
{
   int tests[SIZE] = {87, 75, 98, 100, 82};
   int results;         // Holds the search results

   // Search the array for the value 100
   results = searchList(tests, SIZE, 100);

   // If searchList returned -1, 100 was not found
   if (results == -1)
      cout << "You did not earn 100 points on any test.\n";
   else
   {  // Otherwise results contains the subscript of
      // the first 100 found in the array
      cout << "You earned 100 points on test ";
      cout << (results + 1) << ".\n";
   }
   return 0;
}

/**************************************************************
 *                          searchList                        *
 * This function performs a linear search on an integer array.*
 * The list array, which has size elements, is searched for   *
 * the number stored in value. If the number is found, its array *
 * subscript is returned. Otherwise, -1 is returned.          *
 **************************************************************/
int searchList(int list[], int size, int value)
{
   int index = 0;          // Used as a subscript to search array
   int position = -1;      // Used to record position of search value
   bool found = false;     // Flag to indicate if the value was found

   while (index < size && !found)
   {
      if (list[index] == value)  // If the value is found
      {
         found = true;           // Set the flag
         position = index;       // Record the value's subscript
      }
      index++;                   // Go to the next element
   }
   return position;              // Return the position, or -1
}
```

Program Output

```
You earned 100 points on test 4.
```

Inefficiency of the Linear Search

The advantage of the linear search is its simplicity. It is very easy to understand and implement. Furthermore, it doesn't require the data in the array to be stored in any particular order. Its disadvantage, however, is its inefficiency. If the array being searched contained 20,000 elements, the algorithm would have to look at all 20,000 elements in order to find a value stored in the last element or to determine that a desired element was not in the array.

In a typical case, an item is just as likely to be found near the beginning of the array as near the end. On average, for an array of N items, the linear search will locate an item in N/2 attempts. If an array has 20,000 elements, the linear search will make a comparison with 10,000 of them on average. This is assuming, of course, that the search item is consistently found in the array. (N/2 is the average number of comparisons. The maximum number of comparisons is always N.)

When the linear search fails to locate an item, it must make a comparison with every element in the array. As the number of failed search attempts increases, so does the average number of comparisons. When it can be avoided the linear search should not be used on large arrays if speed is important.

The Binary Search



VideoNote
Performing a Binary Search

The *binary search* is a clever algorithm that is much more efficient than the linear search. Its only requirement is that the values in the array be in order. Instead of testing the array's first element, this algorithm starts with the element in the middle. If that element happens to contain the desired value, then the search is over. Otherwise, the value in the middle element is either greater than or less than the value being searched for. If it is greater than the desired value then the value (if it is in the list) will be found somewhere in the first half of the array. If it is less than the desired value then the value (again, if it is in the list) will be found somewhere in the last half of the array. In either case, half of the array's elements have been eliminated from further searching.

If the desired value wasn't found in the middle element, the procedure is repeated for the half of the array that potentially contains the value. For instance, if the last half of the array is to be searched, the algorithm immediately tests *its* middle element. If the desired value isn't found there, the search is narrowed to the quarter of the array that resides before or after that element. This process continues until the value being searched for is either found or there are no more elements to test.

Here is the pseudocode for a function that performs a binary search on an array whose elements are stored in ascending order.

```
Set first to 0
Set last to the last subscript in the array
Set found to false
Set position to -1
```

```
While found is not true and first is less than or equal to last
   Set middle to the subscript halfway between first and last
   If array[middle] equals the desired value
       Set found to true
       Set position to middle
   Else If array[middle] is greater than the desired value
       Set last to middle - 1
   Else
       Set first to middle + 1
   End If
End While
Return position
```

This algorithm uses three index variables: `first`, `last`, and `middle`. The `first` and `last` variables mark the boundaries of the portion of the array currently being searched. They are initialized with the subscripts of the array's first and last elements. The subscript of the element approximately halfway between `first` and `last` is calculated and stored in the `middle` variable. If there is no precisely central element, the integer division used to calculate `middle` will select the element immediately preceding the midpoint. If the element in the middle of the array does not contain the search value, the `first` or `last` variables are adjusted so that only the top or bottom half of the array is searched during the next iteration. This cuts the portion of the array being searched in half each time the loop fails to locate the search value.

The function `binarySearch` in the following example C++ code is used to perform a binary search on an integer array. The first parameter, `array`, which has `size` elements, is searched for an occurrence of the number stored in `value`. If the number is found, its array subscript is returned. Otherwise, –1 is returned indicating the value did not appear in the array.

```
int binarySearch(int array[], int size, int value)
{
   int  first = 0,                    // First array element
        last = size - 1,              // Last array element
        middle,                       // Midpoint of search
        position = -1;                // Position of search value
   bool found = false;                // Flag

   while (!found && first <= last)
   {
      middle = (first + last) / 2;    // Calculate midpoint
      if (array[middle] == value)     // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value)  // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1;          // If value is in upper half
   }
   return position;
}
```

Program 9-2 is a complete program using the `binarySearch` function. It searches an array of employee ID numbers for a specific value.

Program 9-2

```
// This program performs a binary search on an integer
// array whose elements are in ascending order.
#include <iostream>
using namespace std;

// Function prototype
int binarySearch(int [], int, int);

const int SIZE = 20;

int main()
{
   // Create an array of ID numbers sorted in ascending order
   int IDnums[SIZE] = {101, 142, 147, 189, 199, 207, 222,
                       234, 289, 296, 310, 319, 388, 394,
                       417, 429, 447, 521, 536, 600 };

   int empID,          // Holds the ID to search for
       results;        // Holds the search results

   // Get an employee ID to search for
   cout << "Enter the employee ID you wish to search for: ";
   cin  >> empID;

   // Search for the ID
   results = binarySearch(IDnums, SIZE, empID);

   // If binarySearch returned -1, the ID was not found
   if (results == -1)
      cout << "That number does not exist in the array.\n";
   else
   {  // Otherwise results contains the subscript of
      // the specified employee ID in the array
      cout << "ID " << empID << " was found in element "
           << results << " of the array.\n";
   }
   return 0;
}

```

(program continues)

Program 9-2 *(continued)*

```
40 /*************************************************************
41  *                       binarySearch                        *
42  * This function performs a binary search on an integer array  *
43  * with size elements whose values are stored in ascending     *
44  * order. The array is searched for the number stored in the   *
45  * value parameter. If the number is found, its array subscript *
46  * is returned. Otherwise, -1 is returned.                     *
47  *************************************************************/
48 int binarySearch(int array[], int size, int value)
49 {
50    int  first = 0,                    // First array element
51         last = size - 1,              // Last array element
52         middle,                       // Midpoint of search
53         position = -1;                // Position of search value
54    bool found = false;                // Flag
55
56    while (!found && first <= last)
57    {
58       middle = (first + last) / 2;    // Calculate midpoint
59       if (array[middle] == value)     // If value is found at mid
60       {
61          found = true;
62          position = middle;
63       }
64       else if (array[middle] > value) // If value is in lower half
65          last = middle - 1;
66       else
67          first = middle + 1;          // If value is in upper half
68    }
69    return position;
70 }
```

Program Output with Example Input Shown in Bold

```
Enter the employee ID you wish to search for: 199[Enter]
ID 199 was found in element 4 of the array.
```

The Efficiency of the Binary Search

Obviously, the binary search is much more efficient than the linear search. Every time it makes a comparison and fails to find the desired item, it eliminates half of the remaining portion of the array that must be searched. For example, consider an array with 20,000 elements. If the binary search fails to find an item on the first attempt, the number of elements that remains to be searched is 10,000. If the item is not found on the second attempt, the number of elements that remains to be searched is 5,000. This process continues until the binary search locates the desired value or determines that it is not in the array. With 20,000 elements in the array, this takes a maximum of 15 comparisons. (Compare this to the linear search, which would make an average of 10,000 comparisons!)

Powers of 2 are used to calculate the maximum number of comparisons the binary search will make on an array of any size. (A power of 2 is 2 raised to some integer exponent.)

Simply find the smallest power of 2 that is greater than the number of elements in the array. That will tell you the maximum number of comparisons needed to find an element, or to determine that it is not present. For example, a maximum of 16 comparisons will be made to find an item in an array of 50,000 elements ($2^{16} = 65{,}536$), and a maximum of 20 comparisons will be made to find an item in an array of 1,000,000 elements ($2^{20} = 1{,}048{,}576$).

9.2 Searching an Array of Objects

CONCEPT: Linear and binary searches can also be used to search for a specific entry in an array of objects or structures.

In Programs 9-1 and 9-2 we searched for a particular value in an array of integers. We could just as easily search through an array holding values of some other data type, such as `double` or `string`. We could even search an array of objects or structures. In this case, however, the search value would not be the entire object or structure we are looking for, but rather a value in a particular member variable of that object or structure. The member variable being examined by the search is sometimes called the *key field* and the particular value being looked for is called the *search key*.

Assume we have a class named `Inventory` that includes the following member variables

```
string itemCode;
string description;
double price;
```

as well as methods to *set* and *get* the value of each of these. Assume also that we have set up an array of `Inventory` objects. We might want to search for a particular object in the array, say the object whose `itemCode` is K33, so that we can then call the `getPrice` method for that object. Program 9-3 illustrates how to do this. It searches the array of `Inventory` objects using a `search` function similar to the `searchList` function we used earlier in this chapter. However, it has been modified to work with an array of `Inventory` objects.

Program 9-3

```
1  // This program searches an array of Inventory objects to get
2  // the price of a particular object. It demonstrates how to
3  // perform a linear search using an array of objects.
4  #include <iostream>
5  #include <string>
6  using namespace std;
7
8  // Inventory class declaration
9  class Inventory
10 {   private:
11       string itemCode;
12       string description;
13       double price;
14
```

(program continues)

Program 9-3 *(continued)*

```
15   public:
16      Inventory()                              // Default constructor
17      {  itemCode = "XXX";  description = " ";  price = 0.0; }
18
19      Inventory(string c, string d, double p) // 3 argument constructor
20      {  itemCode = c;
21         description = d;
22         price = p;
23      }
24
25      // Add methods setCode, setDescription, and setPrice here.
26
27      // Get functions to retrieve member variable values
28      string getCode()
29      {  string code = itemCode;
30         return code;
31      }
32
33      string getDescription()
34      {  string d = description;
35         return d;
36      }
37
38      double getPrice()
39      {  return price;
40      }
41
42 }; // End Inventory class declaration
43
44 // Program that uses the Inventory class
45
46 // Function prototype
47 int search(Inventory[], int , string);
48
49 /*****************************************************
50  *                      main                         *
51  *****************************************************/
52 int main()
53 {
54    const int SIZE = 6;
55
56    // Create and initialize the array of Inventory objects
57    Inventory silverware[SIZE] =
58                              { Inventory("S15", "soup spoon",  2.35),
59                                Inventory("S12", "teaspoon",    2.19),
60                                Inventory("F15", "dinner fork", 3.19),
61                                Inventory("F09", "salad fork" , 2.25),
62                                Inventory("K33", "knife",       2.35),
63                                Inventory("K41", "steak knife", 4.15) };
64
65    string desiredCode;     // The itemCode to search for
66    int pos;                // Position of desired object in the array
67    char doAgain;           // Look up another price (Y/N)?
68
```

(program continues)

Program 9-3 *(continued)*

```
69    do
70    {   // Get the itemCode to search for
71        cout << "\nEnter an item code: ";
72        cin  >> desiredCode;
73
74        // Search for the object
75        pos = search(silverware, SIZE, desiredCode);
76
77        // If pos = -1, the code was not found
78        if (pos == -1)
79           cout << "That code does not exist in the array\n";
80        else
81        {  // The object was found, so use pos to get the
82           // description and price
83           cout << "This "    << silverware[pos].getDescription()
84                << " costs $" << silverware[pos].getPrice() << endl;
85        }
86
87        // Does the user want to look up another price?
88        cout << "\nLook up another price (Y/N)? ";
89        cin  >> doAgain;
90
91    } while (doAgain == 'Y' || doAgain == 'y');
92    return 0;
93 }// End main
94
95 /*************************************************************
96  *                          search                           *
97  * This function performs a linear search on an array of     *
98  * Inventory objects, using itemCode as the key field.       *
99  * If the desired code is found, its array subscript is      *
100 * returned. Otherwise, -1 is returned.                      *
101 *************************************************************/
102 int search(Inventory object[], int size, string value)
103 {
104    int index = 0;          // Used as a subscript to search array
105    int position = -1;      // Used to record position of search value
106    bool found = false;     // Flag to indicate if the value was found
107
108    while (index < size && !found)
109    {
110       if (object[index].getCode() == value) // If the value is found
111       {
112          found = true;          // Set the flag
113          position = index;      // Record the value's subscript
114       }
115       index++;                  // Go to the next element
116    }
117    return position;             // Return the position, or -1
118 }// End search
```

(program continues)

Program 9-3 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter an item code: F15[Enter]
This dinner fork costs $3.19

Look up another price (Y/N)? n[Enter]
```



Checkpoint

9.1 Describe the difference between the linear search and the binary search.

9.2 On average, with an array of 20,000 elements, how many comparisons will the linear search perform? (Assume the items being search for are consistently found in the array.)

9.3 With an array of 20,000 elements, what is the maximum number of comparisons the binary search will perform?

9.4 If a linear search is performed on an array, and it is known that some items are searched for more frequently than others, how can the contents of the array be reordered to improve the average performance of the search?

9.3 Introduction to Sorting Algorithms

CONCEPT: Sorting algorithms are used to arrange data into some order.



VideoNote
Sorting a Set of Data

Often the data in an array must be sorted in some order. Customer lists, for instance, are commonly sorted in alphabetical order. Student grades might be sorted from highest to lowest. Mailing label records could be sorted by ZIP code. To sort the data in an array, the programmer must use an appropriate *sorting algorithm*. A sorting algorithm is a technique for scanning through an array and rearranging its contents in some specific order. This section will introduce two simple sorting algorithms: the *bubble sort* and the *selection sort*.

The Bubble Sort

The bubble sort is an easy way to arrange data in *ascending* or *descending order*. If an array is sorted in ascending order, it means the values in the array are stored from lowest to highest. If the values are sorted in descending order, they are stored from highest to lowest. Bubble sort works by comparing each element with its neighbor and swapping them if they are not in the desired order. Let's see how it arranges the following array's elements in ascending order:

7	2	3	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The bubble sort starts by comparing the first two elements in the array. If element 0 is greater than element 1, they are exchanged. After the exchange, the array appears as

2	7	3	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

This process is repeated with elements 1 and 2. If element 1 is greater than element 2, they are exchanged. The array now appears as

2	3	7	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Next, elements 2 and 3 are compared. However, in this array, these two elements are already in the proper order (element 2 is less than element 3), so no exchange takes place.

As the cycle continues, elements 3 and 4 are compared. Once again, no exchange is necessary because they are already in the proper order. When elements 4 and 5 are compared, however, an exchange must take place because element 4 is greater than element 5. The array now appears as

2	3	7	8	1	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

At this point, the entire array has been scanned. This is called the first *pass* of the sort. Notice that the largest value is now correctly placed in the last array element. However, the rest of the array is not yet sorted. So the sort starts over again with elements 0 and 1. Because they are in the proper order, no exchange takes place. Elements 1 and 2 are compared next, but once again, no exchange takes place. This continues until elements 3 and 4 are compared. Because element 3 is greater than element 4, they are exchanged. The array now appears as

2	3	7	1	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Notice that this second pass over the array elements has placed the second largest number in the next to the last array element. This process will continue, with the sort repeatedly passing through the array and placing at least one number in order on each pass, until the array is fully sorted. Ultimately, the array will appear as

1	2	3	7	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Here is the bubble sort in pseudocode. Notice that it uses a pair of nested loops. The outer loop, a `do-while` loop, iterates once for each pass of the sort. The inner loop, a `for` loop, holds the code that does all the comparisons and needed swaps during a pass. If two elements are exchanged, the `swap` flag variable is set to `true`. The outer loop continues iterating, causing additional passes to be made, until it finds the `swap` flag `false`, meaning that no elements were swapped on the previous pass. This indicates that the array is now fully sorted.

```
Do
   Set swap flag to false
   For count = 0 to the next-to-last array subscript
      If array[count] is greater than array[count + 1]
         Swap the contents of array[count] and array[count + 1]
         Set swap flag to true
      End If
   End For
While the swap flag is true   // a swap ocurred on the previous pass
```

The following C++ code implements the bubble sort as a function. The parameter `array` references an integer array to be sorted. The parameter `size` contains the number of elements in `array`.

```
void sortArray(int array[], int size)
{
   int temp;
   bool swap;

   do
   {  swap = false;
      for (int count = 0; count < (size - 1); count++)
      {
         if (array[count] > array[count + 1])
         {
            temp = array[count];
            array[count] = array[count + 1];
            array[count + 1] = temp;
            swap = true;
         }
      }
   } while (swap);
}
```

Let's look more closely at the `for` loop that handles the comparisons and exchanges during a pass. Here is its starting line:

```
for (int count = 0; count < (size - 1); count++)
```

The variable `count` holds the array subscripts. It starts at zero and is incremented as long as it is less than `size - 1`. The value of `size` is the number of elements in the array, and `count` stops just short of reaching this value because the following line compares each element with the one after it:

```
if (array[count] > array[count + 1])
```

When `array[count]` is the next-to-last element, it will be compared to the last element. If the `for` loop were allowed to increment `count` past `size - 1`, the last element in the array would be compared to a value outside the array.

Here is the `if` statement in its entirety:

```
if (array[count] > array[count + 1])
{
   temp = array[count];
   array[count] = array[count + 1];
   array[count + 1] = temp;
   swap = true;
}
```

If `array[count]` is greater than `array[count + 1]`, the two elements must be exchanged. First, the contents of `array[count]` is copied into the variable `temp`. Then the contents of `array[count + 1]` is copied into `array[count]`. The exchange is made complete when `temp` (which holds the previous contents of `array[count]`) is copied to `array[count + 1]`. Last, the `swap` flag variable is set to `true`. This indicates that an exchange has been made.

Program 9-4 demonstrates the bubble sort function in a complete program.

Program 9-4

```
1  // This program uses the bubble sort algorithm to sort an array
2  // of integers in ascending order.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototypes
7  void sortArray(int [], int);
8  void showArray(int [], int);
9
10 int main()
11 {
12    const int SIZE = 6;
13
14    // Array of unsorted values
15    int values[SIZE] = {7, 2, 3, 8, 9, 1};
16
17    // Display the values
18    cout << "The unsorted values are:\n";
19    showArray(values, SIZE);
20
21    // Sort the values
22    sortArray(values, SIZE);
23
24    // Display them again
25    cout << "The sorted values are:\n";
26    showArray(values, SIZE);
27    return 0;
28 }
29
```

(program continues)

Program 9-4 *(continued)*

```
30 /**********************************************************
31  *                        sortArray                       *
32  * This function performs an ascending-order bubble sort on *
33  * array. The parameter size holds the number of elements   *
34  * in the array.                                            *
35  **********************************************************/
36 void sortArray(int array[], int size)
37 {
38    int  temp;
39    bool swap;
40
41    do
42    {   swap = false;
43       for (int count = 0; count < (size - 1); count++)
44       {
45          if (array[count] > array[count + 1])
46          {
47             temp = array[count];
48             array[count] = array[count + 1];
49             array[count + 1] = temp;
50             swap = true;
51          }
52       }
53    } while (swap);
54 }
55
56 /**********************************************************
57  *                        showArray                       *
58  * This function displays the contents of array. The        *
59  * parameter size holds the number of elements in the array. *
60  **********************************************************/
61 void showArray(int array[], int size)
62 {
63    for (int count = 0; count < size; count++)
64       cout << array[count] << " ";
65    cout << endl;
66 }
```

Program Output

```
The unsorted values are:
7 2 3 8 9 1
The sorted values are:
1 2 3 7 8 9
```

The Selection Sort

The bubble sort is inefficient for large arrays because repeated data swaps are often required to place a single item in its correct position. The selection sort, like the bubble sort, places just one item in its correct position on each pass. However, it usually performs fewer exchanges because it moves items immediately to their correct position in the array. Like any sort, it can be modified to sort in either ascending or descending order. An ascending

sort works like this: The smallest value in the array is located and moved to element 0. Then the next smallest value is located and moved to element 1. This process continues until all of the elements have been placed in their proper order.

Let's see how the selection sort works when arranging the elements of the following array:

5	7	2	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The selection sort scans the array, starting at element 0, and locates the element with the smallest value. The contents of this element are then swapped with the contents of element 0. In this example, the 1 stored in element 5 is the smallest value, so it is swapped with the 5 stored in element 0. This completes the first pass and the array now appears as

1	7	2	8	9	5
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The algorithm then repeats the process, but because element 0 already contains the smallest value in the array, it can be left out of the procedure. For the second pass, the algorithm begins the scan at element 1. It locates the smallest value in the unsorted part of the array, which is the 2 in element 2. Therefore, element 2 is exchanged with element 1. The array now appears as

1	2	7	8	9	5
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Once again the process is repeated, but this time the scan begins at element 2. The algorithm will find that element 5 contains the next smallest value and will exchange this element's contents with that of element 2, causing the array to appear as

1	2	5	8	9	7
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Next, the scanning begins at element 3. Its contents is exchanged with that of element 5, causing the array to appear as

1	2	5	7	9	8
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

At this point there are only two elements left to sort. The algorithm finds that the value in element 5 is smaller than that of element 4, so the two are swapped. This puts the array in its final arrangement:

1	2	5	7	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Here is the selection sort algorithm in pseudocode:

```
For startScan = 0 to the next-to-last array subscript
   Set index to startScan
   Set minIndex to startScan
   Set minValue to array[startScan]
   For index = (startScan + 1) to the last subscript in the array
      If array[index] is less than minValue
         Set minValue to array[index]
         Set minIndex to index
      End If
      Increment index
   End For
   Set array[minIndex] to array[startScan]
   Set array[startScan] to minValue
End For
```

The following C++ code implements the selection sort in a function. It accepts two arguments: `array` and `size`. The parameter `array` is an integer array and `size` is the number of elements in the array. The function uses the selection sort to arrange the values in the array in ascending order.

```
void selectionSort(int array[], int size)
{
    int startScan, minIndex, minValue;

    for (startScan = 0; startScan < (size - 1); startScan++)
    {
       minIndex = startScan;
       minValue = array[startScan];
       for (int index = startScan + 1; index < size; index++)
       {
          if (array[index] < minValue)
          {
             minValue = array[index];
             minIndex = index;
          }
       }
       array[minIndex] = array[startScan];
       array[startScan] = minValue;
    }
}
```

As with bubble sort, selection sort uses a pair of nested loops, in this case two `for` loops. The inner loop sequences through the array, starting at `array[startScan + 1]`, searching for the element with the smallest value. When the element is found, its subscript is stored in the variable `minIndex` and its value is stored in `minValue`. The outer loop then exchanges the contents of this element with `array[startScan]` and increments `startScan`. This procedure repeats until the contents of every element have been moved to their proper location. For N pieces of data this requires N-1 passes.

Program 9-5 demonstrates the selection sort function in a complete program.

Program 9-5

```
// This program uses the selection sort algorithm to sort
// an array in ascending order.
#include <iostream>
using namespace std;

// Function prototypes
void selectionSort(int [], int);
void showArray(int [], int);

int main()
{
   const int SIZE = 6;

   // Array of unsorted values
   int values[SIZE] = {5, 7, 2, 8, 9, 1};

   // Display the values
   cout << "The unsorted values are\n";
   showArray(values, SIZE);

   // Sort the array
   selectionSort(values, SIZE);

   // Display the values again
   cout << "The sorted values are\n";
   showArray(values, SIZE);
   return 0;
}

/**********************************************************
 *                        selectionSort                   *
 * This function performs an ascending-order selection sort   *
 * on array. The parameter size holds the number of elements  *
 * in the array.                                              *
 **********************************************************/
void selectionSort(int array[], int size)
{
   int startScan, minIndex, minValue;

```

(program continues)

Program 9-5 *(continued)*

```
40    for (startScan = 0; startScan < (size - 1); startScan++)
41    {
42       minIndex = startScan;
43       minValue = array[startScan];
44       for(int index = startScan + 1; index < size; index++)
45       {
46          if (array[index] < minValue)
47          {
48             minValue = array[index];
49             minIndex = index;
50          }
51       }
52       array[minIndex] = array[startScan];
53       array[startScan] = minValue;
54    }
55 }
56
57 /**********************************************************
58  *                        showArray                       *
59  * This function displays the contents of array. The      *
60  * parameter size holds the number of elements in the array. *
61  **********************************************************/
62 void showArray(int array[], int size)
63 {
64    for (int count = 0; count < size; count++)
65       cout << array[count] << " ";
66    cout << endl;
67 }
```

Program Output

```
The unsorted values are
5 7 2 8 9 1
The sorted values are
1 2 5 7 8 9
```



Checkpoint

9.5 True or false: Any sort can be modified to sort in either ascending or descending order.

9.6 What one line of code would need to be modified in the bubble sort to make it sort in descending, rather than ascending order? How would the revised line be written?

9.7 After one pass of bubble sort, which value is in order?

9.8 After one pass of selection sort, which value is in order?

9.9 Which sort usually requires fewer data values to be swapped, bubble sort or selection sort?

9.4 Sorting an Array of Objects

CONCEPT: **Sorting algorithms can also be used to order elements in an array of objects or structures.**

Programs 9-4 and 9-5 illustrated how to sort an array of integers using bubble sort and selection sort. These sorts could just as easily be used to sort array elements of any other data type. Program 9-6 uses a bubble sort to sort `Inventory` objects, using the `Inventory` class introduced earlier in this chapter. When sorting objects or structures, one must decide which data item to sort on. For example, we could arrange `Inventory` objects in order by `itemCode`, by `description`, or by `price`. To determine if two elements are out of order and should be swapped, we compare *only* the values in the data member we are sorting on. However, if the two array elements are found to be out of order, we swap the *entire* two elements. This is illustrated in Program 9-6.

Program 9-6

```
1  // This program uses bubble sort to sort an array of objects.
2  // It places Inventory objects in ascending order by their itemCode.
3  #include <iostream>
4  #include <iomanip>
5  #include <string>
6  using namespace std;
7
8  // Inventory class declaration
9  class Inventory
10 {   private:
11       string itemCode;
12       string description;
13       double price;
14
15   public:
16      Inventory()                              // Default constructor
17      {  itemCode = "XXX";  description = " ";  price = 0.0; }
18
19      Inventory(string c, string d, double p) // 3 argument constructor
20      {  itemCode = c;
21         description = d;
22         price = p;
23      }
24
25      // Add methods setCode, setDescription, and setPrice here.
26
27      // Get functions to retrieve member variable values
28      string getCode()
29      {  string code = itemCode;
30         return code;
31      }
32
```

(program continues)

Program 9-6 *(continued)*

```
33      string getDescription()
34      {  string d = description;
35         return d;
36      }
37
38      double getPrice()
39      {  return price;
40      }
41
42 }; // End Inventory class declaration
43
44 // Program that uses the Inventory class
45
46 // Function prototype
47 void displayInventory(Inventory[], int);
48 void bubbleSort(Inventory[], int);
49
50 /*****************************************************
51  *                      main                         *
52  *****************************************************/
53 int main()
54 {
55    const int SIZE = 6;
56
57    // Create and initialize the array of Inventory objects
58    Inventory silverware[SIZE] =
59                              { Inventory("S15", "soup spoon",  2.35),
60                                Inventory("S12", "teaspoon",    2.19),
61                                Inventory("F15", "dinner fork", 3.19),
62                                Inventory("F09", "salad fork" , 2.25),
63                                Inventory("K33", "knife",       2.35),
64                                Inventory("K41", "steak knife", 4.15) };
65
66    // Display the inventory
67    cout << "Here is the original data\n";
68    displayInventory(silverware, SIZE);
69
70    // Sort the objects by their itemCode
71    bubbleSort(silverware, SIZE);
72
73    // Display the inventory again
74    cout << "\nHere is the sorted data\n";
75    displayInventory(silverware, SIZE);
76
77    return 0;
78 } //End main
79
```

(program continues)

Program 9-6 *(continued)*

```
/*******************************************************
 *               displayInventory                      *
 * This function displays the entire array.            *
 *******************************************************/
void displayInventory(Inventory object[], int size)
{
   for (int index = 0; index < size; index++)
   {  cout << setw(5)  << left  << object[index].getCode()
           << setw(13) << left  << object[index].getDescription()
           << "$"      << right << object[index].getPrice() << endl;
   }
}// End displayInventory

/*******************************************************
 *                    bubbleSort                       *
 * This function performs a bubble sort on Inventory   *
 * objects, arranging them in ascending itemCode order. *
 *******************************************************/
void bubbleSort(Inventory array[], int size)
{
   Inventory temp;    // Holds an Inventory object
   bool swap;

   do
   {  swap = false;
      for (int count = 0; count < (size - 1); count++)
      {
         if (array[count].getCode() > array[count + 1].getCode())
         {
            temp = array[count];
            array[count] = array[count + 1];
            array[count + 1] = temp;
            swap = true;
         }
      }
   } while (swap);
}// End bubbleSort
```

Program Output

```
Here is the original data
S15  soup spoon   $2.35
S12  teaspoon     $2.19
F15  dinner fork  $3.19
F09  salad fork   $2.25
K33  knife        $2.35
K41  steak knife  $4.15

Here is the sorted data
F09  salad fork   $2.25
F15  dinner fork  $3.19
K33  knife        $2.35
K41  steak knife  $4.15
S12  teaspoon     $2.19
S15  soup spoon   $2.35
```

Let's take a closer look at the `bubbleSort` function. Line 107 contains the code to compare the objects stored in two array elements. Notice that only the `itemCode` values of the objects are compared and that these values are retrieved by using each object's `getCode` method.

Next look at lines 109 through 111 where the actual swap takes place when two objects are out of order. Notice that the entire objects are swapped and that an entire object can be moved in a single statement. It is not necessary to move the member variables one by one.

Finally, notice in line 100 that `temp` is defined as an `Inventory` object. Because it will be used to temporarily hold an array element during each swap, and because the array elements in this case are `Inventory` objects, `temp` must also be defined as an `Inventory` object.

9.5 Sorting and Searching Vectors

CONCEPT: The sorting and searching algorithms you have studied in this chapter can be applied to STL vectors as well as to arrays.

In the previous chapter you learned about the vector class that is part of the Standard Template Library (STL). Once you have properly defined an STL `vector` and populated it with values, you may sort and search the vector with the algorithms presented in this chapter. Simply substitute the vector syntax for the array syntax when necessary.

Program 9-7 modifies program 9-4 to use a STL `vector` instead of an array.

Program 9-7

```
1  // This program uses the bubble sort algorithm to sort
2  // a vector of integers in ascending order.
3  #include <iostream>
4  #include <vector>             // Needed to use vectors
5  using namespace std;
6
7  // Function prototypes
8  void displayVector(vector<int>);
9  void sortVector(vector<int> &);
10
11 int main()
12 {
13    const int SIZE = 6;
14
15    // Create a vector to hold a set of unsorted integers
16    vector<int> values(SIZE);
17
18    // Prompt the user to enter the values to be stored.
19    cout << "Please enter " << SIZE << " integers separated by spaces.\n";
20
21    for (int i = 0; i < SIZE; i++)
22        cin >> values[i];
23
```

(program continues)

Program 9-7 *(continued)*

```
24    // Display the values
25    cout << "\nThe unsorted values entered are:\n";
26    displayVector(values);
27
28    // Sort the values
29    sortVector(values);
30
31    // Display them again
32    cout << "The sorted values are:\n";
33    displayVector(values);
34    return 0;
35 }
36
37 /*********************************************************
38  *                        sortVector                     *
39  * This function performs an ascending-order bubble sort on *
40  * numbers, a vector of integers.                        *
41  *********************************************************/
42 void sortVector(vector<int> &numbers)
43 {  int  temp;
44    bool swap;
45
46    do
47    {  swap = false;
48       for (unsigned count = 0; count < numbers.size()-1; count++)
49       {
50          if (numbers[count] > numbers[count + 1])
51          {
52             temp = numbers[count];
53             numbers[count] = numbers[count + 1];
54             numbers[count + 1] = temp;
55             swap = true;
56          }
57       }
58    } while (swap);
59 }
60
61 /*********************************************************
62  *                     displayVector                     *
63  * This function displays the contents of numbers, a     *
64  * vector of integers.                                   *
65  *********************************************************/
66 void displayVector(vector<int> numbers)
67 {
68    for (unsigned count = 0; count < numbers.size(); count++)
69       cout << numbers[count] << " ";
70    cout << endl;
71 }
```

(program continues)

Program 9-7 *(continued)*

Program Output With Example Input Shown in Bold

```
Please enter 6 integers separated by spaces.
9 4 8 6 3 1[Enter]

The unsorted values entered are:
9 4 8 6 3 1
The sorted values are:
1 3 4 6 8 9
```

Notice the similarities and differences between Program 9-7 and Program 9-4. The code in Program 9-7 that sorts vectors is almost identical to the code in Program 9-4 that sorts arrays. The differences lie in some details of initialization and argument passing.

First, notice that in Program 9-4 the array data is provided in an initialization list when the array is created, but in Program 9-7 the data to be stored in the vector is input by the user. This is done because vectors do not accept initialization lists. Second, notice that in Program 9-7 the vector is passed by reference to the `sortVector` function. This is necessary because, unlike arrays, vectors are passed by value unless the programmer uses a reference variable as a parameter. Finally, notice that in Program 9-7 it is not necessary to pass the size of the vector to the functions that work with it. This is because vectors have the `size` member function to return the number of elements in the vector.

9.6 Introduction to Analysis of Algorithms

CONCEPT: **We can estimate the efficiency of an algorithm by counting the number of steps it requires to solve a problem.**

An algorithm is a mechanical step-by-step procedure for solving a problem and is the basic strategy used in designing a program. There is often more than one algorithm that can be used to solve a given problem. For example, we saw earlier in this chapter that the problem of searching a sorted array can be solved by two different methods: *sequential search* and *binary search*.

How can we decide which of two algorithms for solving a problem is better? To answer this question, we need to establish criteria for judging the "goodness" or efficiency of an algorithm. The two criteria most often used are space and time. The *space* criterion refers to the amount of memory the algorithm requires to solve the problem, while the *time* criterion refers to the length of execution time. In this chapter, we will use the time criterion to evaluate the efficiency of algorithms.

One possibility for comparing two algorithms is to code them and then time the execution of the resulting C++ programs. This experimental approach can yield useful information, but it has the following shortcomings:

- It measures the efficiency of programs rather than algorithms.
- The results obtained depend on the programming language used to code the algorithms, and on the quality of the compiler used to generate machine code. The programs may run faster or slower if they are coded in a different language, or compiled by a different compiler.

- The results obtained depend on how the operating system executes programs, and on the nature of the hardware on which the programs are executing. The execution times may be different if we run the programs on a different computer and a different operating system.
- The results obtained apply only to those inputs that were part of the execution runs and may not be representative of the performance of the algorithms using a different set of inputs.

A better approach is to count the number of basic steps an algorithm requires to process an input of a given size. To make sense of this approach, we need more precise definitions of what we mean by computational problem, problem input, input size, and basic step.

Computational Problems and Basic Steps

A *computational problem* is a problem to be solved using an algorithm. Such a problem is a collection of *instances*, with each instance specified by input data given in some prescribed format. For example, if the problem P is to sort an array of integers, then an instance of P is just a specific integer array to be sorted. The *size* of an instance refers to the amount of memory needed to hold the input data. The input size is usually given as a number that allows us to infer the total number of bits occupied by the input data. If the number of bits occupied by each entry of the array is fixed, say at 64 bits, then the length of the array is a good measure of input size. In contrast, the length of the array is not a good measure of input size if the size of array elements can vary and there is no fixed upper bound on the size of these elements.

A step executed by an algorithm is a *basic step* (also called a *basic operation*) if the algorithm can execute the step in time bounded by a constant regardless of the size of the input. In sorting an array of integers, the step

Swap the elements in positions k and k+1

is basic because the time required to swap two array elements remains constant even if the length of the array increases. In contrast, a step such as

Find the largest element of the array

is not basic because the time required to complete the step depends on the length of the array. Intuitively, a basic step is one that could conceivably be built into the hardware of some physical computer.

The definition of a basic step does not specify the size of the constant that bounds the time required to execute the step. Ignoring the exact value of these constants reflects the reality that the same operation may be executed with different speeds on different hardware, and that an operation that can be executed with one hardware instruction on one computer may require several hardware instructions on another computer. A consequence of this definition is that we can count any constant number of basic steps as one basic step. For example, an algorithm that executes $5n$ basic steps can accurately be described as executing n basic steps.

It is important to realize that ordinary arithmetic and logic operations such as addition and comparison are not basic unless a constant bound is put on the size of the numbers being

added or compared. The size of the bound does not matter as long as the bound is constant. It may be 32, 64, 128, 1024 bits, or even larger, and these operations will still be basic. In the following discussion, we assume that all the numbers used in our algorithms as inputs, outputs, or computed intermediate results are bounded in size. This allows us to consider operations on them as basic.

It only makes sense to describe an algorithm after we have described the problem the algorithm is supposed to solve. A computational problem is described by stating what the input will look like, how big it is, and what output the algorithm solving the problem is supposed to produce. These must be described clearly, so there is no ambiguity, and generally, so the algorithm can work with any data set that fits the description.

Let's look at an example. Suppose the problem *P* is to sum all the integer values in a one-dimensional array. We could describe the problem by saying that the input data is an array of *n* integer values and that the output to be produced is the integer sum of these values. Formally, this is written as follows:

INPUT: an integer array `a[]` of size *n*
SIZE OF INPUT: The number *n* of array entries
OUTPUT: An integer *sum* representing the sum total of the values stored in the array

Notice that the word INPUT used this way does not mean a set of data entered by the user, but rather means the form of the data used by the algorithm solving the problem. Likewise, the word OUTPUT used this way does not mean something displayed on the computer screen by a program. It means the result created by the algorithm that solves the problem. Because we have assumed all the array entries are of some fixed size, such as 32 or 64 bits, the number *n* of elements in the array is a good measure of input size.

Once a computational problem has been described, there can be many different algorithms designed to solve it. Some, of course, are better than others, as we will soon see. Here is one possible algorithm for solving the computational problem just described. Notice that it is expressed in pseudocode, rather than in C++ or any other particular programming language.

Algorithm 1:

```
1: sum = 0
2: k = 0 //array index
3: While k < n do
4:     sum = sum + a[k]
5:     k = k + 1
6: End While
```

Complexity of Algorithms

We can measure the complexity of an algorithm that solves a computational problem by determining the number of basic steps it requires for an input of size *n*. Let's count the number of steps required by Algorithm 1. The algorithm consists of two statements on lines 1 and 2 that are each executed once and two statements inside a loop on lines 4 and 5 that will execute once each time the loop iterates. Recall that because the statements on lines 1 and 2 perform basic operations they can be grouped together and counted as one basic operation. Let's call this operation *A*. Also, because both statements in the loop execute in constant time, independently of the size of *n*, they are also basic operations. Since the

loop body contains only basic operations, the amount of time the algorithm takes to execute a *single* iteration of the loop is also constant, and not dependent on the size of n. This allows us to count each loop iteration as a single basic operation. Let's call this operation B.

Operation A executes only one time, regardless of how big n is. Operation B executes once each time the loop iterates. Because the loop iterates n times, operation B is executed n times. Thus, the total number of operations performed is $1 + n$. When $n = 10$, for example, 11 operations are performed. When $n = 1000$, 1001 operations are performed. When n = 10,000 the number of operations performed is 10,001. Notice that as n becomes large, the 1 becomes insignificant and the number of operations performed is approximately n. We therefore say that the algorithm requires execution time proportional to n to process an input set of size n.

There is another way we could look at Algorithm 1 and determine how many operations it requires. The crucial operation in summing the values in an array is the addition of each value to the variable accumulating the sum. This occurs in line 4, and there are as many additions of array values as there are loop iterations.

Thus, we could get the same result by just counting additions of array elements. It turns out that for most algorithms, it is sufficient to identify and count only one or two basic operations that are in some way crucial to the problem being solved. For example, in many array searching and sorting algorithms, it is sufficient to just count the number of comparisons between array elements.

The array-summing algorithm just considered is particularly simple to analyze because it performs the same amount of work for all input sets of a given size.

This is not the case with all algorithms. Consider the linear search algorithm introduced earlier in this chapter. It searches through an array of values, looking for one that matches a search key. Let's call the key X. The input to the algorithm is the array of n values and the key value X. The output of the algorithm is the subscript of the array location where the value was located or, if it is not found, the determination that the loop control variable has become larger than the subscript of the last array element. Formally, the problem can be stated like this:

INPUT: An integer array $a[\ ]$ of size n, and an integer X
SIZE OF INPUT: The number n of array entries
OUTPUT: An integer k in the range $0 \leq k \leq n - 1$ such that $a[k] = X$, or $k = n$

Algorithm 2, shown here, uses the linear search algorithm to solve the problem.

Algorithm 2:

```
1: k = 0
2: While k < n and a[k] ≠ X do
3:     k = k + 1
4: End While
```

This algorithm starts at one end and searches sequentially through the array. The algorithm stops as soon as it encounters X, but will search the entire array if X is not in the array. The algorithm may stop after making only one comparison (X is found in the first entry examined), or it may not stop until it has made n comparisons (X is found in the last place examined or is not in the array). In fact, the algorithm may perform m comparisons

where m is any value from 1 to n. In cases where an algorithm may perform different amounts of work for different inputs of the same size, it is common to measure the efficiency of the algorithm by the work done on an input of size n that requires the *most* work. This is called measuring the algorithm by its worst-case complexity function.

Worst Case Complexity of Algorithms

The *worst-case complexity function* $f(n)$ of an algorithm is the number of steps it performs on an input of size n that requires the most work. It gives an indication of the longest time the algorithm will ever take to solve an instance of size n, and is a good measure of efficiency to use when we are looking for a performance guarantee.

Let's determine the worst-case complexity of binary search, which was introduced earlier in this chapter. This algorithm is used to locate an item X in an array sorted in ascending order. The worst case occurs when X is not found in the array. In this case, as we will see, the algorithm performs $L + 1$ steps, where L is the number of loop iterations.

Here is the binary search algorithm to search an array of n elements.

Algorithm 3:

```
 1: first = 0
 2: last = n - 1   // n - 1 is the subscript of the last element.
 3: found = false
 4: position = -1
 5: While found is not true and first <= last
 6:    middle = (first + last) / 2
 7:    If a[middle] = X
 8:       found = true
 9:       position = middle
10:    Else if a[middle] > X
11:       last = middle - 1
12:    Else
13:       first = middle + 1
14:    End If
15: End While
16: // When the loop terminates, position holds the subscript
17: // where the value matching X was found, or holds –1 if
18: // the value was not found.
```

The algorithm consists of some initialization of variables followed by a loop. The initialization requires constant time, and can therefore be considered to be one basic operation. Likewise, each iteration of the loop is a basic step because increasing the number of entries in the array does not increase the amount of time required by a single iteration of the loop. This shows that the number of steps required by binary search is $L + 1$. Now L is approximately equal to the integer part of $\log_2 n$, the logarithm of n to the base 2. To see this, notice that the size of the array to be searched is initially n, and each iteration reduces the size of the remaining portion of the array by approximately one half. Because each loop iteration performs at most two comparisons, binary search performs a total of $2 \log_2 n$ comparisons. We can summarize our findings as follows:

In the worst case, binary search requires time proportional to $\log_2 n$.

Let's look at one more algorithm to determine its worst case complexity. The computational problem to be solved is to arrange a set of n integers into ascending order.

INPUT: An array $a[\ ]$ of n integers
SIZE OF INPUT: The number n of array entries
OUTPUT: The array $a[\ ]$ rearranged so that $a[0] \leq a[1] \leq \ldots \leq a[n-1]$

The algorithm we will use is a modification of the selection sort algorithm introduced earlier in this chapter. This version scans for the largest element (instead of the smallest) and moves it to the end in each pass.

Algorithm 4:

```
1:  For (k = n-1; k ≥ 1; k --)
2:      // a[0..k] is what remains to be sorted
3:      Determine position p of largest entry in a[0..k]
4:         Swap a[p] with a[k]
5:  End For
```

To analyze the complexity of this algorithm, let's begin by determining the number of array entry comparisons it makes when sorting an array of n entries. These comparisons occur in step 3. Step 3 is clearly not a basic step, as it requires time proportional to k, and k varies with each iteration of the loop. To better see what is going on, let's restate step 3 using operations that are basic.

INPUT: array $a[0..k]$ of $k + 1$ entries
SIZE OF INPUT: number $k + 1$ of array entries

```
3.0:  p = 0 //Position of largest value in unsorted part of the array
3.1:  For (m = 1; m ≤ k; m ++)
3.2:     If a[m] > a[p] Then
3.3:        p = m
3.4:     End if
3.5:  End For
```

We can see that the loop in line 3.1 through 3.5 iterates k times and on line 3.2 makes one comparison each time it iterates. Therefore this algorithm requires k comparisons between array entries.

Now returning to the main sorting algorithm, we observe that there will be $n-1$ iterations of the loop that starts at line 1 and ends at line 5, one iteration for each value of k in the range $n-1$ to 1. On the first iteration, k equals $n-1$, so step 3, as we learned from the analysis of lines 3.0 through 3.5, performs $n-1$ comparisons between array elements. On the second iteration, k equals $n-2$, so step 3 performs $n-2$ comparisons. This continues until, on the final iteration, k equals 1, and step 3 performs 1 comparison. Here is what it looks like:

$k = n-1$: step 3 performs $n-1$ comparisons
$k = n-2$: step 3 performs $n-2$ comparisons
.
.
$k = 1$: step 3 performs 1 comparison

Generalizing, we can thus say that for every value of k from $n-1$ to 1, on the kth iteration, the step on line 3 will perform k comparisons.

Thus the total number of comparisons performed by this simple sorting algorithm is given by the expression

$1 + 2 + 3 + \ldots + (n - 1) = (n - 1)n/2$

For large n, this expression is very close to $n^2 / 2$. So we say that:

In the worst case, selection sort requires time proportional to n^2.

Average Case Complexity

The worst-case complexity does not, however, give a good indication of how an algorithm will perform in practical situations where inputs that yield worst case performance are rare. Often we are more interested in determining the complexity of the typical, or average case. The *average-case complexity function* can be used when we know the relative frequencies with which different inputs are likely to occur in practice. The average case complexity function uses these frequencies to form a weighted average of the number of steps performed on each input. Unfortunately, although it yields a good measure of the expected performance of an algorithm, accurate estimates of input frequencies may be difficult to obtain.

Asymptotic Complexity and the Big O Notation

We can compare two algorithms F and G for solving a problem by comparing their complexity functions. More specifically, if $f(n)$ and $g(n)$ are the complexity functions for the two algorithms, we can compare the algorithms against each other by looking at what happens to the ratio $f(n)/g(n)$ when n gets large. This is easiest to understand if this ratio tends to some limit. Let us consider some specific examples. Throughout, we assume that $f(n) \geq 1$ and $g(n) \geq 1$ for all $(n) \geq 1$.

- $f(n) = 3n^2 + 5n$ and $g(n) = n^2$. In this case

$$\frac{f(n)}{g(n)} = \frac{3n^2 + 5n}{n^2} = 3 + \frac{5}{n} \rightarrow 3 \text{ as } n \rightarrow \infty$$

That is, the value of $f(n)/g(n)$ gets closer and closer to 3 as n gets large. What this means is that for very large input sizes F performs three times as many basic operations as G. However, because the two algorithms differ in performance only by a constant factor, we consider them to be equivalent in efficiency.

- $f(n) = 3n^2 + 5n$ and $g(n) = 100n$. In this case

$$\frac{f(n)}{g(n)} = \frac{3n^2 + 5n}{100n} = \frac{3n}{100} + \frac{5}{100} \rightarrow \infty \text{ as } n \rightarrow \infty$$

Here, the ratio $f(n)/g(n)$ gets larger and larger as n gets large. This means F does a lot more work than G on large input sizes. This makes G the better algorithm for large inputs.

- $f(n) = 3n^2 + 5n$ and $g(n) = n^3$. In this case

$$\frac{f(n)}{g(n)} = \frac{3n^2 + 5n}{n^3} = \frac{3}{n} + \frac{5}{n^2} \to 0 \text{ as } n \to \infty$$

This means that for large inputs the algorithm G is doing a lot more work than F, making F the more efficient algorithm.

In general, we can compare two complexity functions $f(n)$ and $g(n)$ by looking at what happens to $f(n)/g(n)$ as n gets large. Although thinking in terms of a limit of this ratio is helpful in comparing the two algorithms, we cannot assume that such a limit will always exist. It turns out that a limit does not have to exist for us to gain useful information from this ratio. We can usefully compare the two complexity functions if we can find a positive constant K such that

$$\frac{f(n)}{g(n)} \leq K \text{ for all } n \geq 1$$

If this can be done, it means that the algorithm F is no worse than K times G for large problems. In this case, we say that $f(n)$ is in $O(g(n))$, pronounced "f is in Big O of g." The condition that defines $f(n)$ is in $O(g(n))$ is often written like this

$$f(n) \leq Kg(n) \text{ whenever } n \geq 1.$$

Showing that $f(n)$ is in $O(g(n))$ is usually straightforward. You look at the ratio $f(n)/g(n)$ and try to find a positive constant K that makes $f(n)/g(n) \leq K$ for all $n \geq 1$. For example, to show that $3n^2 + 5n$ is in $O(n^2)$, look at the ratio

$$\frac{3n^2 + 5n}{n^2} = 3 + \frac{5}{n}$$

and notice that $5/n$ will be at most 5 for all $n \geq 1$. So $3 + 5/n \leq 8$. Therefore for $K = 8$, $f(n) / g(n) \leq K$.

To show that $f(n)$ is not in $O(g(n))$, you have to show that there is no way to find a positive K that will satisfy $f(n) / g(n) \leq K$ for all $n \geq 1$. For example, the function $3n^2 + 5n$ is not in $O(n)$ because there is no constant K that satisfies

$$\frac{3n^2 + 5n}{n} = 3n + 5 \leq K \text{ for all } n \geq 1.$$

Although defined for functions, the "Big O" notation and terminology is also used to characterize algorithms and computational problems. Thus, we say that an algorithm F is in $O(g(n))$ for some function $g(n)$ if the worst case complexity function $f(n)$ of F is in Big O of $g(n)$. Accordingly, sequential search of an array is in $O(n)$ whereas binary search is in $O(\log_2 n)$.

Similarly, a computational problem is said to be in $O(g(n))$ if there exists an algorithm for the problem whose worst case complexity function is in $O(g(n))$. Thus, the problem of sorting an array is in $O(n^2)$, whereas the problem of searching a sorted array is in $O(\log_2 n)$.

If $g(n)$ is a function, $O(g(n))$ can be regarded as a family of functions that grow no faster than $g(n)$. These families are called *complexity classes,* and a few of them are important enough to merit specific names. We list them here in order of their rate of growth:

1. O(1): A function $f(n)$ is in this class if there is a constant $K > 0$ such that $f(n) \leq K$ for all $n \geq 1$. An algorithm whose worst case complexity function is in this class is said to run in *constant time*.
2. $O(\log_2 n)$: Algorithms in this class run in *logarithmic time*. Because $\log n$ grows much slower than n, a huge increase in the size of the problem results in only a small increase in the running time of the algorithm. This complexity is characteristic of search problems that eliminate half of the search space with each basic operation. The *binary search* algorithm is in this class.
3. $O(n)$: Algorithms in this class run in *linear time*. Any increase in the size of the problem results in a proportionate increase in the running time of the algorithm. This complexity is characteristic of algorithms like *sequential search* that make a single pass, or a constant number of passes, over their input.
4. $O(n \log_2 n)$: This class is called "$n \log n$" *time*. An increase in the size of the problem results in a slight increase in the running time of the algorithm. The average case complexity of *Quicksort*, a sorting algorithm you will learn about in Chapter 14, is in this class.
5. $O(n^2)$: This class is called *quadratic time*. This performance is characteristic of algorithms that make multiple passes over the input data using two nested loops. An increase in the size of the problem causes a much greater increase in the running time of the algorithm. The worst case complexity functions of *bubble sort, selection sort,* and *Quicksort* all lie in this class.

Checkpoint

9.10 What is a basic operation?

9.11 What is the worst case complexity function of an algorithm?

9.12 One algorithm needs $10n$ basic operations to process an input of size n, and another algorithm needs $25n$ basic operations to process the same input. Which of the two algorithms is more efficient? Or are they equally efficient?

9.13 What does it mean to say that $f(n)$ is in $O(g(n))$?

9.14 Show that $100n^3 + 50n^2 + 75$ is in $O(20n^3)$ by finding a positive K that satisfies the equation $(100n^3 + 50n^2 + 75) / 20n^3 \leq K$.

9.15 Assuming $g(n) \geq 1$ for all $n \geq 1$, show that every function in $O(g(n) + 100)$ is also in $O(g(n))$.

9.7 Case Studies

The following case studies, which contain applications of material introduced in Chapter 9, can be found on the student CD.

Demetris Leadership Center—Parts 1 & 2

Chapter 9 included programs illustrating how to search and sort arrays, including arrays of objects. These two case studies illustrate how to search and sort arrays of structures. Both studies develop programs for DLC, Inc., a fictional company that publishes books, DVDs, and audio CDs. DLC's inventory data, used by both programs, is stored in an array of structures.

Creating an Abstract Array Data Type—Part 2

The `IntList` class, begun as a case study in Chapter 8, is extended to include array searching and sorting capabilities.

9.8 Tying It All Together: *Secret Messages*

Now that you know how to search through an array to locate a desired item, we can write a program to encode and decode secret messages. We will use a simple substitution cipher. This means that for each character in a message, a different character will be substituted. For example, if we substitute `f` for `c`, `t` for `a`, and `x` for `t`, then the word `cat` would be written `ftx`. Can you guess what this message says?

```
*>P;HMAyJHyJH9|[3Lf
```

You'll know if you run the message through the program decoder.

For this program we'll create a `CodeMaker` class that has `encode` and `decode` functions. When a `CodeMaker` object is created, the constructor will open the `code.dat` file that contains the character substitutions to be used. This file is located, along with the program source code file, on the student CD. Be sure to place it in the project directory so the program can open and use it. There is one substitution character for each of the printable ASCII characters, which are represented by the decimal numbers 32 through 126. The program will read in these characters and store them in a one-dimensional array of characters, using the ASCII code of the original character, minus 32, as the index for the stored substitution character. So, for example, the substitution for ASCII character 32, a blank, will be stored in array element 0. The substitution for ASCII character 33, an exclamation point, will be in array element 1, and so forth.

The code will be hard to break because even the blank space will be represented by another character. So someone trying to read the code will not know where one word ends and the next one begins.

When the `encode` method is called, it is passed a string holding the message to be encoded. The method simply uses the ASCII code of each character in the string to compute the array index where its replacement character is located. Once each character in the string has been replaced, the string is returned.

When the `decode` method is called, it is passed a string holding an encoded message to be turned back into its original, or *plain text*, form. However, this method cannot compute an index to reverse the code. Instead, for each character in the string, it must do a search of the array to locate it. When the character is found, its array subscript can be used to compute the ASCII value of the original character. Once each character in the encoded string has been translated back to its original form, the string is returned.

In addition to creating the `CodeMaker` class, we will also write a client program that does the following:

- Create a `CodeMaker` object.
- Have the user input a message and store it as a string.
- Call the `encode` function, passing it the string.
- Display the returned `encoded` string.
- Call the `decode` function, passing it the encoded string.
- Display the returned decoded string. This should equal the original message.

Program 9-8 does all of this.

Program 9-8

```
// This program encodes and decodes secret messages.
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class CodeMaker
{
   private:
      int size;
      char codeChar[94];     // Array to hold the substitutions
                             // for the 94 printable ASCII chars
      int findIt(char[], int, char);

   public:
      CodeMaker();
      string encode(string);
      string decode(string);
};

// Member function implementation section

/*****************************************************
 *     CodeMaker::CodeMaker - the Constructor        *
 * This method reads the substitution characters in  *
 * from a file and stores them it the codeChar array.*
 * It also sets member variable size.                *
 *****************************************************/
CodeMaker::CodeMaker()
{
   size = 94;
   ifstream inFile;
   inFile.open("code.dat");                    // Open the file

   for (int ascii = 32; ascii < 127; ascii++)  // Read in data
      inFile >> codeChar[ascii - 32];
   inFile.close();                             // Close the file
}

/*****************************************************
 *                   CodeMaker::encode               *
 * This method encodes and returns a clear text string.*
 *****************************************************/
string CodeMaker::encode(string s)
{
   int ascii;
   char newChar;
   string newString = "";   // Will hold the encoded string

```

(program continues)

Program 9-8 *(continued)*

```
50    for (unsigned pos = 0; pos < s.length(); pos++)
51    {
52       // Get the original character's ASCII code
53       ascii = s[pos];
54
55       // Get the new replacement character
56       newChar = codeChar[ascii - 32];
57
58       // Concatenate it onto the end of the new string
59       newString += newChar;
60    }
61    return newString;
62 }
63
64 /************************************************
65  *                  CodeMaker::decode           *
66  * This method converts an encoded string back to  *
67  * clear text and returns it.                   *
68  ************************************************/
69 string CodeMaker::decode(string s)
70 {
71    int index;
72    char nextChar;
73    char originalChar;
74    string decodedText = "";
75
76    for (unsigned pos = 0; pos < s.length(); pos++)
77    {
78       // Get the next character from the string
79       nextChar = s[pos];
80
81       // Call findIt to find it in the array and return its index
82       index = findIt(codeChar, size, nextChar);
83
84       // Get the original character by computing its ASCII code
85       originalChar = index + 32;
86
87       // Concatenate this character onto the end of the
88       // decoded text string being constructed
89       decodedText += originalChar;
90    }
91    return decodedText;
92 }
93
94 /****************************************
95  *             CodeMaker::findIt            *
96  * This method performs a linear search on  *
97  * a character array looking for value.     *
98  ****************************************/
```

(program continues)

Program 9-8 *(continued)*

```
99  int CodeMaker::findIt (char A[], int size, char value)
100 {
101    int index = 0;
102    int position = -1;
103    bool found = false;
104
105    while (index < size && !found)
106    {
107       if (A[index] == value)   // If the value is found
108       {  found = true;         // Set the flag
109          position = index;     // Record the value's subscript
110       }
111       index++;                 // Go to the next element
112    }
113    return position;            // Return the position, or -1
114 }
115
116/*****************************************************
117 *                          main                     *
118 * The client "program" that uses the CodeMaker class.*
119 *****************************************************/
120 int main()
121 {
122   string originalText, secretCode, finalText;
123   CodeMaker myCoder;
124
125   // Get text from the user
126   cout << "Enter the message to be encoded.\n";
127   getline(cin, originalText);
128
129   // Send the text to be encoded and display the result
130   secretCode = myCoder.encode(originalText);
131   cout << "\nHere is the encoded message\n" << secretCode << endl;
132
133   // Send the encoded text back to be decoded
134   // and display the result
135   finalText = myCoder.decode(secretCode);
136   cout << "\nHere is the decoded message\n" << finalText << endl;
137
138   return 0;
139 }
```

Program Output with Example Input Shown in Bold

```
Enter the message to be encoded.
I can write a secret message.[Enter]

Here is the encoded message.
xH43DHP|yL[H3HJ[4|[LH=[JJ39[f

Here is the decoded message.
I can write a secret message.
```

Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. The _____ search algorithm steps sequentially through an array, comparing each item with the search value.
2. The _____ search algorithm repeatedly divides the portion of an array being searched in half.
3. The _____ search algorithm is adequate for small arrays but not large arrays.
4. The _____ search algorithm requires that the array's contents be sorted.
5. The *average* number of comparisons performed by linear search to find an item in an array of N elements is _____.
6. The *maximum* number of comparisons performed by linear search to find an item in an array of N elements is _____.
7. A linear search will find the value it is looking for with just one comparison if that value is stored in the _____ array element.
8. A binary search will find the value it is looking for with just one comparison if that value is stored in the _____ array element.
9. In a binary search, after three comparisons have been made, only _____ of the array will be left to search.
10. The maximum number of comparisons that a binary search function will make when searching for a value in a 2,000-element array is _____.
11. If an array is sorted in _____ order, the values are stored from lowest to highest.
12. If an array is sorted in _____ order, the values are stored from highest to lowest.
13. Bubble sort places _____ number(s) in place on each pass through the data.
14. Selection sort places _____ number(s) in place on each pass through the data.
15. To sort N numbers, bubble sort continues making passes through the array until _____.
16. To sort N numbers, selection sort makes _____ passes through the data.
17. Why is selection sort more efficient than bubble sort on large arrays?
18. Which sort, bubble sort or selection sort, would require fewer passes to sort a set of data that is already in the desired order?
19. Complete the following table by calculating the average and maximum number of comparisons the linear search will perform, and the maximum number of comparisons the binary search will perform.

Array Size →	50 Elements	500 Elements	10,000 Elements	100,000 Elements	10,000,000 Elements
Linear Search (Average Comparisons)					
Linear Search (Maximum Comparisons)					
Binary Search (Maximum Comparisons)					

Algorithm Workbench

20. Assume that `empName` and `empID` are two parallel arrays of size `numEmp` that hold employee data. Write a pseudocode algorithm that sorts the `empID` array in ascending ID number order (using any sort you wish), such that the two arrays remain parallel. That is, after sorting, for all indexes in the arrays, `empName[index]` must still be the name of the employee whose ID is in `empID[index]`.

21. Assume an array of structures is in order by the customerID field of the record, where customer IDs go from 101 to 500.

    A) Write the most efficient pseudocode algorithm you can to find the record with a specific customerID if every single customer ID from 101 to 500 is used and the array has 400 elements.

    B) Write the most efficient pseudocode algorithm you can to find a record with a customer ID near the end of the IDs, say 494, if not every single customer ID in the range of 101 to 500 is used and the array size is only 300.

Soft Skills

Deciding how to organize and access data is an important part of designing a program. You are already familiar with many structures and methods that allow you to organize data. These include one-dimensional arrays, vectors, multidimensional arrays, parallel arrays, structures, classes, arrays of structures, and arrays of class objects. You are also now familiar with some techniques for arranging (i.e., sorting) data and for locating (i.e., searching for) data items.

22. Team up with two to three other students and jointly decide how you would organize, order, and locate the data used in the following application. Be prepared to present your group's design to the rest of the class.

    The program to be developed is a menu-driven program that will keep track of parking tickets issued by the village that is hiring you. When a ticket is issued the program must be able to accept and store the following information: ticket number, officer number, vehicle license plate state and number, location, violation code (this indicates which parking law was violated), and date and time written. The program must store information on the amount of the fine associated with each violation code. When a ticket is paid the program must be able to accept and store the information that it has been paid, the amount of the payment, and the date the payment was received. The program must be able to accept inquiries such as displaying the entire ticket record when a ticket number is entered. The program must also be able to produce the following reports:

    - A list of all tickets issued on a specific date, ordered by ticket number
    - A list of all tickets for which payment was received on a specific date and the total amount of money collected that day
    - A report of all tickets issued in a one-month period, ordered by officer number, with a count of how many tickets each officer wrote
    - A report of all tickets that have not yet been paid, or for which payment received was less than payment due, ordered by vehicle license number

Programming Challenges

These programming challenges can all be written either with or without the use of classes. Your instructor will tell you which approach you should use.

1. Charge Account Validation

Write a program that lets the user enter a charge account number. The program should determine if the number is valid by checking for it in the following list:

5658845	4520125	7895122	8777541	8451277	1302850
8080152	4562555	5552012	5050552	7825877	1250255
1005231	6545231	3852085	7576651	7881200	4581002

Initialize a one-dimensional array with these values. Then use a simple linear search to locate the number entered by the user. If the user enters a number that is in the array, the program should display a message saying the number is valid. If the user enters a number not in the array, the program should display a message indicating it is invalid.

2. Lottery Winners



VideoNote
Solving the Lottery Winners Problem

A lottery ticket buyer purchases ten tickets a week, always playing the same ten five-digit "lucky" combinations. Write a program that initializes an array with these numbers and then lets the player enter this week's winning five-digit number. The program should perform a linear search through the list of the player's numbers and report whether or not one of the tickets is a winner this week. Here are the numbers:

13579	26791	26792	33445	55555
62483	77777	79422	85647	93121

3. Lottery Winners Modification

Modify the program you wrote for Programming Challenge 2 (Lottery Winners) so it performs a binary search instead of a linear search.

4. Annual Rainfall Report

Write a program that displays the name of each month in a year and its rainfall amount, sorted in order of rainfall from highest to lowest. The program should use an array of structures, where each structure holds the name of a month and its rainfall amount. Use a constructor to set the month names. Make the program modular by calling on different functions to input the rainfall amounts, to sort the data, and to display the data.

5. Hit the Slopes

Write a program that can be used by a ski resort to keep track of local snow conditions for one week. It should have a seven-element array of structures, where each structure holds a date and the number of inches of snow in the base on that date. The program should have the user input the name of the month, the starting and ending date of the seven-day period being measured, and then the seven base snow depths. The program should then sort the data in ascending order by base depth and display the results. Here is a sample report.

```
Snow Report December 12 – 18
   Date  Base
    13   42.3
    12   42.5
    14   42.8
    15   43.1
    18   43.1
    16   43.4
    17   43.8
```

6. String Selection Sort

Modify the `selectionSort` function presented in this chapter so it sorts an array of strings instead of an array of `int`s. Test the function with a driver program. Use Program 9-9 as a skeleton to complete.

Program 9-9

```
// Include needed header files here.

int main()
{
   const int SIZE = 20;

   string name[SIZE] =
   {"Collins, Bill",  "Smith, Bart",  "Michalski, Joe", "Griffin, Jim",
    "Sanchez, Manny", "Rubin, Sarah", "Taylor, Tyrone", "Johnson, Jill",
    "Allison, Jeff",  "Moreno, Juan", "Wolfe, Bill",    "Whitman, Jean",
    "Moretti, Bella", "Wu, Hong",     "Patel, Renee",   "Harrison, Rose",
    "Smith, Cathy",   "Conroy, Pat",  "Kelly, Sean",    "Holland, Beth"};

   // Insert your code to complete this program.
}
```

7. Binary String Search

Modify the `binarySearch` function presented in this chapter so it searches an array of strings instead of an array of `int`s. Test the function with a driver program. Use Program 9-8 as a skeleton to complete. (The array must be sorted before the binary search will work.)

8. Search Benchmarks

Write a program that has an array of at least 20 integers. It should call a function that uses the linear search algorithm to locate one of the values. The function should keep a count of the number of comparisons it makes until it finds the value. The program then should call a function that uses the binary search algorithm to locate the same value. It should also keep count of the number of comparisons it makes. Display these values on the screen.



9. Sorting Benchmarks

Write a program that uses two identical arrays of at least 20 integers. It should call a function that uses the bubble sort algorithm to sort one of the arrays in ascending order. The function should count the number of exchanges it makes. The program should then call a function that uses the selection sort algorithm to sort the other array. It should also count the number of exchanges it makes. Display these values on the screen.

10. Sorting Orders

Write a program that uses two identical arrays of eight integers. It should display the contents of the first array, then call a function to sort the array using an ascending order bubble sort modified to print out the array contents after each pass of the sort. Next the program should display the contents of the second array, then call a function to sort the array using an ascending order selection sort modified to print out the array contents after each pass of the sort.

11. Ascending Circles

Program 8-31 from Chapter 8 creates an array of four Circle objects, then displays the area of each object. Using a copy of that program as a starting point, modify it to create an array of eight Circle objects initialized with the following radii: 2.5, 4.0, 1.0, 3.0, 6.0, 5.5, 3.5, 2.0. Then use a bubble sort to arrange the objects in ascending order of radius size before displaying the area of each object.

12. Modified Bin Manager Class

Modify the `BinManager` class you wrote for Programming Challenge 14 in Chapter 8 to overload its `getQuantity`, `addParts`, and `removeParts` functions as shown here:

```
bool addParts(string itemDescription, int q);
bool removeParts(string itemDescription, int q)
int getQuantity(string itemDescription);
```

These new functions allow parts to be added, parts to be removed, and the quantity in stock for a particular item to be retrieved by using an *item description*, rather than a bin number, as an argument. In addition to writing the three overloaded functions, you will need to create a `private BinManager` class function that uses the item description as a search key to locate the index of the desired bin.

Test the new class functions with the same client program you wrote for Programming Challenge 14 in Chapter 8, modifying it to call the new functions. Be sure to use some descriptions that match bins in the array and some that do not.

As you did in the previous Bin Manager program, if an add or remove operation is successfully carried out, make the function return `true`. If it cannot be done—for example, because the string passed to it does not match any item description in the array—make the function return `false`. If the `getQuantity` function cannot locate any item whose description matches the one passed to it, make it return –1.

13. Using Files—String Selection Sort Modification

Modify the program you wrote for Programming Challenge 6 so it reads in the 20 strings from a file. The data can be found in the `names.dat` file.

14. Using Vectors—String Selection Sort Modification

Modify the program you wrote for Programming Challenge 13 so it stores the names in a vector of strings, rather than in an array of strings. Create the vector without specifying a size and then use the `push_back` member function to add an element holding each string to the vector as it is read in from a file. Instead of assuming there are always 20 strings, read in the strings and add them to the vector until there is no data left in the file. The data can be found in the `names.dat` file.

CHAPTER 10

Pointers

TOPICS

10.1 Pointers and the Address Operator
10.2 Pointer Variables
10.3 The Relationship Between Arrays and Pointers
10.4 Pointer Arithmetic
10.5 Initializing Pointers
10.6 Comparing Pointers
10.7 Pointers as Function Parameters
10.8 Pointers to Constants and Constant Pointers
10.9 Focus on Software Engineering: *Dynamic Memory Allocation*
10.10 Focus on Software Engineering: *Returning Pointers from Functions*
10.11 Pointers to Class Objects and Structures
10.12 Focus on Software Engineering: *Selecting Members of Objects*
10.13 United Cause Relief Agency Case Study
10.14 Tying It All Together: *Pardon Me, Do You Have the Time?*

10.1 Pointers and the Address Operator

CONCEPT: **Every variable is assigned a memory location whose address can be retrieved using the address operator &. The address of a memory location is called a *pointer.***

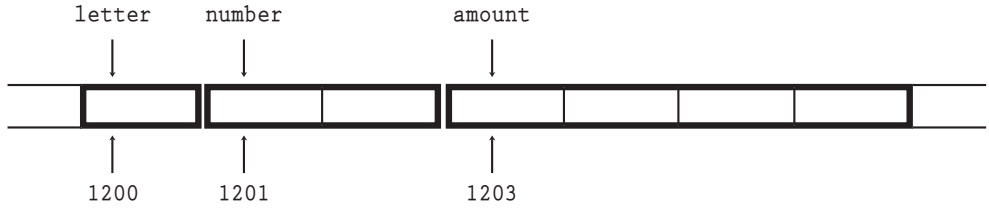
Every variable in an executing program is allocated a section of memory large enough to hold a value of that variable's type. Current C++ compilers that run on PCs usually allocate a single byte to variables of type `char`, two bytes to variables of type `short`, four bytes to variables of type `float` and `long`, and 8 bytes to variables of type `double`.

Each byte of memory has a unique address. A variable's address is the address of the first byte allocated to that variable. Suppose that the following variables are defined in a program:

```
char letter;
short number;
float amount;
```

Figure 10-1 illustrates how they might be arranged in memory and shows their addresses.

Figure 10-1



In Figure 10-1, the variable `letter` is shown at address 1200, `number` is at address 1201, and `amount` is at address 1203.

The addresses of the variables shown in Figure 10.1 are somewhat arbitrary, and are used for illustrative purposes only. In fact, most compilers allocate space in such a way that individual variables are always assigned even addresses. This is because current computer hardware can access data that resides at even addresses faster than data that resides at odd addresses.

C++ has an *address operator* & that can be used to retrieve the address of any variable. To use it, place it before the variable whose address you want. Here is an expression that returns the address of the variable `amount`:

```
&amount
```

And here is a statement that displays the variable's address to the screen:

```
cout << long(&amount);
```

By default, C++ prints addresses in hexadecimal. Here we have used a function-style cast to `long` to make the address print in the usual decimal format. Program 10-1 demonstrates the use of the address operator to display addresses of variables.

Program 10-1

```
1 // This program uses the & operator to determine a
2 // variable's address.
3 #include <iostream>
4 using namespace std;
5
6 char letter;
7 short number;
8 float amount
```

(program continues)

Program 10-1 *(continued)*

```
 9 double profit;
10 char ch;
11
12 int main()
13 {
14    // Print address of each variable.
15    // The cast to long makes addresses print in decimal
16    // rather than in hexadecimal.
17    cout << "Address of letter is:  "
18         << long(&letter) << endl;
19    cout << "Address of number is:  "
20         << long(&number) << endl;
21    cout << "Address of amount is:  "
22         << long(&amount) << endl;
23    cout << "Address of profit is:  "
24         << long(&profit) << endl;
25    cout << "Address of ch is:      "
26         << long(&ch) << endl;
27    return 0;
28 }
```

Program Output

```
Address of letter is:  4468752
Address of number is:  4468754
Address of amount is:  4468756
Address of profit is:  4468760
Address of ch is:      4468768
```

The value `&amount` specifies the location of the variable `amount` in the computer's memory: in a sense, it points to `amount`. A value that represents the address of a memory location, or holds the address of some variable, is called a *pointer*.

10.2 Pointer Variables

CONCEPT: A pointer variable is a variable that holds addresses of memory locations.



VideoNote
Pointer Variables

Like other data values, memory addresses, or pointer values, can be stored in variables of the appropriate type. A variable that stores an address is called a *pointer variable*, but is often simply referred to as just a *pointer*. The definition of a pointer variable, say `ptr`, must specify the type of data that `ptr` will point to. Here is an example:

```
int *ptr;
```

The asterisk before the variable name indicates that `ptr` is a pointer variable, and the `int` data type indicates that `ptr` can only be used to point to, or hold addresses of, integer variables. This definition is read as "`ptr` is a pointer to `int`." It is also useful to think of `*ptr` as the "variable that `ptr` points to." With this view, the definition of `ptr` just given can be read as "the variable that `ptr` points to has type `int`." Because the asterisk (`*`) allows you to pass from a pointer to the variable being pointed to, it is called the *indirection operator*.

Some programmers prefer to declare pointers with the asterisk next to the type name, rather than the variable name. For example, the declaration shown above could be written as:

```
int* ptr;
```

This style of declaration might visually reinforce the fact that `ptr`'s data type is not `int`, but pointer-to-`int`. Both declaration styles are correct.

Program 10-2 demonstrates a very simple usage of a pointer: storing and printing the address of another variable.

Program 10-2

```
1  // This program stores the address of a variable in a pointer.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int x = 25;  // int variable
8     int *ptr;    // Pointer variable, can point to an int
9
10    ptr = &x;    // Store the address of x in ptr
11    cout << "The value in x is " << x << endl;
12    cout << "The address of x is " << ptr << endl;
13    return 0;
14 }
```

Program Output

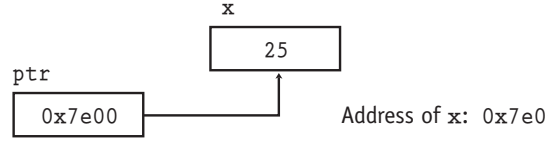
```
The value in x is 25
The address of x is 0x7e00
```

In Program 10-2, two variables are defined: `x` and `ptr`. The variable `x` is an `int`, while `ptr` is a pointer to an `int`. The variable `x` is initialized with `25`, while `ptr` is assigned the address of `x` with the following statement:

```
ptr = &x;
```

Figure 10-2 illustrates the relationship between `ptr` and `x`.

Figure 10-2



As shown in Figure 10-2, the variable `x` is located at memory address `0x7e00` and contains the number `25`, while the pointer `ptr` contains the address `0x7e00`. In essence, `ptr` "points" to the variable `x`.

You can use a pointer to indirectly access and modify the variable being pointed to. In Program 10-2, for instance, `ptr` could be used to change the contents of the variable `x`. When the indirection operator is placed in front of a pointer variable name, it *dereferences* the pointer. When you are working with a dereferenced pointer, you are actually working with the value the pointer is pointing to. This is demonstrated in Program 10-3.

Program 10-3

```
1  // This program demonstrates the use of the indirection
2  // operator.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 25;      // int variable
9     int *ptr;        // Pointer variable, can point to an int
10
11    ptr = &x;        // Store the address of x in ptr
12
13    // Use both x and ptr to display the value in x.
14    cout << "Here is the value in x, printed twice:\n";
15    cout << x << "  " << *ptr << endl;
16
17    // Assign 100 to the location pointed to by ptr.
18    // This will actually assign 100 to x.
19    *ptr = 100;
20
21    // Use both x and ptr to display the value in x.
22    cout << "Once again, here is the value in x:\n";
23    cout << x << "  " << *ptr << endl;
24    return 0;
25 }
```

Program Output

```
Here is the value in x, printed twice:
25  25
Once again, here is the value in x:
100  100
```

Every time the expression `*ptr` appears in Program 10-3, the program indirectly uses the variable `x`. The following `cout` statement displays the value in `x` twice:

```
cout << x << "  " << *ptr << endl;
```

And the following statement stores 100 in `x`:

```
*ptr = 100;
```

With the indirection operator, `ptr` can be used to indirectly access the variable it is pointing to. Program 10-4 demonstrates that pointers can point to different variables.

Program 10-4

```
1  // This program demonstrates the ability of a pointer to
2  // point to different variables.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 25, y = 50, z = 75;        // Three int variables
9     int *ptr;                          // Pointer variable
10
11    // Display the contents of x, y, and z.
12    cout << "Here are the values of x, y, and z:\n";
13    cout << x << "  " << y << "  " << z << endl;
14
15    // Use the pointer to manipulate x, y, and z.
16
17    ptr = &x;          // Store the address of x in ptr
18    *ptr *= 2;         // Multiply value in x by 2
19
20    ptr = &y;          // Store the address of y in ptr
21    *ptr *= 2;         // Multiply value in y by 2
22
23    ptr = &z;          // Store the address of z in ptr
24    *ptr *= 2;         // Multiply value in z by 2
25
26    //Display the contents of x, y, and z.
27    cout  << "Once again, here are the values "
28          << "of x, y, and z:\n";
29    cout  << x << "  " << y << "  " << z << endl;
30    return 0;
31 }
```

Program Output

```
Here are the values of x, y, and z:
25  50  75
Once again, here are the values of x, y, and z:
50  100  150
```



NOTE: So far you've seen three different uses of the asterisk in C++:

- As the multiplication operator, in statements such as

  ```
  distance = speed * time;
  ```

- In the definition of a pointer variable, such as

  ```
  int *ptr;
  ```

- As the indirection operator, in statements such as

  ```
  *ptr = 100;
  ```

10.3 The Relationship Between Arrays and Pointers

CONCEPT: **Array names can be used as pointer constants, and pointers can be used as array names.**

You learned earlier that an array name, without brackets and a subscript, actually represents the starting address of the array. This means that an array name is really a pointer. Program 10-5 illustrates this by showing an array name being used with the indirection operator.

Program 10-5

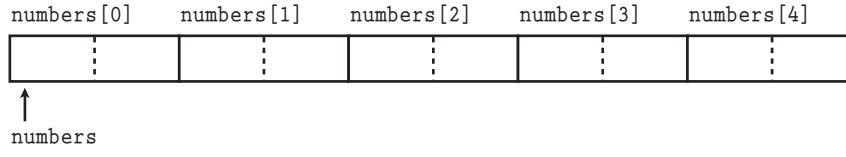
```
1  // This program shows an array name being dereferenced with the *
2  // operator.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     short numbers[] = {10, 20, 30, 40, 50};
9
10    cout << "The first element of the array is ";
11    cout << *numbers << endl;
12    return 0;
13 }
```

Program Output

```
The first element of the array is 10
```

Because `numbers` works like a pointer to the starting address of the array in Program 10-5, the first element is retrieved when `numbers` is dereferenced. So, how could the entire contents of an array be retrieved using the indirection operator? Remember, array elements are stored together in memory, as illustrated in Figure 10-3.

Figure 10-3



It makes sense that if `numbers` is the address of `numbers[0]`, values could be added to `numbers` to get the addresses of the other elements in the array. It's important to know, however, that pointers do not work like regular variables when used in mathematical statements. In C++, when you add a value to a pointer, you are actually adding that value *times the size of the data type being referenced by the pointer.* In other words, if you add one to

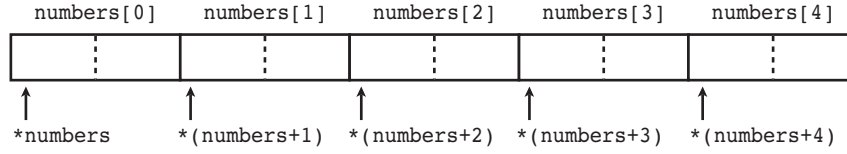
numbers, you are actually adding `1 * sizeof(short)` to numbers. If you add two to numbers, the result is `numbers + 2 * sizeof(short)`, and so forth. On a PC, this means the following are true, because short integers typically use 2 bytes:

`*(numbers + 1)` is the value at address `numbers + 1 * 2`
`*(numbers + 2)` is the value at address `numbers + 2 * 2`
`*(numbers + 3)` is the value at address `numbers + 3 * 2`

and so forth.

This automatic conversion means that an element in an array can be retrieved by using its subscript or by adding its subscript to a pointer to the array. If the expression `*numbers`, which is the same as `*(numbers + 0)`, retrieves the first element in the array, then `*(numbers + 1)` retrieves the second element. Likewise, `*(numbers + 2)` retrieves the third element, and so forth. Figure 10-4 shows the equivalence of subscript notation and pointer notation.

Figure 10-4





NOTE: The parentheses are critical when adding values to pointers. The * operator has precedence over the + operator, so the expression `*numbers + 1` is not equivalent to `*(numbers + 1)`. The expression `*numbers + 1` adds one to the contents of the first element of the array, while `*(numbers + 1)` adds one to the address in numbers, then dereferences it.

Program 10-6 shows the entire contents of the array being accessed, using pointer notation.

Program 10-6

```
1  // This program processes an array using pointer notation.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     const int SIZE = 5;     // Size of the array
8     int numbers[SIZE];      // Array of integers
9
10    // Get values to store in the array.
11    // Use pointer notation instead of subscripts.
12    cout << "Enter " << SIZE << " numbers: ";
13    for (int count = 0; count < SIZE; count++)
14       cin >> *(numbers + count);
15
```

(program continues)

Program 10-6 *(continued)*

```
16    // Display the values in the array.
17    // Use pointer notation instead of subscripts.
18    cout << "Here are the numbers you entered:\n";
19    for (int count = 0; count < SIZE; count++)
20       cout << *(numbers + count) << " ";
21    cout << endl;
22    return 0;
23 }
```

Program Output with Example Input Shown in Bold

```
Enter 5 numbers: 5 10 15 20 25[Enter]
Here are the numbers you entered:
5 10 15 20 25
```

When working with arrays, remember the following rule:

`array[index]` is equivalent to `*(array + index)`



WARNING! Remember that C++ performs no bounds checking with arrays. When stepping through an array with a pointer, it's possible to give the pointer an address outside of the array.

To demonstrate just how close the relationship is between array names and pointers, look at Program 10-7. It defines an array of `doubles` and a `double` pointer, which is assigned the starting address of the array. Not only is pointer notation then used with the array name, but subscript notation is used with the pointer!

Program 10-7

```
1  // This program uses subscript notation with a pointer
2  // variable and pointer notation with an array name.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     const int NUM_COINS = 5;
10    double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
11    double *doublePtr;   // Pointer to a double
12
13    // Assign the address of the coins array to doublePtr.
14    doublePtr = coins;
15
16    // Display the contents of the coins array.
17    // Use subscripts with the pointer!
18    cout << setprecision(2);
19    cout << "Here are the values in the coins array:\n";
20    for (int count = 0; count < NUM_COINS; count++)
21       cout << doublePtr[count] << " ";
22
```

(program continues)

Program 10-7 *(continued)*

```
23    // Display the contents of the coins array again, but
24    // this time use pointer notation with the array name!
25    cout << "\nAnd here they are again:\n";
26    for (int count = 0; count < NUM_COINS; count++)
27       cout << *(coins + count) << " ";
28    cout << endl;
29    return 0;
30 }
```

Program Output

```
Here are the values in the coins array:
0.05 0.1 0.25 0.5 1
And here they are again:
0.05 0.1 0.25 0.5 1
```

Notice that the address operator is not needed when an array's address is assigned to a pointer. Since the name of an array is already an address, use of the `&` operator would be incorrect. You can, however, use the address operator to get the address of an individual element in an array. For instance, `&numbers[1]` gets the address of `numbers[1]`. This technique is used in Program 10-8.

Program 10-8

```
1  // This program uses the address of each element in the array.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  int main()
7  {
8     const int NUM_COINS = 5;
9     double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
10    double *doublePtr; // Pointer to a double
11
12    // Use the pointer to display the values in the array.
13    cout << setprecision(2);
14    cout << "Here are the values in the coins array:\n";
15    for (int count = 0; count < NUM_COINS; count++)
16    {
17       doublePtr = &coins[count];
18       cout << *doublePtr << " ";
19    }
20    cout << endl;
21    return 0;
22 }
```

Program Output

```
Here are the values in the coins array:
0.05 0.1 0.25 0.5 1
```

The only difference between array names and pointer variables is that you cannot change the address an array name points to. For example, given the following definitions:

```
double readings[20], totals[20];
double *dptr;
```

These statements are legal:

```
dptr = readings; // Make dptr point to readings.
dptr = totals;   // Make dptr point to totals.
```

But these are illegal:

```
readings = totals;   // ILLEGAL! Cannot change readings
totals = dptr;       // ILLEGAL! Cannot change totals
```

Array names are *pointer constants*. You can't make them point to anything but the array they represent.

10.4 Pointer Arithmetic

CONCEPT: Some mathematical operations may be performed on pointers.

The contents of pointer variables may be changed with mathematical statements that perform addition or subtraction. This is demonstrated in Program 10-9. The first loop increments the pointer variable, stepping it through each element of the array. The second loop decrements the pointer, stepping it through the array backwards.

Program 10-9

```
1  // This program uses a pointer to display
2  // the contents of an array.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int SIZE = 8;
9     int set[ ] = {5, 10, 15, 20, 25, 30, 35, 40};
10    int *numPtr;   // Pointer
11
12    // Make numPtr point to the set array.
13    numPtr = set;
14
15    // Use the pointer to display the array elements.
16    cout << "The numbers in set are:\n";
17       for (int index = 0; index < SIZE; index++)
18    {
19       cout << *numPtr << " ";
20       numPtr++;
21    }
22
```

(program continues)

Program 10-9 *(continued)*

```
23    // Display the array elements in reverse order.
24    cout << "\nThe numbers in set backwards are:\n";
25    for (int index = 0; index < SIZE; index++)
26    {
27       numPtr--;
28       cout << *numPtr << " ";
29    }
30    return 0;
31 }
```

Program Output

```
The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backwards are:
40 35 30 25 20 15 10 5
```



NOTE: Because `numPtr` is a pointer, the increment operator adds the size of one integer to `numPtr`, so it points to the next element in the array. Likewise, the decrement operator subtracts the size of one integer from the pointer.

Not all arithmetic operations may be performed on pointers. For example, you cannot use multipication or division with pointers. The following operations are allowable:

- The ++ and -- operators may be used to increment or decrement a pointer variable.
- An integer may be added to or subtracted from a pointer variable. This may be performed with the + and - operators, or the += and -= operators.
- A pointer may be subtracted from another pointer.

10.5 Initializing Pointers

CONCEPT: Pointers may be initialized with the address of an existing object.

Remember that a pointer is designed to point to an object of a specific data type. When a pointer is initialized with an address, it must be the address of an object the pointer can point to. For instance, the following definition of `pint` is legal because `myValue` is an integer:

```
int myValue;
int *pint = &myValue;
```

The following is also legal because `ages` is an array of integers:

```
int ages[20];
int *pint = ages;
```

But the following definition of `pint` is illegal because `myFloat` is not an `int`:

```
float myFloat;
int *pint = &myFloat;   // Illegal!
```

Pointers may be defined in the same statement as other variables of the same type. The following declaration defines an integer variable, `myValue`, and then defines a pointer, `pint`, which is initialized with the address of `myValue`:

```
int myValue, *pint = &myValue;
```

And the following definition defines an array, `readings`, and a pointer, `marker`, which is initialized with the address of the first element in the array:

```
double readings[50], *marker = readings;
```

Of course, a pointer can only be initialized with the address of an object that has already been defined. The following is illegal because `pint` is being initialized with the address of an object that does not exist yet:

```
int *pint = &myValue;   // Illegal!
int myValue;
```

In most computers, memory at address 0 is inaccessible to user programs because it is occupied by operating system data structures. This fact allows programmers to signify that a pointer variable does not point to a memory location accessible to the program by initializing the pointer to 0. For example, if `ptrToint` is a pointer to `int`, and `ptrTofloat` is a pointer to `float`, we can indicate that neither of them points to a legitimate address by assigning 0 to both:

```
int *ptrToint = 0;
float *ptrTofloat = 0;
```

Many header files, including `iostream`, `fstream`, and `cstdlib`, define a constant named `NULL` whose value is zero. Thus, assuming one of these header files has been included, the code can be written as

```
int *ptrToint = NULL;
float *ptrTofloat = NULL;
```

A pointer whose value is the address 0 is often called a *null* pointer.



Checkpoint

10.1 Write a statement that displays the address of the variable `count`.

10.2 Write a statement defining a variable `dPtr`. The variable should be a pointer to a `double`.

10.3 List three uses of the `*` symbol in C++.

10.4 What is the output of the following program?

```
#include <iostream>
using namespace std;
int main()
{
   int x = 50, y = 60, z = 70;
   int *ptr;

   cout << x << "  " << y << "  " << z << endl;
   ptr = &x;
```

```
    *ptr *= 10;
    ptr = &y;
    *ptr *= 5;
    ptr = &z;
    *ptr *= 2;
    cout << x << "  " << y << "  " << z << endl;
    return 0;
}
```

10.5 Rewrite the following loop so it uses pointer notation (with the indirection operator) instead of subscript notation.

```
for (int x = 0; x < 100; x++)
    cout << array[x] << endl;
```

10.6 Assume `ptr` is a pointer to an `int` and holds the address 12000. On a system with 4-byte integers, what address will be in `ptr` after the following statement?

```
ptr += 10;
```

10.7 Assume `pint` is a pointer variable. For each of the following statements, determine whether the statement is valid or invalid. For those that are invalid, explain why.

A)
```
pint++;
```
B)
```
--pint;
```
C)
```
pint /= 2;
```
D)
```
pint *= 4;
```
E)
```
pint += x;  // Assume x is an int.
```

10.8 For each of the following variable definitions, determine whether the statement is valid or invalid. For those that are invalid, explain why.

A)
```
int ivar;
int *iptr = &ivar;
```
B)
```
int ivar, *iptr = &ivar;
```
C)
```
float fvar;
int *iptr = &fvar;
```
D)
```
int nums[50], *iptr = nums;
```
E)
```
int *iptr = &ivar;
int ivar;
```

10.6 Comparing Pointers

CONCEPT: C++'s relational operators may be used to compare pointer values.

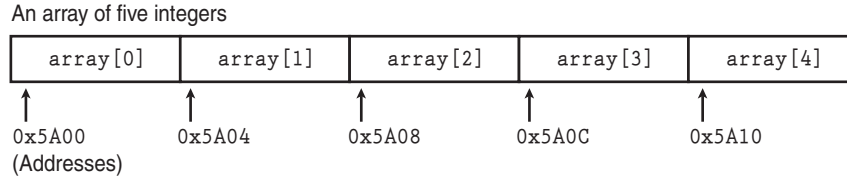
Pointers may be compared by using any of C++'s relational operators:

```
> < == != >= <=
```

If one address comes before another address in memory, the first address is considered "less than" the second. In an array, all the elements are stored in consecutive memory locations,

so the address of element 1 is greater than the address of element 0. This is illustrated in Figure 10-5.

Figure 10-5



Because the addresses grow larger for each subsequent element in the array, the following boolean expressions are all true:

```
&array[1] > &array[0]
array < &array[4]
array == &array[0]
&array[2] != &array[3]
```



NOTE: Comparing two pointers is not the same as comparing the values the two pointers point to. For example, the following `if` statement compares the addresses stored in the pointer variables `ptr1` and `ptr2`:

```
if (ptr1 < ptr2)
```

The following statement, however, compares the values that `ptr1` and `ptr2` point to:

```
if (*ptr1 < *ptr2)
```

The capability of comparing addresses gives you another way to be sure a pointer does not go beyond the boundaries of an array. Program 10-10 initializes the pointer `numPtr` with the starting address of the array `set`. The pointer `numPtr` is then stepped through the array `set` until the address it contains is equal to the address of the last element of the array. Then the pointer is stepped backwards through the array until it points to the first element.

Program 10-10

```
1  // This program uses a pointer to display the contents
2  // of an integer array. It illustrates the comparison of
3  // pointers.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     const int SIZE = 8;
10    int set[ ] = {5, 10, 15, 20, 25, 30, 35, 40};
11    int *numPtr = set;        // Make numPtr point to set
12
13    cout << "The numbers in set are:\n";
14    cout << *numPtr << " ";   // Display first element
```

(program continues)

Program 10-10 *(continued)*

```
15    while (numPtr < &set[SIZE-1])
16    {
17       // Advance numPtr to the next element.
18       numPtr++;
19       // Display the value pointed to by numPtr.
20       cout << *numPtr << " ";
21    }
22
23    // Display the numbers in reverse order
24    cout << "\nThe numbers in set backwards are:\n";
25    cout << *numPtr << " "; // Display last element
26    while (numPtr > set)
27    {
28       // Move backward to the previous element.
29       numPtr--;
30       // Display the value pointed to by numPtr.
31       cout << *numPtr << " ";
32    }
33    return 0;
34 }
```

Program Output

```
The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backwards are:
40 35 30 25 20 15 10 5
```

Most comparisons involving pointers compare a pointer to 0 (`NULL`) to determine whether the pointer points to a legitimate address. For example, assuming that `ptrToInt` has been defined as a pointer to `int`, the code

```
if (ptrToInt != 0)
   cout << *ptrToInt;
else
   cout << "null pointer";
```

prints the integer pointed to by the pointer only after checking that the pointer is not 0. Many programmers, when checking to see if the value of a pointer such as `ptrToInt` is different from 0, omit the comparison to 0 and simply write

```
if (ptrToInt)
   cout << *ptrToInt;
else
   cout << "null pointer";
```

The two ways of writing the test turn out to be equivalent because when

```
ptrToInt != 0
```

is true, `ptrToInt` is nonzero and, as you learned in Chapter 4, nonzero values are interpreted as being true. This means that the expressions `ptrToInt != 0` and `ptrToInt` have the same truth value.

10.7 Pointers as Function Parameters

CONCEPT: A pointer can be used as a function parameter. It gives the function access to the original argument, much like a reference parameter does.

In Chapter 6 you were introduced to the concept of reference variables being used as function parameters. A reference variable acts as an alias to the original variable used as an argument. This gives the function access to the original argument variable, allowing it to change the variable's contents. When a variable is passed into a reference parameter, the argument is said to be passed by reference.

An alternative to passing an argument by reference is to use a pointer variable as the parameter. Admittedly, reference variables are much easier to work with than pointers. Reference variables hide all the "mechanics" of dereferencing and indirection. You should still learn to use pointers as function arguments, however, because some tasks, especially when dealing with C-strings, are best done with pointers.* Also, the C++ library has many functions that use pointers as parameters.

Here is the definition of a function that uses a pointer parameter:

```
void doubleValue(int *val)
{
   *val *= 2;
}
```

The purpose of this function is to double the variable pointed to by `val` with the following statement:

```
*val *= 2;
```

When `val` is dereferenced, the *= operator works on the variable pointed to by `val`. This statement multiplies the original variable, whose address is stored in `val`, by two. Of course, when the function is called, the address of the variable that is to be doubled must be used as the argument, not the variable itself.

Here is an example of a call to the `doubleValue` function:

```
doubleValue(&number);
```

This statement uses the address operator (`&`) to pass the address of `number` into the `val` parameter. After the function executes, the contents of `number` will have been multiplied by two.

The use of this function is illustrated in Program 10-11.

* It is also important to learn the technique in case you ever have to write a C program. In C, the only way to get the effect of pass by reference is to use a pointer.

Program 10-11

```
1  // This program uses two functions that accept
2  // addresses of variables as arguments.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototypes
7  void getNumber(int *);
8  void doubleValue(int *);
9
10 int main()
11 {
12    int number;
13
14    // Call getNumber and pass the address of number.
15    getNumber(&number);
16
17    // Call doubleValue and pass the address of number.
18    doubleValue(&number);
19
20    // Display the value in number.
21    cout << "That value doubled is " << number << endl;
22    return 0;
23 }
24
25 //*****************************************************
26 // Definition of getNumber. The parameter, input, is a *
27 // pointer. This function asks the user for a number.  *
28 // The value entered is stored in the variable         *
29 // pointed to by input.                                *
30 //*****************************************************
31
32 void getNumber(int *input)
33 {
34    cout << "Enter an integer number: ";
35    cin >> *input;
36 }
37
38 //*****************************************************
39 // Definition of doubleValue. The parameter, val, is a *
40 // pointer. This function multiplies the variable      *
41 // pointed to by val by two.                           *
42 //*****************************************************
43
44 void doubleValue(int *val)
45 {
46    *val *= 2;
47 }
```

Program Output with Example Input Shown in Bold

```
Enter an integer number: 10[Enter]
That value doubled is 20
```

Program 10-11 has two functions that use pointers as parameters. Notice the function prototypes:

```
void getNumber(int *);
void doubleValue(int *);
```

Each one uses the notation `int *` to indicate the parameter is a pointer to an `int`. As with all other types of parameters, it isn't necessary to specify the name of the variable in the prototype. The `*` is required, though.

The `getNumber` function asks the user to enter an integer value. The following `cin` statement stores the value entered by the user in memory:

```
cin >> *input;
```

The indirection operator causes the value entered by the user to be stored, not in `input`, but in the variable pointed to by `input`.



WARNING! It's critical that the indirection operator be used in the previous statement. Without it, `cin` would store the value entered by the user in `input`, as if the value were an address. If this happens, `input` will no longer point to the `number` variable in function `main`. Subsequent use of the pointer will result in erroneous, if not disastrous, results.

When the `getNumber` function is called, the address of the `number` variable in function `main` is passed as the argument. After the function executes, the value entered by the user is stored in `number`. Next, the `doubleValue` function is called, with the address of `number` passed as the argument. This causes `number` to be multiplied by two.

Pointer variables can also be used to accept array addresses as arguments. Either subscript or pointer notation may then be used to work with the contents of the array. This is demonstrated in Program 10-12.

Program 10-12

```
1  // This program demonstrates that a pointer may be used as a
2  // parameter to accept the address of an array. Either subscript
3  // or pointer notation may be used.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  // Function prototypes
9  void getSales(double *, int);
10 double totalSales(double *, int);
11
12 int main()
13 {
14   const int QUARTERS = 4;
15   double sales[QUARTERS];
16
```

(program continues)

Program 10-12 *(continued)*

```
17   getSales(sales, QUARTERS);
18   cout << setprecision(2);
19   cout << fixed << showpoint;
20   cout << "The total sales for the year are $";
21   cout << totalSales(sales, QUARTERS) << endl;
22   return 0;
23 }
24
25 //*************************************************************
26 // Definition of getSales. This function uses a pointer to accept *
27 // the address of an array of doubles. The number of elements in  *
28 // in the array is passed as a separate integer parameter. The     *
29 // The function asks the  user to enter the sales figures for      *
30 // four quarters, then stores those figures in the array using     *
31 // subscript notation.                                             *
32 //*************************************************************
33 void getSales(double *array, int size)
34 {
35   for (int count = 0; count < size; count++)
36   {
37     cout << "Enter the sales figure for quarter ";
38     cout << (count + 1) << ": ";
39     cin >> array[count];
40   }
41 }
42
43 //*************************************************************
44 // Definition of totalSales. This function uses a pointer to    *
45 // accept the address of an array of doubles whose size is      *
46 // is passed as a separate parameter. The function uses pointer *
47 // notation to sum the elements of the array.                   *
48 //*************************************************************
49 double totalSales(double *array, int size)
50 {
51    double sum = 0.0;
52
53    for (int count = 0; count < size; count++)
54    {
55      sum += *array;
56      array++;
57    }
58    return sum;
59 }
```

Program Output with Example Input Shown in Bold

```
Enter the sales figure for quarter 1: 10263.98[Enter]
Enter the sales figure for quarter 2: 12369.69[Enter]
Enter the sales figure for quarter 3: 11542.13[Enter]
Enter the sales figure for quarter 4: 14792.06[Enter]
The total sales for the year are $48967.86
```

Notice that in the `getSales` function in Program 10-12, even though the parameter `array` is defined as a pointer, subscript notation is used in the `cin` statement:

```
cin >> array[count];
```

In the `totalSales` function, `array` is used with the indirection operator in the following statement:

```
sum += *array;
```

And in the next statement, the address in `array` is incremented to point to the next element:

```
array++;
```



NOTE: The two previous statements could be combined into the following statement:

```
sum += *array++;
```

The * operator will first dereference `array`, then the ++ operator will increment the address in `array`.

10.8 Pointers to Constants and Constant Pointers

CONCEPT: **A pointer to a constant may not be used to change the value it points to; a constant pointer may not be changed after it has been initialized.**

Pointers to Constants

You have seen how an item's address can be passed into a pointer parameter, and the pointer can be used to modify the item that was passed as an argument. Sometimes it is necessary to pass the address of a `const` item into a pointer. When this is the case, the pointer must be defined as a pointer to a `const` item. For example, consider the following array definition:

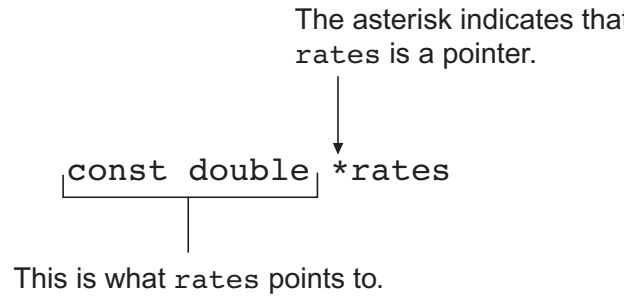
```
const int SIZE = 6;
const double payRates[SIZE] = { 18.55, 17.45,
                                12.85, 14.97,
                                10.35, 18.89 };
```

In this code, `payRates` is an array of `const doubles`. This means that each element in the array is a `const double`, and the compiler will not allow us to write code that changes the array's contents. If we want to pass the `payRates` array into a pointer parameter, the parameter must be declared as a pointer to `const double`. The following function shows such an example:

```
void displayPayRates(const double *rates, int size)
{
   // Set numeric output formatting.
   cout << setprecision(2) << fixed << showpoint;

   // Display all the pay rates.
   for (int count = 0; count < size; count++)
   {
      cout << "Pay rate for employee " << (count + 1)
           << " is $" << *(rates + count) << endl;
   }
}
```

In the function header, notice that the `rates` parameter is defined as a pointer to `const double`. It should be noted that the word `const` is applied to the thing that rates points to, not `rates` itself. This is illustrated in Figure 10-6.

Figure 10-6



Because `rates` is a pointer to a `const`, the compiler will not allow us to write code that changes the thing that `rates` points to.

When passing the address of a constant into a pointer variable, the variable must be defined as a pointer to a constant. If the word `const` has been left out of the definition of the `rates` parameter, a compiler error would have resulted.

Passing a Non-Constant Argument into a Pointer to a Constant

Although a constant's address can be passed only to a pointer to `const`, a pointer to `const` can also receive the address of a non-constant item. For example, look at Program 10-13.

Program 10-13

```
1  // This program demonstrates a pointer to const parameter
2  #include <iostream>
3  using namespace std;
4
5  void displayValues(const int *, int);
6
7  int main()
8  {
9     // Array sizes
10    const int SIZE = 6;
11
12    // Define an array of const ints.
13    const int array1[SIZE] = { 1, 2, 3, 4, 5, 6 };
14
15    // Define an array of non-const ints.
16    int array2[SIZE] = { 2, 4, 6, 8, 10, 12 };
17
18    // Display the contents of the const array.
19    displayValues(array1, SIZE);
20
```

(program continues)

Program 10-13 *(continued)*

```
21    // Display the contents of the non-const array.
22    displayValues(array2, SIZE);
23    return 0;
24 }
25
26 //************************************************
27 // The displayValues function uses a pointer to     *
28 // parameter to display the contents of an array.   *
29 //************************************************
30
31 void displayValues(const int *numbers, int size)
32 {
33    // Display all the values.
34    for (int count = 0; count < size; count++)
35    {
36       cout << *(numbers + count) << " ";
37    }
38    cout << endl;
39 }
```

Program Output

```
1 2 3 4 5 6
2 4 6 8 10 12
```



NOTE: When writing a function that uses a pointer parameter, and the function is not intended to change the data the parameter points to, it is always a good idea to make the parameter a pointer to `const`. Not only will this protect you from writing code in the function that accidentally changes the argument, but the function will be able to accept the addresses of both constant and non-constant arguments.

Constant Pointers

In the previous section we discussed pointers to `const`. That is, pointers that point to `const` data. You can also use the `const` key word to define a constant pointer. Here is the difference between a pointer to `const` and a `const` pointer:

- A pointer to `const` points to a constant item. The data that the pointer points to cannot change, but the pointer itself can change.
- With a `const` pointer, it is the pointer itself that is constant. Once the pointer is initialized with an address, it cannot point to anything else.

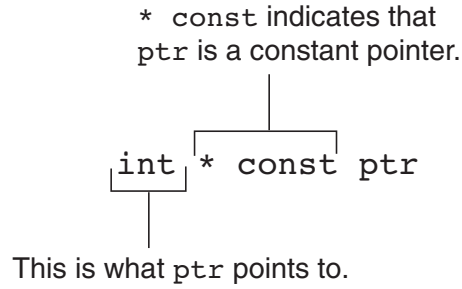
The following code shows an example of a `const` pointer.

```
int value = 22;
int * const ptr = &value;
```

Notice in the definition of `ptr` the word `const` appears after the asterisk. This means that `ptr` is a `const` pointer. This is illustrated in Figure 10-7. In the code, `ptr` is initialized with the address of the `value` variable. Because `ptr` is a constant pointer, a compiler error will

result if we write code that makes `ptr` point to anything else. An error will not result, however, if we use `ptr` to change the contents of `value`. This is because `value` is not constant, and `ptr` is not a pointer to `const`.

Figure 10-7



Constant pointers must be initialized with a starting value, as shown in the previous example code. If a constant pointer is used as a function parameter, the parameter will be initialized with the address that is passed as an argument into it, and cannot be changed to point to anything else while the function is executing. Here is an example that attempts to violate this rule:

```
void setToZero(int * const ptr)
{
   ptr = 0;  // ERROR!! Cannot change the contents of ptr.
}
```

This function's parameter, `ptr`, is a `const` pointer. It will not compile because we cannot have code in the function that changes the contents of `ptr`. However, `ptr` does not point to a `const`, so we can have code that changes the data that `ptr` points to. Here is an example of the function that will compile:

```
void setToZero(int * const ptr)
{
   *ptr = 0;
}
```

Although the parameter is `const` pointer, we can call the function multiple times with different arguments. The following code will successfully pass the addresses of `x`, `y`, and `z` to the `setToZero` function:

```
int x, y, z;
// Set x, y, and z to 0.
setToZero(&x);
setToZero(&y);
setToZero(&z);
```

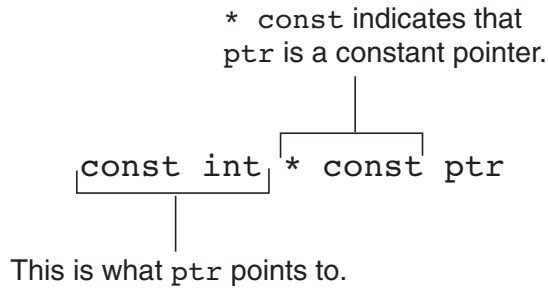
Constant Pointers to Constants

So far, when using `const` with pointers we've seen pointers to constants and we've seen constant pointers. You can also have constant pointers to constants. For example, look at the following code:

```
int value = 22;
const int * const ptr = &value;
```

In this code `ptr` is a `const` pointer to a `const int`. Notice the word `const` appears before `int`, indicating that `ptr` points to a `const int`, and it appears after the asterisk, indicating that `ptr` is a constant pointer. This is illustrated in Figure 10-8.

Figure 10-8



In the code, `ptr` is initialized with the address of `value`. Because `ptr` is a `const` pointer, we cannot write code that makes `ptr` point to anything else. Because `ptr` is a pointer to `const`, we cannot use it to change the contents of `value`. The following code shows one more example of a `const` pointer to a `const`. This is another version of the `displayValues` function in Program 10-13.

```
void displayValues(const int * const numbers, int size)
{
   // Display all the values.
   for (int count = 0; count < size; count++)
   {
      cout << *(numbers + count) << " ";
   }
   cout << endl;
}
```

In this code, the parameter `numbers` is a `const` pointer to a `const int`. Although we can call the function with different arguments, the function itself cannot change what `numbers` points to, and it cannot use `numbers` to change the contents of an argument.

10.9 Focus on Software Engineering: *Dynamic Memory Allocation*

CONCEPT: Variables may be created and destroyed while a program is running.

As long as you know how many variables you will need during the execution of a program, you can define those variables up front. For example, a program to calculate the area of a rectangle will need three variables: one for the rectangle's length, one for the rectangle's width, and one to hold the area. If you are writing a program to compute the

payroll for 30 employees, you'll probably create an array of 30 elements to hold the amount of pay for each person.

But what about those times when you don't know how many variables you need? For instance, suppose you want to write a test-averaging program that will average any number of tests. Obviously the program would be very versatile, but how do you store the individual test scores in memory if you don't know how many variables to define? Quite simply, you allow the program to create its own variables "on the fly." This is called *dynamic memory allocation* and is only possible through the use of pointers.

To dynamically allocate memory means that a program, while running, asks the computer to set aside a chunk of unused memory large enough to hold a variable of a specific data type. Let's say a program needs to create an integer variable. It will make a request to the computer that it allocate enough bytes to store an `int`. When the computer fills this request, it finds and sets aside a chunk of unused memory large enough for the variable. It then gives the program the starting address of the chunk of memory. The program can only access the newly allocated memory through its address, so a pointer is required to use those bytes.

The way a C++ program requests dynamically allocated memory is through the `new` operator. Assume a program has a pointer to an `int` defined as

```
int *iptr;
```

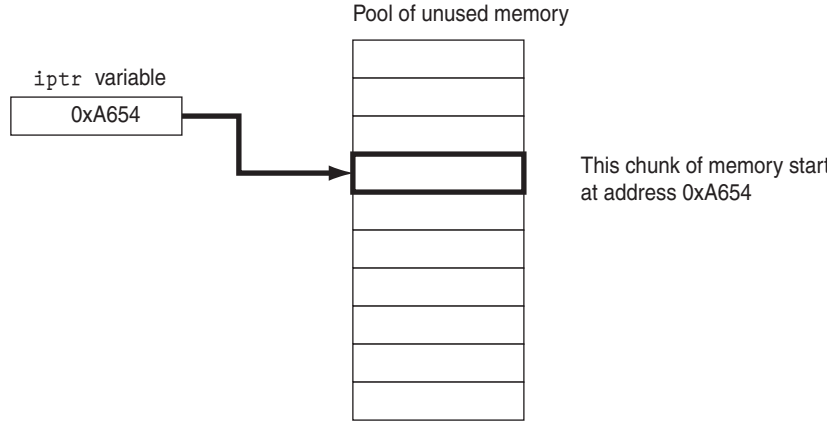
Here is an example of how this pointer may be used with the `new` operator:

```
iptr = new int;
```

This statement is requesting that the computer allocate enough memory for a new `int` variable. The operand of the new operator is the data type of the variable being created. This is illustrated in Figure 10-9. Once the statement executes, `iptr` will contain the address of the newly allocated memory. A value may be stored in this new variable by dereferencing the pointer:

```
*iptr = 25;
```

Figure 10-9



Any other operation may be performed on the new variable by simply using the dereferenced pointer. Here are some example statements:

```
cout << *iptr;       // Display the contents of the new variable.
cin >> *iptr;        // Let the user input a value.
total += *iptr;      // Use the new variable in a computation.
```

Although these statements illustrate the use of the `new` operator, there's little purpose in dynamically allocating a single variable. A more practical use of the `new` operator is to dynamically create an array. Here is an example of how a 100-element array of integers may be allocated:

```
iptr = new int[100];
```



VideoNote
Dynamically Allocating an Array

Once the array is created, the pointer may be used with subscript notation to access it. For instance, the following loop could be used to store the value 1 in each element:

```
for (int count = 0; count < 100; count++)
   iptr[count] = 1;
```

Every call to `new` allocates storage from a special area of the program's memory called the *heap*. If a program makes a lot of requests for dynamic memory, the heap will eventually become depleted and additional calls to `new` will fail. When this happens, the C++ runtime system will *throw* a `bad_alloc` exception to notify the calling program that the requested memory cannot be allocated. An *exception* is a mechanism for notifying a program that something has gone drastically wrong with the operation that was being executed and that the results of the operation cannot be trusted. The default action of an exception is to force the executing program to terminate.

Older C++ compilers do not throw `bad_alloc`. Instead, they indicate memory-allocation failure by returning the address 0 (`NULL`) from the call to `new`. Memory-allocation code written for these compilers should check the returned address to make sure it is not 0 before using it, as in the following code fragment:

```
// For older versions of C++
iptr = new int[100];
if (iptr == NULL)
{
   cout << "Memory allocation error.";
   exit(1);
}
// Rest of code to use the allocated array
```

A program that has finished using a dynamically allocated block of memory should *free* the memory and return it to the heap to make it available for future use. This is accomplished by calling the `delete` operator and passing it the address of the memory to be deallocated. Here is an example of how `delete` is used to free up a single variable pointed to by `iptr`:

```
delete iptr;
iptr = 0;
```

If `iptr` points to a dynamically allocated array, a pair of square brackets must be placed between `delete` and `iptr`:

```
delete [ ] iptr;
iptr = 0;
```

Dangling Pointers and Memory Leaks

A pointer is said to be *dangling* if it is pointing to a memory location that has been freed by a call to `delete`. When you access a dangling pointer, you are trying to use memory that has already been freed and returned to the heap. In fact, such memory may already be reallocated by another call to `new`. The use of dangling pointers can cause errors in your program that are difficult to trace. You can avoid the use of dangling pointers by setting pointers to 0 as soon as they are freed, as shown in the examples above.

A *memory leak* is said to occur in your program if after you have finished using a block of memory allocated by `new`, you forget to free it via `delete`. The leaked block of memory remains unavailable for use until the program terminates. Memory leaks are especially serious when they occur in loops. They are even more serious in programs such as Web servers and other network programs that are expected to run for months or even years without being shut down. Over time, a Web server with a memory leak will exhaust all memory in the computer on which it is running, requiring both it and the computer to be shut down and restarted.



WARNING! Only use pointers with `delete` that were previously used with `new`. If you use a pointer with `delete` that does not reference dynamically allocated memory, unexpected problems could result!

Program 10-14 demonstrates the use of `new` and `delete`. It asks for sales figures for any number of days. The figures are stored in a dynamically allocated array, and then totaled and averaged.

Program 10-14

```
1  // This program totals and averages the sales figures for
2  // any number of days. The figures are stored in a
3  // dynamically allocated array.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  int main()
9  {
10    double *sales,        // To dynamically allocate an array
11           total = 0.0,   // Accumulator
12           average;       // To hold average sales
13    int numDays;          // To hold number of days of sales
14
15    // Get number of days of sales.
16    cout << "How many days of sales figures do you wish ";
17    cout << "to process? ";
18    cin >> numDays;
19
20    // Dynamically allocate an array large enough
21    // to hold that many days of sales amounts.
22    sales = new double[numDays];  // Allocate memory
23
```

(program continues)

Program 10-14 *(continued)*

```
24    // Get the sales figures for each day.
25    cout << "Enter the sales figures below.\n";
26    for (int count = 0; count < numDays; count++)
27    {
28       cout << "Day " << (count + 1) << ": ";
29       cin >> sales[count];
30    }
31
32    // Calculate the total sales.
33    for (int count = 0; count < numDays; count++)
34    {
35       total += sales[count];
36    }
37
38    // Calculate the average sales per day.
39    average = total / numDays;
40
41    // Display the results.
42    cout << setprecision(2) << fixed << showpoint;
43    cout << "\n\nTotal Sales: $" << total << endl;
44    cout << "Average Sales: $" << average << endl;
45
46    // Free dynamically allocated memory.
47    delete [] sales;
48    sales = 0;
49    return 0;
50 }
```

Program Output with Example Input Shown in Bold

```
How many days of sales figures do you wish to process? 5[Enter]
Enter the sales figures below.
Day 1: 898.63[Enter]
Day 2: 652.32[Enter]
Day 3: 741.85[Enter]
Day 4: 852.96[Enter]
Day 5: 921.37[Enter]

Total Sales: $4067.13
Average Sales: $813.43
```

The statement in line 23 dynamically allocates memory for an array of `doubles`, using the value in `numDays` as the number of elements. The `new` operator returns the starting address of the memory allocated, and this address is assigned to the `sales` pointer variable. The `sales` variable is then used throughout the program to store the sales amounts in the array and perform the necessary calculations. In line 48 the `delete` operator is used to free the allocated memory.

Notice that in line 49 the value 0 is assigned to the `sales` pointer. It is good practice to store 0 in a pointer variable after using `delete` on it. First, it prevents code from inadvertently using the pointer to access the area of memory that was freed. Second, it prevents

errors from occurring if `delete` is accidentally called on the pointer again. The `delete` operator is designed to have no effect when used on a null pointer.

10.10 Focus on Software Engineering: *Returning Pointers from Functions*

CONCEPT: Functions can return pointers, but you must be sure the item the pointer references still exists.

A function may return a pointer to any data type. For example, the following function locates the null terminator in a string and returns a pointer to it.

```
char *findNull(char *str)
{
   char *ptr = str;

   while (*ptr != '\0')
          ptr++;
   return ptr;
}
```

The `char *` return type in the function header indicates the function returns a pointer to a `char`:

```
char *findNull(char *str)
```

When writing functions that return pointers, however, you should take care not to create elusive bugs. For instance, see if you can determine what's wrong with the following function.

```
char *getName()
{
   char name[81];
   cout << "Enter your name: ";
   cin.getline(name, 81);
   return name;
}
```

The problem, of course, is that the function returns a pointer to an array that no longer exists. Because `name` is local to the function, it is destroyed when the function returns. Attempting to use the pointer will result in erroneous and unpredictable results.

You should only return a pointer from a function if it is

- A pointer to an item that was passed into the function as an argument
- A pointer to dynamically allocated memory

For instance, the following function is acceptable:

```
char *getName(char *name)
{
```

```
   cout << "Enter your name: ";
   cin.getline(name, 81);
   return name;
}
```

This function accepts a pointer to the memory location where the user's input is to be stored. Since the pointer references a memory location that was valid prior to the function being called, it is safe to return a pointer to the same location. Here is another acceptable function:

```
char *getName()
{
   char *name;

   name = new char[81];
   cout << "Enter your name: ";
   cin.getline(name, 81);
   return name;
}
```

This function uses the `new` operator to allocate a section of memory. This memory will remain allocated until the `delete` operator is used or the program ends, so it's safe to return a pointer to it.

Program 10-15 shows another example. This program uses a function, `getRandomNumbers`, to get a pointer to an array of random numbers. The function accepts an integer argument that is the number of random numbers in the array. The function dynamically allocates an array, uses the system clock to seed the random number generator, populates the array with random values, and then returns a pointer to the array.

Program 10-15

```
1  // This program demonstrates a function that returns
2  // a pointer.
3  #include <iostream>
4  #include <cstdlib>   // For rand and srand
5  #include <ctime>     // For the time function
6  using namespace std;
7
8  // Function prototype
9  int *getRandomNumbers(int);
10
11 int main()
12 {
13    int *numbers;  // To point to the numbers
14
15    // Get an array of five random numbers.
16    numbers = getRandomNumbers(5);
17
18    // Display the numbers.
19    for (int count = 0; count < 5; count++)
20       cout << numbers[count] << endl;
21
```

(program continues)

Program 10-15 *(continued)*

```
22    // Free the memory.
23    delete [] numbers;
24    numbers = 0;
25    return 0;
26 }
27
28 //************************************************
29 // The getRandomNumbers function returns a pointer *
30 // to an array of random integers. The parameter   *
31 // indicates the number of numbers requested.      *
32 //************************************************
33
34 int *getRandomNumbers(int num)
35 {
36    int *array;    // Array to hold the numbers
37
38    // Return null if num is zero or negative.
39    if (num <= 0)
40       return NULL;
41
42    // Dynamically allocate the array.
43    array = new int[num];
44
45    // Seed the random number generator by passing
46    // the return value of time(0) to srand.
47    srand( time(0) );
48
49    // Populate the array with random numbers.
50    for (int count = 0; count < num; count++)
51       array[count] = rand();
52
53    // Return a pointer to the array.
54    return array;
55 }
```

Program Output with Example Input Shown in Bold

```
2712
9656
24493
12483
7633
```



Checkpoint

10.9 Assuming `array` is an array of `int`s, which of the following program segments will display "True" and which will display "False"?

A)
```
if (array < &array[1])
   cout << "True";
else
   cout << "False";
```

B)
```
if (&array[4] < &array[1])
   cout << "True";
else
   cout << "False";
```
C)
```
if (array != &array[2])
   cout << "True";
else
   cout << "False";
```
D)
```
if (array != &array[0])
   cout << "True";
else
   cout << "False";
```

10.10 Give an example of the proper way to call the following function in order to negate the variable `int num = 7;`

```
void makeNegative(int *val)
{
   if (*val > 0)
       *val = -(*val);
}
```

10.11 Complete the following program skeleton. When finished, the program should ask the user for a length (in inches), convert that value to centimeters, and display the result. You are to write the function `convert`. (Note: 1 inch = 2.54 cm. Do not modify function `main`.)

```
#include <iostream>
#include <iomanip>
using namespace std;

// Write your function prototype here.

int main()
{
   double measurement;

   cout << "Enter a length in inches, and I will convert\n";
   cout << "it to centimeters: ";
   cin >> measurement;
   convert(&measurement);
   cout << setprecision(4);
   cout << fixed << showpoint;
   cout << "Value in centimeters: " << measurement << endl;
   return 0;
}
//
// Write the function convert here.
//
```

10.12 Look at the following array definition:

```
const int numbers[SIZE] = { 18, 17, 12, 14 };
```

Suppose we want to pass the array to the function `processArray` in the following manner:

```
processArray(numbers, SIZE);
```

Which of the following function headers is the correct one for the `processArray` function?

A) `void processArray(const int *array, int size)`

B) `void processArray(int * const array, int size)`

10.13 Assume `ip` is a pointer to an `int`. Write a statement that will dynamically allocate an integer variable and store its address in `ip`, then write a statement that will free the memory allocated in the statement you just wrote.

10.14 Assume `ip` is a pointer to an `int`. Write a statement that will dynamically allocate an array of 500 integers and store its address in `ip`, then write a statement that will free the memory allocated in the statement you just wrote.

10.15 What is a null pointer?

10.16 Give an example of a function that correctly returns a pointer.

10.17 Give an example of a function that incorrectly returns a pointer.

10.11 Pointers to Class Objects and Structures

CONCEPT: Pointers and dynamic memory allocation can be used with class objects and structures.

Declaring a pointer to a class is the same as declaring any other pointer type. For example, if `Rectangle` is defined as

```
class Rectangle
{
   int width, height;
};
```

you can declare a pointer to `Rectangle` and create a `Rectangle` object by writing

```
Rectangle *pRect; // Pointer to Rectangle
Rectangle rect;   // Rectangle object
```

and you can assign the address of `rect` to `pRect` as follows:

```
pRect = &rect;
```

Now suppose that you want to access the members of the `Rectangle` object through the pointer `pRect`. Because `*pRect` is just another way of accessing `rect`, you might think that the expression

```
*pRect.width
```

will access `rect.width`, but this is not so. The reason is that the dot selector has higher priority than the `*` operator, so `*pRect.width` is equivalent to `*(pRrect.width)`. This last expression is a type error. To get it right, you must use parentheses to force the indirection operator `*` to be applied first, as shown here:

```
(*pRect).width
```

The following statements will correctly set the dimensions of the rectangle to 10 and 20.

```
(*pRect).width = 10;
(*pRect).height = 20;
```

The combined use of parentheses, the indirection operator, and the dot selector to access members of class objects via pointers can result in expressions that are hard to read. To solve this problem, C++ provides the *structure pointer operator* -> to use when you want to access a member of a class object through a pointer. It consists of a hyphen – and a greater-than symbol > written next to each other to look like an arrow. Using this operator, you can set the dimensions of the rectangle with these statements:

```
pRect->width = 10;
pRect->height = 20;
```

Member functions of class objects can be called through a pointer. In particular, if `ptr` is a pointer to an object that has a member function `fun()`, then the function can be called with either one of these two (equivalent) expressions:

```
(*ptr).fun();
ptr->fun();
```

Dynamic Allocation of Class Objects

Dynamically allocated class objects are used in programs that build and manage advanced data structures such as lists (studied in Chapter 17) and binary trees (studied in Chapter 19). The `new` operator is used to allocate such objects in the same way that it is used to allocate variables of other types. For example, the following statements allocate a single `Rectangle` object and set its dimensions

```
pRect = new Rectangle;
pRect->width = 10;
pRect->height = 3;
```

If `Rectangle` has a constructor that takes two integer parameters, then you can simultaneously allocate the object and invoke the constructor like this:

```
pRect = new Rectangle(10, 30);
```

Program 10-16 illustrates these concepts.

Program 10-16

```
1 // This program uses pointers to dynamically allocate
2 // structures and class objects.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
```

(program continues)

Program 10-16 *(continued)*

```
// Person class
class Person
{
private:
    string name;
    int  age;
public:
    Person(string name1, int age1)
    {
        name = name1;
        age = age1;
    }
    int getAge() { return age; }
    string getName(){ return name; }
};

// Rectangle structure
struct Rectangle
{
    int width, height;
};

int main()
{
    Rectangle *pRect;    // Pointer to Rectangle
    Person *pPerson;     // Pointer to Person

    // Create a rectangle object and access it through a pointer
    Rectangle rect;
    pRect = &rect;
    (*pRect).height = 12;
    pRect->width = 10;
    cout << "Area of the first rectangle is "
         << pRect->width * pRect->height;

    // Allocate a Rectangle object and access it through a pointer
    pRect = new Rectangle;
    pRect->height = 6;
    pRect->width = 5;
    cout << "\nArea of the second rectangle is "
        << pRect->width * pRect->height;

    // Allocate a Person object and call its methods through a pointer
    pPerson = new Person("Miguel E. Gonzalez", 23);
    cout << "\n\nThe person's name is " << pPerson->getName();
    cout << "\nThe person's age is " << pPerson->getAge() << endl;

    return 0;
}
```

(program continues)

Program 10-16 *(continued)*

Program Output

```
Area of the first rectangle is 120
Area of the second rectangle is 30

The person's name is Miguel E. Gonzalez
The person's age is 23
```

Pointers to Class Objects as Function Parameters

Pointers to structures and class variables can be passed to functions as parameters. The function receiving the pointer can then use it to access or modify members of the structure. This is shown in Program 10-17.

Program 10-17

```
1  // This program illustrates pointers to class objects
2  // and structures as parameters of functions.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  // Person class
8  class Person
9  {
10 private:
11     string name;
12     int  age;
13 public:
14     Person(string name1, int age1)
15     {
16         name = name1;
17         age = age1;
18     }
19     int getAge() { return age; }
20     string getName(){ return name; }
21 };
22
23 // Rectangle structure
24 struct Rectangle
25 {
26     int width, height;
27 };
28
29 // Prototypes
30 void magnify(Rectangle *pRect, int mfactor);
31 int lengthOfName(Person *p);
32 void output(Rectangle *pRect);
33
```

(program continues)

Program 10-17 *(continued)*

```
34 int main()
35 {
36     // Create, then magnify a Rectangle by a factor of 3
37     Rectangle rect;
38     rect.width = 4;
39     rect.height = 2;
40     cout << "Initial size of  rectangle is ";
41     output(&rect);
42     magnify(&rect, 3);
43     cout << "Size of Rectangle after magnification is ";
44     output(&rect);
45
46     // Create a Person and find the length of the person's name
47     Person *pPerson = new Person("Susan Wu", 32);
48     cout << "The name " << pPerson->getName()
49          << " has length " << lengthOfName(pPerson) << endl;
50
51     return 0;
52 }
53
54 //*******************************************
55 // Output the dimensions of a rectangle       *
56 //*******************************************
57 void output(Rectangle *pRect)
58 {
59     cout << "width: " << pRect->width << " height: "
60          << pRect->height << endl;
61 }
62
63 //*****************************************************
64 // Returns the number of characters in a person's name *
65 //*****************************************************
66 int lengthOfName(Person *p)
67 {
68     string name = p->getName();
69     return name.length();
70 }
71
72 //*****************************************************
73 // Stretch the width and height of a rectangle by      *
74 // a specified factor                                  *
75 //*****************************************************
76 void magnify(Rectangle *pRect, int factor)
77 {
78     pRect->width = pRect->width * factor;
79     pRect->height = pRect->height * factor;
80 }
81
```

(program continues)

Program 10-17 *(continued)*

Program Output

```
Initial size of rectangle is width: 4 height: 2
Size of Rectangle after magnification is width: 12 height: 6
The name Susan Wu has length 8
```

Stopping Memory Leaks

It is important for programs that use dynamically allocated memory to ensure that each call to `new` is eventually followed by a call to `delete` that frees the allocated memory and returns it to the heap. A program that fails to do this will suffer from *memory leaks*, a condition in which the program loses track of dynamically allocated storage and therefore never calls `delete` to free the memory. There are two rules of the thumb that can be used to avoid memory leaks:

- The function that invokes `new` to allocate storage should also be the function that invokes `delete` to deallocate the storage.
- A class that needs to dynamically allocate storage should invoke `new` in its constructors and invoke the corresponding `delete` in its destructor. Because the destructor is automatically called by the system whenever an object is deleted or goes out of scope, a `delete` statement placed in a destructor will always be called.

By following these rules, you will always be able to find the `delete` operation that corresponds to a given call to `new`, thereby verifying that a particular call to `new` does not result in a memory leak. Program 10-18 is an example of a program that follows these rules. Note that the `Squares` class allocates dynamic memory in its constructor and has a `delete` statement in its destructor. The program is garnished with output statements in strategic places to show when the `new` and `delete` operators in constructors and destructors are called.

Program 10-18

```
1  // This program illustrates the use of constructors
2  // and destructors in the allocation and deallocation of memory.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  class Squares
8  {
9  private:
10     int length;  // How long is the sequence
11     int *sq;     // Dynamically allocated array
12 public:
13     // Constructor allocates storage for sequence
14     // of squares and creates the sequence
15     Squares(int len)
```

(program continues)

Program 10-18 *(continued)*

```
16      {
17          length = len;
18          sq = new int[length];
19          for (int k = 0; k < length; k++)
20          {
21              sq[k] = (k+1)*(k+1);
22          }
23          // Trace
24          cout << "Construct an object of size " << length << endl;
25      }
26      // Print the sequence
27      void print()
28      {
29          for (int k = 0; k < length; k++)
30              cout << sq[k] << "  ";
31          cout << endl;
32      }
33      // Destructor deallocates storage
34      ~Squares()
35      {
36          delete [ ] sq;
37          // Trace
38          cout << "Destructor for object of size " << length <<  endl;
39      }
40  };
41
42  //*********************************************
43  // Outputs the sequence of squares in a         *
44  // Squares object                               *
45  //*********************************************
46  void outputSquares(Squares *sqPtr)
47  {
48      cout << "The list of squares is: ";
49      sqPtr->print();
50
51  }
52  int main()
53  {
54      Squares sqs(5);
55      cout << "The first 5 squares are: ";
56      sqs.print();
57
58      // Main allocates a Squares object
59      Squares *sqPtr = new Squares(3);
60      outputSquares(sqPtr);
61      // Main deallocates the Squares object
62      delete sqPtr;
63
64      return 0;
65  }
```

(program continues)

Program 10-18 *(continued)*

Program Output

```
Construct an object of size 5
The first 5 squares are: 1  4  9  16  25
Construct an object of size 3
The list of squares is: 1  4  9
Destructor for object of size 3
Destructor for object of size 5
```

Everything we have said in this section applies to structures as well.

10.12 Focus on Software Engineering: *Selecting Members of Objects*

Sometimes structures and classes contain pointers as members. For example, the following structure declaration has an `int` pointer member:

```
struct GradeInfo
{
   char name[25];  // Student name
   int *testScores;// Dynamically allocated array
   double average; // Test average
};
```

It's important to remember that the structure pointer operator (`->`) is used to dereference a pointer to a structure or class object, not a pointer that is a member of a structure or class. If a program dereferences the `testScores` pointer in the structure in the example, the indirection operator must be used. For example, assuming the following variable has been defined:

```
GradeInfo student1;
```

The following statement will display the value pointed to by the `testScores` member:

```
cout << *student1.testScores;
```

It's still possible to define a pointer to a structure that contains a pointer member. For instance, the following statement defines `stPtr` as a pointer to a `GradeInfo` structure:

```
GradeInfo *stPtr;
```

Assuming `stPtr` points to a valid `GradeInfo` variable, the following statement will display the value pointed to by its `testScores` member:

```
cout << *stPtr->testScores;
```

In this statement, the `*` operator dereferences `stPtr->testScores`, while the `->` operator dereferences `stPtr`. It might help to remember that the expression

```
stPtr->testScores
```

is equivalent to

```
(*stPtr).testScores
```

So, the expression

```
*stPtr->testScores
```

is the same as

```
*(*stPtr).testScores
```

The awkwardness of this expression shows the necessity of the `->` operator. Table 10-1 lists some expressions using the `*`, `->`, and `.` operators, and describes what each references. The table is easier to understand if you remember that the operators `->` and `.` for selecting members of structures have higher precedence than the dereferencing operator `*`.

Table 10-1 Dereferencing Pointers to Structures

Expression	Description
`s->m`	`s` is a pointer to a structure variable or class object, and `m` is a member. This expression accesses the `m` member of the structure or class object pointed to by `s`.
`*a.p`	`a` is a structure variable or class object and `p`, a pointer, is a member of `a`. This expression accesses the value pointed to by `a.p`.
`(*s).m`	`s` is a pointer to a structure variable or class object, and `m` is a member. The `*` operator dereferences `s`, causing the expression to access the `m` member of the object `*s`. This expression is the same as `s->m`.
`*s->p`	`s` is a pointer to a structure variable or class object and `p`, a pointer, is a member of the object pointed to by `s`. This expression accesses the value pointed to by `s->p`.
`*(*s).p`	`s` is a pointer to a structure variable or class object and `p`, a pointer, is a member of the object pointed to by `s`. This expression accesses the value pointed to by `(*s).p`. This expression is the same as `*s->p`.



Checkpoint

Assume the following structure declaration exists for questions 10.17 through 10.19:

```
struct Rectangle
{
   int length;
   int width;
};
```

10.18 Write the definition of a pointer to a `Rectangle` structure.

10.19 Assume the pointer you defined in question 10.17 points to a valid `Rectangle` structure. Write the statement that displays the structure's members through the pointer.

10.20 Assume `rptr` is a pointer to a `Rectangle` structure. Which of the expressions, A, B, or C, is equivalent to the expression:

```
rptr->width
```

A) `*rptr.width`
B) `(*rptr).width`
C) `rptr.(*width)`

10.13 United Cause Relief Agency Case Study

CONCEPT: This case study demonstrates how an array of pointers can be used to display the contents of a second array in sorted order, without sorting the second array.

The United Cause, a charitable relief agency, solicits donations from businesses. The local United Cause office received the following donations from the employees of CK Graphics, Inc:

$5, $100, $5, $25, $10, $5, $25, $5, $5, $100, $10, $15, $10, $5, $10

The donations were received in the order they appear. The United Cause manager has asked you to write a program that displays the donations in ascending order, as well as in their original order.

You decide to create a class, `DonationList`, that will hold and process the donation data. The class declaration is

```
class DonationList
{
private:
      int numDonations;
      double *donations;
      double **arrPtr;
      void selectSort();
public:
      DonationList(int num, double gifts[]);
      ~DonationList();
      void show();
      void showSorted();
};
```

Table 10-2 lists and describes the class's member variables

Table 10-2 Description of Member Variables

Member Variable	Description
`numDonations`	An integer that will hold the number of donations received. This value will be used to dynamically allocate arrays for holding and processing the donation values.
`donations`	A pointer that will point to a dynamically allocated array of `doubles` containing the donation amounts.
`arrPtr`	A pointer that will point to an array of pointers. The array of pointers will be dynamically allocated. Each element of the array will point to an element of the `donations` array.

In this class, the donation values will be stored in their original order in a dynamically allocated array of `doubles`. The `donations` member will point to the array. We will refer to this array as the `donations` array. The following statement shows how the `donations` member will be defined.

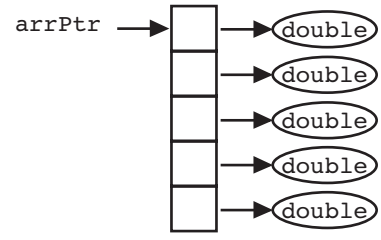
```
double *donations;
```

The `arrPtr` member will also point to a dynamically allocated array. Its array, however, will be an array of pointers. The elements of the array are pointers to `doubles`. The following statement shows how the `arrPtr` member will be declared.

```
double **arrPtr;
```

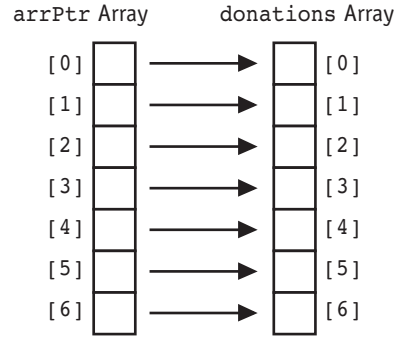
Since the `arrPtr` member will point to an array of pointers, it is a *pointer-to-a-pointer.* That is why two asterisks appear in the declaration statement. The pointer that `arrPtr` points to is a pointer to a `double`. Figure 10-10 illustrates `arrPtr` as a pointer to an array of pointers-to-`doubles`.

Figure 10-10



We will refer to the array of pointers as the `arrPtr` array. Once the `arrPtr` array is allocated in memory, its elements will be initialized so they point to the elements of the `donations` array, as illustrated in Figure 10-11.

Figure 10-11



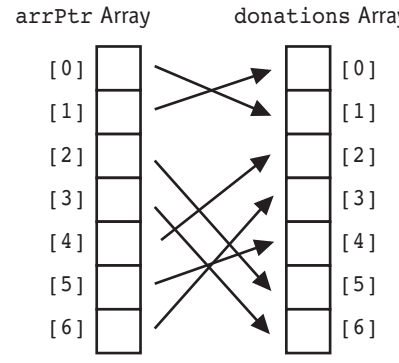
The elements of the `arrPtr` array will initially point to the elements of the `donations` array in their natural order (as shown in Figure 10-11). In other words, `arrPtr[0]` will point to

donations[0], arrPtr[1] will point to donations[1], and so forth. In that arrangement, the following statement would cause the contents of donations[5] to be displayed:

```
cout << *(arrPtr[5]) << endl;
```

After the arrPtr array is sorted, however, arrPtr[0] will point to the smallest value in the donations array, arrPtr[1] will point to the next-to-smallest value in the donations array, and so forth. This is illustrated in Figure 10-12.

Figure 10-12



This technique gives us access to the elements of the donations array in a sorted order without actually disturbing the contents of the donations array itself.

Table 10-3 lists the class's member functions.

Table 10-3 Description of Member Functions

Member Function	Description
Constructor	The constructor accepts as arguments an integer, which indicates the number of donations received, and an array of doubles, which holds the list of donation values. When the constructor is finished, the donations member will point to a dynamically allocated array holding the list of donation values (in their original order), and the arrPtr member will point to a dynamically allocated array of pointers. The elements of the arrPtr array will point, in ascending order, to the elements of the donations array.
Destructor	Deletes the storage that was allocated by the constructor.
selectSort	This function, which performs an ascending-order selection sort on arrPtr, is called by the constructor. Before the sort, arrPtr [0]points to donations [0], arrPtr [1] points to donations [1], and so forth. After the sort, arrPtr will point to the elements of the donations array in ascending order.
show	Displays the contents of the donations array. (This function displays the donations in their original order.)
showSorted	Displays the contents of what each element of the arrPtr array points to. (This function displays the contents of the donations array in sorted order.)

Let's look at each member function in more detail.

The Constructor

The constructor's code is as follows:

```
DonationList::DonationList(int num, double gifts[])
{
   numDonations = num;
   if (num > 0)
   {
      // Allocate an array of doubles.
      donations = new double[num];
      // Allocate an array of pointers-to-doubles.
      arrPtr = new double*[num];
      // Initialize the arrays.
      for (int count = 0; count < numDonations; count++)
      {
         donations[count] = gifts[count];
         arrPtr[count] = &donations[count];
      }
      // Now sort the array of pointers.
      selectSort();
   }
}
```

The `num` argument (which is copied to `numDonations`) holds the number of donations, and the `gifts` array contains the list of donation values.

If the value in `num` is greater than 0, the constructor allocates and initializes the elements of the `donations` and `arrPtr` arrays. The elements of the `gifts` array are copied to the elements of the `donations` array, and the elements of the `arrPtr` array are set to point to the elements of the `donations` array.

The `selectSort` Member Function

The `selectSort` function is a modified version of the selection sort algorithm. The only difference is that this function sorts an array of pointers. Instead of sorting on the contents of the array's elements, the array is sorted on the contents of what its elements point to. Here is the pseudocode.

```
For scan is set to the values 0 up to (but not including) the next-
to-last subscript in arrPtr
   Set minIndex variable to scan
   Set minElem pointer to arrPtr[scan]
   For index variable is set to the values from (scan + 1) through
   the next-to-last subscript in arrPtr
      If *(arrPtr[index]) is less than *minElem
         Set minElem to arrPtr[index]
         Set minIndex to index
      End If
   End For
   Set arrPtr[minIndex] to arrPtr[scan]
   Set arrPtr[scan] to minElem
End For
```

Here is the C++ code for the function:

```
void DonationList::selectSort()
{
   int minIndex;
   double *minElem;

   for (int scan = 0; scan < (numDonations - 1); scan++)
   {
      minIndex = scan;
      minElem = arrPtr[scan];
      for(int index = scan + 1; index < numDonations; index++)
      {
         if (*(arrPtr[index]) < *minElem)
         {
            minElem = arrPtr[index];
            minIndex = index;
         }
      }
      arrPtr[minIndex] = arrPtr[scan];
      arrPtr[scan] = minElem;
   }
}
```

The `show` Member Function

The `show` member function simply displays the contents of the `donations` array sequentially. Here is its pseudocode:

```
For every element in donations Array
   Display the element's contents
End For
```

Here is the function's actual C++ code:

```
void DonationList::show()
{
   for (int count = 0; count < numDonations; count++)
      cout << donations[count] << " ";
   cout << endl;
}
```

The `showSorted` Member Function

The `showSorted` function displays the values pointed to by the elements of the `arrPtr` array. Here is its pseudocode:

```
For every element in the arrPtr array
   Dereference the element and display what it points to
End For
```

Here is the function's C++ code:

```
void DonationList::showSorted()
{
   for (int count = 0; count < numDonations; count++)
      cout << *(arrPtr[count]) << " ";
   cout << endl;
}
```

The Entire `Class` Listing

The class in its entirety is shown here.

Contents of `donlist.h`

```
1  #ifndef DONLIST_H
2  #define DONLIST_H
3
4  class DonationList
5  {
6  private:
7     int numDonations;
8     double *donations;
9     double **arrPtr;
10    void selectSort();
11 public:
12    DonationList(int num, double gifts[]);
13    ~DonationList();
14    void show();
15    void showSorted();
16 };
17 #endif
```

Contents of `donlist.cpp`

```
1  #include <iostream>   //needed for cout
2  #include "donlist.h"
3  using namespace std;
4
5  //*********************************************************
6  // Constructor.                                           *
7  // The argument passed to num indicates the number of     *
8  // elements in array passed to gifts. The gifts array     *
9  // holds the list of donation values. The constructor     *
10 // allocates the donations and arrPtr arrays. The gifts   *
11 // array is copied to the donations array. The elements   *
12 // of the arrPtr array are made to point to the elements  *
13 // of the donations array, and then sorted in ascending   *
14 // order by the selectSort function.                      *
15 //*********************************************************
16 DonationList::DonationList(int num, double gifts[])
17 {
18    numDonations = num;
19    if (num > 0)
20    {
21       // Allocate an array of doubles.
```

```
22       donations = new double[num];
23       // Allocate an array of pointers-to-doubles.
24       arrPtr = new double*[num];
25       // Initialize the arrays.
26       for (int count = 0; count < numDonations; count++)
27       {
28          donations[count] = gifts[count];
29          arrPtr[count] = &donations[count];
30       }
31       // Now, sort the array of pointers.
32       selectSort();
33    }
34 }
35
36 //********************************************************
37 // Destructor frees the memory allocated by the constructor*
38 //********************************************************
39 DonationList::~DonationList()
40 {
41    if (numDonations > 0)
42    {
43       delete [ ] donations;
44       donations = 0;
45       delete [ ] arrPtr;
46       arrPtr = 0;
47    }
48 }
49
50 //********************************************************
51 // The selecSort function performs a selection sort on the   *
52 // arrPtr array of pointers. The array is sorted on the      *
53 // values its elements point to.                             *
54 //********************************************************
55 void DonationList::selectSort()
56 {
57    int minIndex;
58    double *minElem;
59
60    for (int scan = 0; scan < (numDonations - 1); scan++)
61    {
62       minIndex = scan;
63       minElem = arrPtr[scan];
64       for(int index = scan + 1; index < numDonations; index++)
65       {
66          if (*(arrPtr[index]) < *minElem)
67          {
68             minElem = arrPtr[index];
69             minIndex = index;
70          }
71       }
72       arrPtr[minIndex] = arrPtr[scan];
73       arrPtr[scan] = minElem;
74    }
75 }
76
```

```
77 //*****************************************************
78 // The show function uses cout to display the donations *
79 // array in sequential order.                          *
80 //*****************************************************
81 void DonationList::show()
82 {
83    for (int count = 0; count < numDonations; count++)
84         cout << donations[count] << " ";
85         cout << endl;
86 }
87
88 //*****************************************************
89 // The showSorted function uses cout to display the values*
90 // pointed to by the elements of the arrPtr array. Since  *
91 // arrPtr is sorted, this function displays the elements  *
92 // of the donations array in ascending order.             *
93 //*****************************************************
94 void DonationList::showSorted()
95 {
96    for (int count = 0; count < numDonations; count++)
97       cout << *(arrPtr[count]) << " ";
98    cout << endl;
99 }
```

Implementing the Class

Program 10-19 shows how the class is used. The `funds` array is initialized with the 15 donation values.

Program 10-19

```
1  // This program shows the donations made to the United Cause
2  // by the employees of CK Graphics, Inc. It displays
3  // the donations in order from lowest to highest
4  // and in the original order they were received.
5  #include <iostream>
6  #include "donlist.h"
7  using namespace std;
8
9  int main()
10 {
11    double funds[] = {5,  100, 5,  25, 10,
12                      5,  25,  5,  5,  100,
13                      10, 15,  10, 5,  10 };
14    DonationList ckGraphics(15, funds);
15    cout << "The donations sorted in ascending order are:\n";
16    ckGraphics.showSorted();
17    cout << "The donations in their original order are:\n";
18    ckGraphics.show();
19    return 0;
20 }
```

(program continues)

Program 10-19 *(continued)*

Program Output

```
The donations sorted in ascending order are:
5 5 5 5 5 5 10 10 10 10 15 25 25 100 100
The donations in their original order are:
5 100 5 25 10 5 25 5 5 100 10 15 10 5 10
```

When the `ckGraphics` object is defined, the value 15 is passed to the constructor's `num` parameter, and the `funds` array is passed to the `gifts` parameter. The `showSorted` member function is called to display the donation values in ascending order (by using the `arrPtr` array to display them), and the `show` member function is called to display the values in their original order.

10.14 Tying It All Together: *Pardon Me, Do You Have the Time?*

Professor Susan Gonzalez wants to have her students take some of their tests online and has asked you to write a program to administer the tests. For each student, the program must record the student's starting time, the student's answer for each question, and the student's ending time. Before you write the program, you want to make sure that you can write code to accurately capture a student's start and end time. You decide to write a short program that experiments with the C++ library functions for telling time.

C++ libraries provide a number of data types and functions that can be used to determine the current calendar time. By convention, many computers mark calendar time by the number of seconds that have elapsed since a point in time that has come to be known among computer scientists as *the epoch*. In case you want to know, the epoch is midnight January 1, 1970.

The C++ data type `time_t` is used to represent the number of seconds since the epoch. The library function

```
time_t time (time_t * epSecs);
```

takes as parameter a pointer to a `time_t` object that will hold the value representing the current time. Program 10-20 illustrates the use of this function.

Program 10-20

```
1  // This program illustrates the use of the time function.
2  #include <iostream>
3  #include <ctime>    // Needed to use the time functions and types
4  using namespace std;
5
6  int main()
7  {
8      time_t epSeconds;
9      time(&epSeconds);
10     cout << "The number of seconds since the epoch is "
11          << epSeconds << endl;
12     return 0;
13 }
```

(program continues)

Program 10-20 *(continued)*

Program Output

```
The number of seconds since the epoch is 1247930628
```

Somewhat redundantly, the value stored in the parameter `epSecs` is also returned by the `time` function. This allows the programmer to pass `NULL` for the parameter to `time()` and use the returned value instead. The following program is equivalent to Program 10-20.

Program 10-21

```
1  // This program illustrates the use of the time function.
2  #include <iostream>
3  #include <ctime>    // Needed to use the time functions and types
4  using namespace std;
5
6  int main()
7  {
8      time_t epSeconds;
9      epSeconds = time(NULL);
10     cout << "Number of seconds since the epoch is "
11          << epSeconds << endl;
12     return 0;
13 }
```

Program Output

```
Number of seconds since the epoch is 1247930807
```

As useful as `time()` is, it does not solve all time-related problems. First, Professor Gonzalez would much prefer that chronological times be stated in a form such as

```
Friday June 6, 2009, 4:29PM
```

instead of as so many seconds after the epoch. Second, she wants the program to take differences in time zones into account and always give the correct local time. The C++ function

```
tm * localtime(const time_t *eps)
```

is exactly what is needed: it takes a `time_t` value, converts it into a structure of type `tm`, and returns the address of that structure. The members of `tm` are integers and have the meanings shown here:

```
int tm_min;  // Minutes after the hour (0..59)
int tm_hour; // Hours after midnight (0..23)
int tm_mday; // Day of the month (1..31)
int tm_mon;  // Month since January (0..11)
int tm_year; // Years since 1900
int tm_wday; // Weekday (Sunday=0, Monday=1, .. Saturday=6)
```

The following is an example of how to use `time()` in conjunction with `localtime()` to print the number of the current month:

```
time_t epSecs;       // Seconds since epoch
tm *pCalendarTime;   // Pointer to calendar time
// Get seconds since epoch
epSecs = time(NULL);
// Convert to local time
pCalendarTime = localtime(&epSecs);
// Print number of current month
cout << pCalendarTime->tm_mon;
```

The following program determines and prints the day of the week, month, and year of the time of its execution:

Program 10-22

```
1  // This program prints "today's" date
2  #include <iostream>
3  #include <ctime>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9      time_t epSeconds;        // Seconds since epoch
10     tm *pCalendarTime;       // Pointer to calendar time
11     // Array of weekday names
12     string wDay[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
13                      "Thursday", "Friday", "Saturday"
14                     };
15     // Array of month names
16     string month[] = {"January", "February", "March", "April",
17                       "May", "June", "July", "August", "September",
18                       "October", "November", "December"
19                      };
20
21     epSeconds = time(NULL);                // Seconds since epoch
22     pCalendarTime = localtime(&epSeconds); // Convert to local time
23
24     // Print day of month and day of week
25     cout << "Today is " << wDay[pCalendarTime->tm_wday]
26          << " " << month[pCalendarTime->tm_mon]
27          << " " << pCalendarTime->tm_mday
28          << ", " << 1900 + pCalendarTime->tm_year << endl;
29
30     return 0;
31 }
```

Program Output

```
Today is Friday July 17, 2009
```

Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. Each byte in memory is assigned a unique _____.
2. The _____ operator can be used to determine a variable's address.
3. _____ variables are designed to hold addresses.
4. The _____ operator can be used to work with the variable a pointer points to.
5. Array names can be used as _____ and vice-versa.
6. Creating variables while a program is running is called _____.
7. The _____ operator is used to dynamically allocate memory.
8. If the `new` operator cannot allocate the amount of memory requested, it throws _____.
9. A pointer that contains the address 0 is called a(n) _____ pointer.
10. When a program is finished with a chunk of dynamically allocated memory, it should free it with the _____ operator.
11. You should only use the `delete` operator to deallocate memory that was dynamically acquired with the _____ operator.
12. What does the indirection operator do?
13. Look at the following code.

```
int x = 7;
int *ptr = &x;
```

    What will be displayed if you send the expression `*iptr` to `cout`? What happens if you send the expression `ptr` to `cout`?
14. Name two different uses for the C++ operator *.
15. Which arithmetic operations can be applied to pointers?
16. Assuming that `ptr` is a pointer to an `int`, what happens when you add 4 to it?
17. Look at the following array definition.

```
int numbers [] = {2, 4, 6, 8, 10};
```

    What will the following statement display?

```
cout << *(numbers + 3) << endl;
```

18. What is the purpose of the new operator?
19. What happens when a program uses the `new` operator to allocate a block of memory, but the amount of requested memory isn't available? How do programs written with older compilers handle this?
20. Under what circumstances can you successfully return a pointer from a function?
21. What is the purpose of the `delete` operator?
22. What is the difference between a pointer to a constant and a constant pointer?
23. Show C++ code for defining a variable `ptr` that is a pointer to a constant `int`.
24. Show C++ code for defining a variable `ptr` that is a constant pointer to `int`.

C++ Language Elements

25. Consider the function

```
void change(int *p)
{
   *p = 20;
}
```

Show how to call the `change` function so that it sets the integer variable

```
int i;
```

to 20.

26. Consider the function

```
void modify(int & x)
{
   x = 10;
}
```

Show how to call the `modify` function so that it sets the integer

```
int i;
```

to 10.

Algorithm Workbench

27. Write a function whose prototype is

```
void exchange(int *p, int *q);
```

that takes two pointers to integer variables and exchanges the values in those variables.

28. Write a function

```
void switchEnds(int *array, int size);
```

that is passed the address of the beginning of an array and the size of the array. The function swaps the values in the first and last entries of the array.

Predict the Output

29. Given the variable initializations

```
int a[5] = {0, 10, 20, 30, 40};
int k = 3;
int *p = a + 1;
```

determine the output from each of the following statements:

A) `cout << a[k];`
B) `cout << *(a + k);`
C) `cout << *a;`
D) `cout << a[*a];`
E) `cout << a[*a + 2];`
F) `cout << *p;`
G) `cout << p[0];`
H) `cout << p[1];`

Find the Error

30. Each of the following declarations and program segments has errors. Locate as many as you can.

A)
```
int ptr*;
```

B)
```
int x, *ptr;
&x = ptr;
```

C)
```
int x, *ptr;
*ptr = &x;
```

D)
```
int x, *ptr;
ptr = &x;
ptr = 100; // Store 100 in x
cout << x << endl;
```

E)
```
int numbers[] = {10, 20, 30, 40, 50};
cout << "The third element in the array is ";
cout << *numbers + 3 << endl;
```

F)
```
int values[20], *iptr;
iptr = values;
iptr *= 2;
```

G)
```
double level;
int dPtr = &level;
```

H)
```
int *iptr = &ivalue;
int ivalue;
```

I)
```
void doubleVal(int val)
{
   *val *= 2;
}
```

J)
```
int *pint;
new pint;
```

K)
```
int *pint;
pint = new int;
pint = 100;
```

L)
```
int *pint;
pint = new int[100]; // Allocate memory
  .
  .
Process the array
  .
  .
delete pint;// Free memory
```

M)
```
int *getNum()
{
    int wholeNum;

    cout << "Enter a number: ";
    cin >> wholeNum;
    return &wholeNum;
}
```

Soft Skills

31. Suppose that you are a manager of a programming team. To facilitate project development and maintenance, you have decided to establish some programming and coding guidelines. Make a list of pointer-related programming guidelines you think will improve program readability and decrease pointer-related bugs.

Programming Challenges



1. Test Scores #1

Write a program that dynamically allocates an array large enough to hold a user-defined number of test scores. Once all the scores are entered, the array should be passed to a function that sorts them in ascending order. Another function should be called that calculates the average score. The program should display the sorted list of scores and averages with appropriate headings. Use pointer notation rather than array notation whenever possible.

Input Validation: Do not accept negative numbers for test scores.

2. Test Scores #2

Modify the program of Programming Challenge 1 to allow the user to enter name-score pairs. For each student taking a test, the user types a string representing the name of the student, followed by an integer representing the student's score. Modify both the sorting and average-calculating functions so they take arrays of structures, with each structure containing the name and score of a single student. In traversing the arrays, use pointers rather than array indices.



3. Money Money Money #1

Modify Program 10-19 (the *United Cause* case study program) so it can be used with any set of donations. The program should dynamically allocate the `donations` array and ask the user to input its values.

4. Money Money Money #2

Modify Program 10-19 (the *United Cause* case study program) so the `arrPtr` array is sorted in descending order instead of ascending order.



5. Pie a la Mode

In statistics the *mode* of a set of values is the value that occurs most often. Write a program that determines how many pieces of pie most people eat in a year. Set up an integer array that can hold responses from 30 people. For each person, enter the number of pieces they say they eat in a year. Then write a function that finds the mode of these 30 values. This will be the number of pie slices eaten by the most people. The function that finds and returns the mode should accept two arguments, an array of integers, and a value indicating how many elements are in the array.

6. Median Function

In statistics the median of a set of values is the value that lies in the middle when the values are arranged in sorted order. If the set has an even number of values, then the median is

taken to be the average of the two middle values. Write a function that determines the median of a sorted array. The function should take an array of numbers and an integer indicating the size of the array and return the median of the values in the array. You may assume the array is already sorted. Use pointer notation whenever possible.

7. Movie Statistics

Write a program that can be used to gather statistical data about the number of movies college students see in a month. The program should ask the user how many students were surveyed and dynamically allocate an array of that size. The program should then allow the user to enter the number of movies each student has seen. The program should then calculate the average, median, and mode of the values entered.



VideoNote
Solving the Days in Current Month Problem

8. Days in Current Month

Write a program that can determine the number of days in a month for a specified month and year. The program should allow a user to enter two integers representing a month and a year, and it should determine how many days are in the specified month. The integers 1 through 12 will be used to identify the months of January through December. The user indicates the end of input by entering 0 0 for the month and year. At that point, the program prints the number of days in the current month and terminates.

Use the following criteria to identify leap years:

1. A year *Y* is divisible by 100. Then *Y* is a leap year if and if only it is divisible by 400. For example, 2000 is a leap year but 2100 is not.
2. A year *Y* is not divisible by 100. Then *Y* is a leap year if and if only it is divisible by 4. For example, 2008 is a leap year but 2009 is not.

Here is sample run of the program:

```
Enter month and year: 2 2008[Enter]
29 days
Enter month and year: 0 0[Enter]

The current month, September 2009, has 30 days.
```

9. Age

Write a program that asks for the user's name and year of birth, greets the user by name, and declares the user's age in years. Users are assumed to be born between the years 1800 and 2099, and should enter the year of birth in one of the three formats 18XX, 19XX, or 20XX. A typical output should be "Hello Caroline, you are 23 years old."

CHAPTER 11

More About Classes and Object-Oriented Programming

TOPICS

11.1 The `this` Pointer and Constant Member Functions
11.2 Static Members
11.3 Friends of Classes
11.4 Memberwise Assignment
11.5 Copy Constructors
11.6 Operator Overloading
11.7 Type Conversion Operators
11.8 Convert Constructors
11.9 Aggregation and Composition
11.10 Inheritance
11.11 Protected Members and Class Access
11.12 Constructors, Destructors, and Inheritance
11.13 Overriding Base Class Functions
11.14 Tying It All Together: *Putting Data on the World Wide Web*

11.1 The `this` Pointer and Constant Member Functions

CONCEPT: By default, the compiler provides each member function of a class with an implicit parameter that points to the object through which the member function is called. The implicit parameter is called **`this`**. A constant member function is one that does not modify the object through which it is called.

The `this` Pointer

Consider the class

```
class Example
{
   int x;
   public:
      Example(int a){ x = a;}
      void setValue(int);
      int getValue();
};
```

with the member function

```
int Example::getValue()
{
   return x;
}
```

that simply returns the value in an object of the class. As an example, the `getValue` function might be invoked in a program such as

```
int main()
{
   Example ob1(10), ob2(20);
   cout << ob1.getValue() << " " << ob2.getValue();
   return 0;
}
```

in which case the program would print out the values `10  20`.

You learned in an earlier chapter that the different objects of a structure or class type are called *instances* of that class, and that each instance of a class has its own copy of the data members listed in the class. These data members, called *instance members* because they belong to instances of the class, can have different values in different objects. Thus in the preceding example, the instance member `x` in the `ob1` object has a value of `10` while `x` in `ob2` has a value of `20`.

Now consider again the code for the member function

```
int Example::getValue()
{
   return x;
}
```

This function is supposed to return the `x` member of some object of the `Example` class, but how does it know which object to use? What happens is that by default, the compiler provides each member function of every class with an implicit parameter that is a pointer to an object of the class. Thus for example, the `getValue` function is equipped with a single parameter of type pointer to `Example`. Similarly, the member function

```
void Example::setValue(int a)
{
   x = a;
}
```

although written by the programmer to take a single parameter of type `int`, in reality has two parameters: an pointer to an object of the class `Example`, and the

```
int a
```

parameter specified by the programmer. In all cases, the actual parameter for the implicit object parameter is the address of the object through which the member function is being called. Thus in the call

```
ob1.getValue()
```

the implicit parameter passed to `getValue` is the address of `ob1`, whereas in the call

```
ob2.setValue(78)
```

the implicit parameter passed to `setValue` is `&ob2`.

The implicit pointer passed by the compiler to a member function can be accessed by code inside that function by using the reserved keyword `this`. So for example, a member function of the `Example` class could access the object through which it is called by using the expression

```
*this
```

and it could also access any of the members of that object through the same pointer. Program 11-1 illustrates these concepts. It modifies the `Example` class to include a member function that uses the `this` pointer to print the address of the object through which it is called as well as the value of the instance member `x` in the same object.

Program 11-1

Contents of `ThisExample.h`

```
1 class Example
2 {
3    int x;
4  public:
5    Example(int a){ x = a;}
6    void setValue(int);
7    void printAddressAndValue();
8 };
```

Contents of `ThisExample.cpp`

```
1 #include "ThisExample.h"
2 #include <iostream>
3 using namespace std;
4
5 //*****************************************
6 // Set value of object.                   *
7 //*****************************************
8 void Example::setValue(int a)
9 {
10   x = a;
11 }
12 //*****************************************
13 // Print address and value.                *
14 //*****************************************
15 void Example::printAddressAndValue()
16 {
17   cout << "The object at address " << this << " has "
18        << "value " << (*this).x << endl;
19 }
```

(program continues)

Program 11-1 *(continued)*

Contents of main program, pr11-1.cpp

```
1  // This program illustrates the this pointer.
2  #include <iostream>
3  #include "ThisExample.h"
4  using namespace std;
5
6  int main()
7  {
8     Example ob1(10), ob2(20);
9
10    // Print the addresses of the two objects.
11    cout << "Addresses of objects are " << &ob1
12         << " and " << &ob2 << endl;
13
14    // Print the addresses and values from within
15    // the member function.
16    ob1.printAddressAndValue();
17    ob2.printAddressAndValue();
18
19    return 0;
20 }
```

Program Output

```
Addresses of objects are 0x241ff5c and 0x241ff58
The object at address 0x241ff5c has value 10
The object at address 0x241ff58 has value 20
```

As an example of a common use of the `this` pointer, consider the member function

```
void Example::setValue(int a)
{
   x = a;
}
```

It is natural to name the parameter to be used to set the value of the member `x` using an identifier that makes its connection to `x` explicit, perhaps `xValue` or even `x` itself. However, a formal parameter of a member function with the same identifier as a member of the class will hide the class member, making it inaccessible inside the function. The `this` pointer can be used to qualify the name of the class member and make it visible again. Here is the `setValue` member function rewritten in this manner:

```
void Example::setValue(int x)
{
   this->x = x;
}
```

Recall from Chapter 10 that the notation `this->x` is equivalent to `(*this).x`.

Constant Member Functions

A parameter that is passed to a function by reference or through a pointer may be modified by that function. The `const` key word is used with a parameter to prevent the called function from modifying it. For example, a function declared as

```
void fun(const string *str);
```

takes a pointer to a string object as a parameter, but will not be able to modify the object. There is a similar mechanism that can be used to protect the implicit parameter `*this` from being modified by a member function. When placed right after the parameter list in the definition of a member function, the `const` keyword serves as an indication to the compiler that the member function should not be allowed to modify its object. If the member function is defined outside the class, both the in-class declaration and the definition must have the `const`. Here is an example

```
class ConstExample
{
   int x;
public:
   ConstExample(int a){ x = a;}
   void setValue(int);
   int getValue() const;
};
```

The definition of the `getValue` function would be

```
int ConstExample::getValue() const
{
   return x;
}
```

11.2 Static Members

CONCEPT: **If a member variable is declared `static`, all objects of that class have access to that variable. If a member function is declared `static`, it may be called before any instances of the class are defined.**

Each class object (an instance of a class) has its own copy of the class's member variables. An object's member variables are separate and distinct from the member variables of other objects of the same class. For example, consider the following declaration:

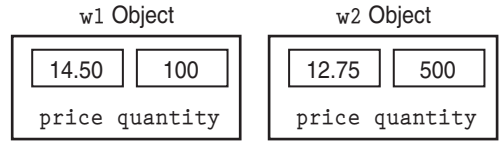
```
class Widget
{
   private:
      double price;
      int quantity;
   public:
      Widget(double p, int q)
          { price = p; quantity = q; }
      double getPrice()
          { return price; }
      int getQuantity()
          { return quantity; }
};
```

Assume that in a program, two separate instances of the `Widget` class are created by the following declaration:

```
Widget w1(14.50, 100), w2(12.75, 500);
```

This statement creates `w1` and `w2`, two distinct objects. Each has its own `price` and `quantity` member variables. This is illustrated by Figure 11-1.

Figure 11-1



When the `getQuantity` member function of either instance is called, it returns the value stored in the calling object's `quantity` variable. Based on the values initially stored in the objects, the statement

```
cout << w1.getQuantity() << " " << w2.getQuantity();
```

will cause `100 500` to be displayed.

Static Member Variables

It's possible to create a member variable that is shared by all the objects of the same class. To create such a member, simply place the key word `static` in front of the variable declaration, as shown in the following class:

```
class StatDemo
{
  private:
     static int x;
     int y;
  public:
     void setx(int a) { x = a; }
     void sety(int b) { y = b; }
     int getx() { return x; }
     int gety() { return y; }
};
```

Next, place a separate definition of the variable outside the class, such as:

```
int StatDemo::x;
```

In this example, the member variable `x` will be shared by all objects of the `StatDemo` class. When one class object puts a value in `x`, it will appear in all other `StatDemo` objects. For example, assume the following statements appear in a program:

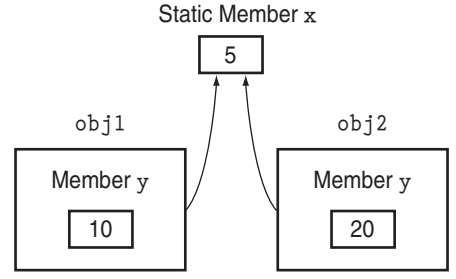
```
StatDemo obj1, obj2;
obj1.setx(5);
obj1.sety(10);
obj2.sety(20);
cout << "x: " << obj1.getx() << " " << obj2.getx() << endl;
cout << "y: " << obj1.gety() << " " << obj2.gety() << endl;
```

The cout statements shown will display

```
x: 5 5
y: 10 20
```

The value 5 is stored in the static member variable x by the object obj1. Since obj1 and obj2 share the variable x, the value 5 shows up in both objects. This is illustrated by Figure 11-2.

Figure 11-2



Both obj1 and obj2 share the static member x

A more practical use of a static member variable is demonstrated in Program 11-2. The Budget class is used to gather the budget requests for all the divisions of a company. The class uses a static member, corpBudget, to hold the amount of the overall corporate budget. When the member function addBudget is called, its argument is added to the current contents of corpBudget. By the time the program is finished, corpBudget will contain the total of all the values placed there by all the Budget class objects.

Program 11-2

Contents of budget.h

```
1  #ifndef BUDGET_H
2  #define BUDGET_H
3
4  class Budget
5  {
6  private:
7     static double corpBudget;
8     double divBudget;
9  public:
10    Budget() { divBudget = 0; }
11    void addBudget(double b)
12       { divBudget += b; corpBudget += divBudget; }
13    double getDivBudget() { return divBudget; }
14    double getCorpBudget() { return corpBudget; }
15 };
16 #endif
```

(program continues)

Program 11-2 *(continued)*

Contents of main program, pr11-2.cpp

```
// This program demonstrates a static class member variable.
#include <iostream>
#include <iomanip>
#include "budget.h"          // For Budget class declaration
using namespace std;

// Definition of the static member of the Budget class.
double Budget::corpBudget = 0;

int main()
{
   const int N_DIVISIONS = 4;
   Budget divisions[N_DIVISIONS];

   // Get the budget request for each division.
   for (int count = 0; count < N_DIVISIONS; count++)
   {
      double bud;

      cout << "Enter the budget request for division ";
      cout << (count + 1) << ": ";
      cin >> bud;
      divisions[count].addBudget(bud);
   }

   // Display the budget request for each division.
   cout << setprecision(2);
   cout << showpoint << fixed;
   cout << "\nHere are the division budget requests:\n";
   for (int count = 0; count < N_DIVISIONS; count++)
   {
     cout << "Division " << (count + 1) << "\t$ ";
     cout << divisions[count].getDivBudget() << endl;
   }

   // Display the total budget request.
   cout << "Total Budget Requests:\t$ ";
   cout << divisions[0].getCorpBudget() << endl;

   return 0;
}
```

(program continues)

Program 11-2 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter the budget request for division 1: 102000[Enter]
Enter the budget request for division 2: 201000[Enter]
Enter the budget request for division 3: 570000[Enter]
Enter the budget request for division 4: 100100[Enter]

Here are the division budget requests:
Division 1      $ 102000.00
Division 2      $ 201000.00
Division 3      $ 570000.00
Division 4      $ 100100.00
Total Budget Requests:  $ 973100.00
```



NOTE: Static member variables furnish a good example of the distinction between C++ declarations and C++ definitions. A *declaration* provides information about the existence and type of a variable or function. A *definition* provides all the information contained in a declaration, and in addition, causes memory to be allocated for the variable or function being defined. Static member variables must be *declared* inside the class and *defined* outside of it.

In general, we can divide the member variables and functions of a class into two groups: *instance members* and *static members*. An instance member is one whose use must be associated with a particular instance of the class. In particular, an instance variable of a class must be accessed through a specific instance of its class, and an instance member function must be called through a specific instance of its class.

In contrast, the use of a static member variable, or the call of a static member function, does not need to be associated with any instance. Only the class of the static member needs to be specified.

Static Member Functions

A member function of a class can be declared static by prefixing its declaration with the keyword `static`. Here is the general form:

```
static <return type><function name>(<parameter list>)
```

Static member functions are normally used to work with static member variables of the class. In fact, member functions that do not access any nonstatic members of their class, such as `getCorpBudget()` in Program 11-2, should be made static.

Program 11-3, a modification of Program 11-2, demonstrates this. It asks the user to enter the main office's budget request before any division requests are entered. The `Budget` class has been modified to include a static member function named `mainOffice`. This function adds its argument to the static `corpBudget` variable and is called before any instance of the `Budget` class is defined. The `getCorpBudget()` function has also been made static.

Program 11-3

Contents of budget2.h

```
#ifndef BUDGET_H
#define BUDGET_H

class Budget
{
private:
   static double corpBudget;
   double divBudget;
public:
   Budget() { divBudget = 0; }
   void addBudget(double b)
      { divBudget += b; corpBudget += divBudget; }
   double getDivBudget() { return divBudget; }
   static double getCorpBudget() { return corpBudget; }
   static void mainOffice(double);
};
#endif
```

Contents of budget2.cpp

```
#include "budget2.h"

// Definition of the static member of Budget class.
double Budget::corpBudget = 0;

//*********************************************************
// Definition of static member function mainOffice        *
// This function adds the main office's budget request to  *
// the corpBudget variable.                                *
//*********************************************************
void Budget::mainOffice(double budReq)
{
   corpBudget += budReq;
}
```

Contents of main program, pr11-3.cpp

```
// This program demonstrates a static class member function.
#include <iostream>
#include <iomanip>
#include "budget2.h"        // For Budget class declaration
using namespace std;

int main()
{
   const int N_DIVISIONS = 4;

```

(program continues)

Program 11-3 *(continued)*

```
11    // Get the budget requests for each division.
12    cout << "Enter the main office's budget request: ";
13    double amount;
14    cin >> amount;
15    // Call the static member function of the Budget class.
16    Budget::mainOffice(amount);
17    // Create instances of the Budget class.
18    Budget divisions[N_DIVISIONS];
19    for (int count = 0; count < N_DIVISIONS; count++)
20    {
21       double bud;
22
23       cout << "Enter the budget request for division ";
24       cout << (count + 1) << ": ";
25       cin >> bud;
26       divisions[count].addBudget(bud);
27    }
28
29    // Display the budget for each division.
30    cout << setprecision(2);
31    cout<< showpoint << fixed;
32    cout << "\nHere are the division budget requests:\n";
33    for (int count = 0; count < N_DIVISIONS; count++)
34    {
35       cout << "\tDivision " << (count + 1) << "\t$ ";
36       cout << divisions[count].getDivBudget() << endl;
37    }
38
39    // Print total budget requests.
40    cout << "Total Requests (including main office): $ ";
41    cout << Budget::getCorpBudget() << endl;
42    return 0;
43 }
```

Program Output with Example Input Shown in Bold

```
Enter the main office's budget request: 400000[Enter]
Enter the budget request for division 1: 102000[Enter]
Enter the budget request for division 2: 210000[Enter]
Enter the budget request for division 3: 240000[Enter]
Enter the budget request for division 4: 105000[Enter]

Here are the division budget requests:
        Division 1      $ 102000.00
        Division 2      $ 210000.00
        Division 3      $ 240000.00
        Division 4      $ 105000.00
Total Requests (including main office): $ 1057000.00
```

Notice the statement that calls the static function `mainOffice`:

```
Budget::mainOffice(amount);
```

Calls to static member functions are normally made by connecting the function name to the class name with the scope resolution operator. If objects of the class have been defined, static member functions can also be called by connecting their names to the object with the dot operator. Thus the last output statement of Program 11-3 could be written as

```
cout << divisions[0].getCorpBudget() << endl;
```

The `this` pointer cannot be used in a static member function, because static member functions are not called through any instance of their class. Moreover, a static member function cannot access an instance member of its class unless it specifies what instance the member belongs to. For example, in the class

```
class StatAccess
{
   private:
      int x;
   public:
   static void output()
      {
         cout << x;  // incorrect access of non-static member
      }
   StatAccess(int x) { this->x = x; }
};
```

The attempt to access `x` in the statement `cout << x` is incorrect because it is tantamount to an implicit use of the `this` pointer, which the static function `output` does not have. In contrast, in the following modified example of the same class, the static member function `print` correctly accesses the nonstatic member `x` because it qualifies it with the name of a class object passed to it as a parameter.

```
class StatAccess
{
   private:
      int x;
   public:
      static void print(StatAccess a)
      {
         cout << a.x;
      }
   StatAccess(int x) { this->x = x; }
};
```

An advantage of static member functions is they can be called before any instances of the class have been created. This allows them to be used to perform complex initialization tasks that have to be done before objects of the class have been created.

11.3 Friends of Classes

CONCEPT: A friend is a function that is not a member of a class, but has access to the private members of the class.

Private members are hidden from all parts of the program outside the class, and accessing them requires a call to a public member function. Sometimes you will want to create an exception to that rule. A *friend* function is a function that is not a member of a class, but that has access to the class's private members. In other words, a friend function is treated as if it were a member of the class. A friend function can be a regular stand-alone function, or it can be a member of another class. (In fact, an entire class can be declared a friend of another class.)

In order for a function or class to become a friend of another class, it must be declared as such by the class granting it access. Classes keep a "list" of their friends, and only the external functions or classes whose names appears in the list are granted access. A function is declared a friend by placing the keyword `friend` in front of a prototype of the function. Here is the general format:

```
friend <return type><function name>(<parameter type list>);
```

In the following declaration of the `Budget` class, the `addBudget` function of another class, `Aux`, has been declared a friend:

```
class Budget
{
private:
   static double corpBudget;
   double divBudget;
public:
   Budget() { divBudget = 0; }
   void addBudget(double b)
      { divBudget += b; corpBudget += divBudget; }
   double getDivBudget() { return divBudget; }
   static double getCorpBudget() { return corpBudget; }
   static void mainOffice(double);
   friend void Aux::addBudget(double);    // A friend
};
```

Let's assume another class `Aux` represents a division's auxiliary office, perhaps in another country. The auxiliary office makes a separate budget request, which must be added to the overall corporate budget. The friend declaration of the `Aux::addBudget` function tells the compiler that the function is to be granted access to `Budget`'s private members. The function takes an argument of type `double` representing an amount to be added to the corporate budget:

```
class Aux
{
private:
   double auxBudget;
```

```
public:
   Aux() { auxBudget = 0; }
   void addBudget(double);
   double getDivBudget() { return auxBudget; }
};
```

And here is the definition of the `Aux addBudget` member function:

```
void Aux::addBudget(double b)
{
   auxBudget += b;
   Budget::corpBudget += auxBudget;
}
```

The parameter `b` is added to the corporate budget, which is accessed by using the expression `Budget::corpBudget`. Program 11-4 demonstrates the classes in a complete program.

Program 11-4

Contents of `auxil.h`

```
1  #ifndef AUXIL_H
2  #define AUXIL_H
3
4  // Aux class declaration.
5  class Aux
6  {
7  private:
8     double auxBudget;
9  public:
10    Aux() { auxBudget = 0; }
11    void addBudget(double);
12    double getDivBudget() { return auxBudget; }
13 };
14 #endif
```

Contents of `budget3.h`

```
1  #ifndef BUDGET3_H
2  #define BUDGET3_H
3  #include "auxil.h"   // For Aux class declaration
4
5  // Budget class declaration.
6  class Budget
7  {
8  private:
9     static double corpBudget;
10    double divBudget;
11 public:
12    Budget() { divBudget = 0; }
13    void addBudget(double b)
14       { divBudget += b; corpBudget += divBudget; }
15    double getDivBudget() { return divBudget; }
16    static double getCorpBudget() { return corpBudget; }
17    static void mainOffice(double);
18    friend void Aux::addBudget(double);
19 };
20 #endif
```

(program continues)

Program 11-4 *(continued)*

Contents of budget3.cpp

```
1  #include "budget3.h"
2
3  // Definition of static member.
4  double Budget::corpBudget = 0;
5
6  //*********************************************************
7  // Definition of static member function mainOffice        *
8  // This function adds the main office's budget request to  *
9  // the corpBudget variable.                               *
10 //*********************************************************
11 void Budget::mainOffice(double budReq)
12 {
13    corpBudget += budReq;
14 }
```

Contents of auxil.cpp

```
1  #include "auxil.h"
2  #include "budget3.h"
3
4  //*********************************************************
5  // Definition of member function addBudget                *
6  // This function is declared a friend by the Budget class  *
7  // It adds the value of argument b to the static corpBudget *
8  // member variable of the Budget class.                   *
9  //*********************************************************
10
11 void Aux::addBudget(double b)
12 {
13    auxBudget += b;
14    Budget::corpBudget += auxBudget;
15 }
```

Contents of main program pr11-4.cpp

```
1  // This program demonstrates a static class member variable.
2  #include <iostream>
3  #include <iomanip>
4  #include "budget3.h"
5  using namespace std;
6
7  int main()
8  {
9     const int N_DIVISIONS = 4;
10
11    // Get the budget requests for the divisions and
12    // offices.
13    cout << "Enter the main office's budget request: ";
14    double amount;
15    cin >> amount;
16    Budget::mainOffice(amount);
17
```

(program continues)

Program 11-4 *(continued)*

```
18    // Create the division and auxiliary offices.
19    Budget divisions[N_DIVISIONS];
20    Aux auxOffices[N_DIVISIONS];
21
22    cout << "\nEnter the budget requests for the divisions and  "
23         << "\ntheir auxiliary offices as prompted:\n";
24    for (int count = 0; count < N_DIVISIONS; count++)
25    {
26        double bud;
27        cout <<  "Division " << (count + 1) << ": ";
28        cin >> bud;
29        divisions[count].addBudget(bud);
30        cout << "Division " << (count + 1) << "'s auxiliary office: ";
31        cin >> bud;
32        auxOffices[count].addBudget(bud);
33     }
34
35     // Print the budgets.
36     cout << setprecision(2);
37     cout << showpoint << fixed;
38     cout << "Here are the division budget requests:\n";
39     for (int count = 0; count < N_DIVISIONS; count++)
40     {
41        cout << "\tDivision: " << (count + 1) << "\t\t\t$ ";
42        cout << setw(7);
43        cout << divisions[count].getDivBudget() << endl;
44        cout << "\tAuxiliary Office of Division " << (count+1);
45        cout << "\t$  ";
46        cout << auxOffices[count].getDivBudget() << endl;
47     }
48     // Print total requests.
49     cout << "\tTotal Requests (including main office): $ ";
50     cout << Budget::getCorpBudget() << endl;
51     return 0;
52 }
```

Program Output with Example Input Shown in Bold

```
Enter the main office's budget request: 100000[Enter]

Enter the budget requests for the divisions and
their auxiliary offices as prompted:
Division 1: 100000[Enter]
Division 1's auxiliary office: 500000[Enter]
Division 2: 200000[Enter]
Division 2's auxiliary office: 40000[Enter]
Division 3: 300000[Enter]
Division 3's auxiliary office: 700000[Enter]
Division 4: 400000[Enter]
Division 4's auxiliary office: 650000[Enter]
```

(program output continues)

Program 11-4 *(continued)*

```
Here are the division budget requests:
        Division: 1                     $ 100000.00
        Auxiliary Office of Division 1  $  50000.00
        Division: 2                     $ 200000.00
        Auxiliary Office of Division 2  $  40000.00
        Division: 3                     $ 300000.00
        Auxiliary Office of Division 3  $  70000.00
        Division: 4                     $ 400000.00
        Auxiliary Office of Division 4  $  65000.00
        Total Requests (including main office): $ 1325000.00
```



NOTE: As mentioned before, it is possible to make an entire class a friend of another class. The `Budget` class could make the `Aux` class its friend with the following declaration:

```
friend class Aux;
```

This may not be a good idea, however. Every member function of `Aux` (including ones that may be added later) would have access to the private members of `Budget`. The best practice is to declare as friends only those functions that must have access to the private members of the class.



Checkpoint

11.1 What is the difference between an instance member variable and a static member variable?

11.2 Static member variables are declared inside the class declaration. Where are static member variables defined?

11.3 Does a static member variable come into existence in memory before, at the same time as, or after any instances of its class?

11.4 What limitation does a static member function have?

11.5 What action is possible with a static member function that isn't possible with an instance member function?

11.6 If class `X` declares function `f` as a friend, does function `f` become a member of class `X`?

11.7 Suppose that class `Y` is a friend of class `X`, meaning that the member functions of class `Y` have access to all the members of class `X`. Should the friend key word appear in class `Y`'s declaration or in class `X`'s declaration?

11.4 Memberwise Assignment

CONCEPT: **The = operator may be used to assign one object to another, or to initialize one object with another object's data. By default, each member of one object is copied to its counterpart in the other object.**

Like other variables (except arrays), objects may be assigned to each other using the = operator. As an example, consider Program 11-5, which uses a `Rectangle` class similar to the one discussed in Chapter 7:

Program 11-5

```
1  // This program demonstrates object assignment.
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle
6  {
7  private:
8      double width, length;
9  public:
10     Rectangle(double width, double length)
11     {
12         this->width = width;
13         this->length = length;
14     }
15     double getWidth() { return width; }
16     double getLength() { return length; }
17     void output()
18     {
19         cout << "Width is " << width << ", "
20              << "Length is " << length << endl;
21     }
22 };
23
24 int main()
25 {
26    // Set up two rectangle objects.
27    Rectangle box1(10, 20), box2(5, 10);
28
29    // Display the rectangle objects.
30    cout << "Before the assignment:\n";
31    cout << "Box 1 data:\t";  box1.output();
32    cout << "Box 2 data:\t";  box2.output();
33
34    // Assignment.
35    box2 = box1;
36
37    // Display the rectangle objects.
38    cout << "\nAfter the assignment:\n";
```

(program continues)

Program 11-5 *(continued)*

```
39    cout << "Box 1 data:\t"; box1.output();
40    cout << "Box 2 data:\t"; box2.output();
41    return 0;
42 }
```

Program Output

```
Before the assignment:
Box 1 data:     Width is 10, Length is 20
Box 2 data:     Width is 5, Length is 10

After the assignment:
Box 1 data:     Width is 10, Length is 20
Box 2 data:     Width is 10, Length is 20
```

As you can see, the statement

```
box2 = box1
```

copied the `width` and `length` variables of `box1` directly into the `width` and `length` variables of `box2`.

Memberwise assignment also occurs when one object is initialized with another object's values. Remember the difference between assignment and initialization: assignment occurs between two objects that already exist, and initialization happens to an object being created. Consider the following program segment:

```
Rectangle box1(10, 50);
Rectangle box2 = box1;
```

The second statement defines a `Rectangle` object `box2` and initializes it to the values stored in `box1`. Because memberwise assignment takes place, the `box2` object will contain the same values as the `box1` object.

11.5 Copy Constructors

CONCEPT: **A copy constructor is a special constructor that is called whenever a new object is created and initialized with the data of another object of the same class.**

Many times it makes sense to create an object and have it start out with its data being the same as that of another, previously created object. For example, if Mary and Joan live in the same house and an address object for Mary has already been created, it makes sense to initialize Joan's address object to a copy of Mary's. In particular, suppose we have the following class to represent addresses:

```
class Address
{
private:
   string street;
```

```
public:
   Address() { street = ""; }
   Address(string st) { setStreet(st); }
   void setStreet(string st) { street = st; }
   string getStreet() { return street; }
};
```

We could then create Mary's address and then initialize Joan's address to a copy of Mary's using the following code:

```
Address mary("123 Main St");
Address joan = mary;
```

Recall that a constructor must execute whenever an object is being created. When an object is created and initialized with another object of the same class, the compiler automatically calls a special constructor, called a *copy constructor,* to perform the initialization using the existing object's data. This copy constructor can be specified by the programmer, as we will shortly show.

The Default Copy Constructor

If the programmer does not specify a copy constructor for the class, then the compiler automatically calls a *default copy constructor.* This default copy constructor simply copies the data of the existing object to the new object using memberwise assignment.

Most of the time, the default copy constructor provides the kind of behavior that we want. For example, if after initializing Joan's address with Mary's, Joan later moves out and gets her own place, we can change Joan's address without affecting Mary's. This is illustrated in Program 11-6.

Program 11-6

```
1  // This program demonstrates the operation of the
2  // default copy constructor.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  class Address
8  {
9  private:
10   string street;
11 public:
12   Address() { street = ""; }
13   Address(string st) { setStreet(st); }
14   void setStreet(string st) { street = st; }
15   string getStreet() { return street; }
16 };
17
18 int main()
19 {
20   // Mary and Joan live at same address.
```

(program continues)

Program 11-6 *(continued)*

```
21   Address mary("123 Main St");
22   Address joan = mary;
23   cout << "Mary lives at " << mary.getStreet() << endl;
24   cout << "Joan lives at " << joan.getStreet() << endl;
25
26   // Now Joan moves out.
27   joan.setStreet("1600 Pennsylvania Ave");
28   cout << "Now Mary lives at " << mary.getStreet() << endl;
29   cout << "Now Joan lives at " << joan.getStreet() << endl;
30
31   return 0;
32 }
```

Program Output

```
Mary lives at 123 Main St
Joan lives at 123 Main St
Now Mary lives at 123 Main St
Now Joan lives at 1600 Pennsylvania Ave
```

Deficiencies of Default Copy Constructors

There are times, however, when the behavior of the default copy constructor is not what we expect. Consider a class

```
class NumberArray
{
private:
   double *aPtr;
   int arraySize;
public:
   NumberArray(int size, double value);
   // ~NumberArray(){ if (arraySize > 0) delete [] aPtr;}
   void print();
   void setValue(double value);
};
```

that encapsulates an array of numbers of type `double` (in practice there may be other members of the class as well). To allow flexibility for different size arrays, the class contains a pointer to the array instead of directly containing the array itself. The constructor of the class, whose code is shown below, allocates an array of a specified size, then sets all the entries of the array to a given value. The class has member functions for printing the array and for setting the entries of the array to a given (possibly different) value. The class's destructor uses the delete `[]` statement to deallocate the array (see Chapter 10) but is currently commented out to avoid problems caused by the default copy constructor. We shall shortly point out the specific nature of these problems.

Program 11-7 creates an object of the class, creates and initializes a second object with the data of the first, and then changes the array in the second object. As shown by the output of the program, changing the second object's data changes the data in the first object. In many cases, this is undesirable and leads to bugs.

Program 11-7

Contents of NumberArray.h

```
1  #include <iostream>
2  using namespace std;
3
4  class NumberArray
5  {
6  private:
7     double *aPtr;
8     int arraySize;
9  public:
10    NumberArray(int size, double value);
11    // ~NumberArray(){ if (arraySize > 0) delete [ ] aPtr;}
12    // Commented out to avoid problems with the
13    // default copy constructor.
14    void print();
15    void setValue(double value);
16 };
```

Contents of NumberArray.cpp

```
1  #include <iostream>
2  #include "NumberArray.h"
3  using namespace std;
4
5  //*****************************************
6  //Constructor allocates an array of the       *
7  //given size and sets all its entries to the  *
8  //given value.                                *
9  //*****************************************
10 NumberArray::NumberArray(int size, double value)
11 {
12    arraySize = size;
13    aPtr = new double[arraySize];
14    setValue(value);
15 }
16
17 //*********************************************************
18 //Sets all the entries of the array to the same value.  *
19 //*********************************************************
20 void NumberArray::setValue(double value)
21 {
22    for(int index = 0; index < arraySize; index++)
23       aPtr[index] = value;
24 }
25
26 //**************************************
27 //Prints all the entries of the array.  *
28 //**************************************
29 void NumberArray::print()
30 {
31    for(int index = 0; index < arraySize; index++)
32       cout << aPtr[index] << "  ";
33 }
```

(program continues)

Program 11-7 *(continued)*

Contents of `Pr11-7.cpp`

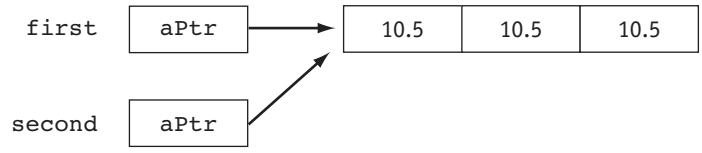
```
1  // This program demonstrates the deficiencies of
2  // the default copy constructor.
3  #include <iostream>
4  #include <iomanip>
5  #include "NumberArray.h"
6  using namespace std;
7
8  int main()
9  {
10    // Create an object.
11    NumberArray first(3, 10.5);
12
13    // Make a copy of the object.
14    NumberArray second = first;
15
16    // Display the values of the two objects.
17    cout << setprecision(2) << fixed << showpoint;
18    cout << "Value stored in first object is ";
19    first.print();
20    cout << endl << "Value stored in second object is ";
21    second.print();
22    cout << endl << "Only the value in second object "
23         << "will be changed."  << endl;
24
25    // Now change the  value stored in the second object.
26    second.setValue(20.5);
27
28    // Display the values stored in the two objects.
29    cout << "Value stored in first object is ";
30    first.print();
31    cout << endl << "Value stored in second object is ";
32    second.print();
33
34    return 0;
35 }
```

Program Output

```
Value stored in first object is 10.50  10.50  10.50
Value stored in second object is 10.50  10.50  10.50
Only the value in second object will be changed.
Value stored in first object is 20.50  20.50  20.50
Value stored in second object is 20.50  20.50  20.50
```

The reason changing the data in one object changes the other object is that the memberwise assignment performed by the default copy constructor copies the value of the pointer in the first object to the pointer in the second object, *leaving both pointers pointing to the same data*. Thus when one of the objects changes its data through its pointer, it affects the other object as well. This is illustrated in Figure 11-3.

Figure 11-3



The fact that the two pointers point to the same memory location will also cause problems when the destructors for the two objects try to deallocate the same memory (that is why the destructor code in the above class is commented out). In general, classes with pointer members will not behave correctly under the default copy constructor provided by the compiler. They must be provided with a copy constructor written by the programmer.

Programmer-Defined Copy Constructors

A programmer can define a copy constructor for a class. A programmer-defined copy constructor must have a single parameter that is a reference to the *same* class. Thus in the case of the previous example, the prototype for the copy constructor would be

```
NumberArray::NumberArray(NumberArray &obj)
```

This copy constructor avoids the problems of the default copy constructor by allocating separate memory for the pointer of the new object before doing the copy:

```
NumberArray::NumberArray(NumberArray &obj)
{
   arraySize = obj.arraySize;
   aPtr = new double[arraySize];
   for(int index = 0; index < arraySize; index++)
      aPtr[index] = obj.aPtr[index];
}
```

Program 11-8 demonstrates the use of the `NumberArray` class modified to have a copy constructor. The class declaration is in the `NumberArray2.h` file, with the implementations of its member functions being given in `NumberArray2.cpp`.

Program 11-8

Contents of `NumberArray2.h`

```
1  #include <iostream>
2  using namespace std;
3
4  class NumberArray
5  {
6  private:
7    double *aPtr;
8    int arraySize;
9  public:
10   NumberArray(NumberArray &);
11   NumberArray(int size, double value);
```

(program continues)

Program 11-8 *(continued)*

```
12   ~NumberArray() { if (arraySize > 0) delete [] aPtr; }
13   void print();
14   void setValue(double value);
15 };
```

Contents of NumberArray2.cpp

```
 1 #include <iostream>
 2 #include "NumberArray2.h"
 3 using namespace std;
 4
 5 //*****************************************
 6 //Copy constructor allocates a new        *
 7 //array and copies into it the entries    *
 8 //of the array in the other object.       *
 9 //*****************************************
10 NumberArray::NumberArray(NumberArray &obj)
11 {
12    arraySize = obj.arraySize;
13    aPtr = new double[arraySize];
14    for(int index = 0; index < arraySize; index++)
15      aPtr[index] = obj.aPtr[index];
16 }
17
18 //*****************************************
19 //Constructor allocates an array of the       *
20 //given size and sets all its entries to the  *
21 //given value.                                *
22 //*****************************************
23 NumberArray::NumberArray(int size, double value)
24 {
25    arraySize = size;
26    aPtr = new double[arraySize];
27    setValue(value);
28 }
29
30 //*****************************************************
31 //Sets all the entries of the array to the same value. *
32 //*****************************************************
33 void NumberArray::setValue(double value)
34 {
35    for(int index = 0; index < arraySize; index++)
36       aPtr[index] = value;
37 }
38
39 //*************************************
40 //Prints all the entries of the array. *
41 //*************************************
42 void NumberArray::print()
43 {
44    for(int index = 0; index < arraySize; index++)
45    cout << aPtr[index] << "  ";
46 }
```

(program continues)

Program 11-8 *(continued)*

Contents of `Pr11-8.cpp`

```
1  // This program demonstrates the use of copy constructors.
2  #include <iostream>
3  #include <iomanip>
4  #include "NumberArray2.h"
5
6  using namespace std;
7
8  int main()
9  {
10    NumberArray first(3, 10.5);
11
12    //Make second a copy of first object.
13    NumberArray second = first;
14
15    // Display the values of the two objects.
16    cout << setprecision(2) << fixed << showpoint;
17    cout << "Value stored in first object is ";
18    first.print();
19    cout << "\nValue stored in second object is ";
20    second.print();
21    cout <<  "\nOnly the value in second object will "
22         <<  "be changed.\n";
23
24    //Now change value stored in second object.
25    second.setValue(20.5);
26
27    // Display the values stored in the two objects.
28    cout << "Value stored in first object is ";
29    first.print();
30    cout << endl << "Value stored in second object is ";
31    second.print();
32    return 0;
33 }
```

Program Output

```
Value stored in first object is 10.50  10.50  10.50
Value stored in second object is 10.50  10.50  10.50
Only the value in second object will be changed.
Value stored in first object is 10.50  10.50  10.50
Value stored in second object is 20.50  20.50  20.50
```



NOTE: A copy constructor must have a single parameter that is a reference to the same class. Forgetting the `&` that identifies reference parameters will result in compiler errors.

The copy constructor is also automatically called by the compiler to create a copy of an object whenever an object is being passed by *value* in a function call. It is for this reason that the parameter to the copy constructor must be passed by reference; if it was passed by value when the constructor was called, then the constructor would immediately have to be

called again to create the copy to be passed by value, leading to an endless chain of calls to the constructor.

The copy constructor is also called to create a copy of an object to be returned from a function.

Using `const` Parameters

Because copy constructors are required to use reference parameters, they have access to their argument's data. Since the purpose of a copy constructor is to make a copy of the argument, there is no reason the constructor should modify the argument's data. With this in mind, it's a good idea to make a copy constructor's parameter constant by specifying the `const` key word in the parameter list. Here is an example:

```
NumberArray::NumberArray(const NumberArray &obj)
{
   arraySize = obj.arraySize;
   aPtr = new double[arraySize];
   for(int index = 0; index < arraySize; index++)
      aPtr[index] = obj.aPtr[index];
}
```

Invocation of Copy Constructors

Copy constructors are automatically called by the system whenever an object is being created by initializing it with another object of the same class. For example, the copy constructor for the `Rectangle` class is called for each of the following initialization statements:

```
Rectangle box(5, 10);
Rectangle b = box;    // Initialization statement
Rectangle b1(box);    // Initialization statement
```

Copy constructors are also automatically called when a function call receives a value parameter of the class type. For example, for a function of the form

```
void fun(Rectangle rect)
{
}
```

a call such as

```
fun(box);
```

will cause the `Rectangle` copy constructor to be called. Finally, copy constructors are automatically called whenever a function returns an object of the class by value. Thus, in the function

```
Rectangle makeRectangle()
{
    Rectangle rect(12, 3);
    return rect;

}
```

the copy constructor will be called when the return statement is executed. This is because the return statement must create a nonlocal copy of the object that will be available to

the caller after the function is done executing. To summarize, a class copy constructor is called when

- A variable is being initialized from an object of the same class
- A function is called with a value parameter of the class
- A function is returning a value that is an object of the class



NOTE: Copy constructors are not called when a parameter of the class is passed by reference or through a pointer, nor are they called when a function returns a reference or pointer to an object of the class.



Checkpoint

11.8 Briefly describe what is meant by memberwise assignment.

11.9 Describe two scenarios in which memberwise assignment occurs.

11.10 Describe a situation in which memberwise assignment should not be used.

11.11 When is a copy constructor called?

11.12 How does the compiler know that a member function is a copy constructor?

11.13 What action is performed by a class's default copy constructor?

11.6 Operator Overloading

CONCEPT: **C++ allows you to redefine how standard operators work when used with class objects.**



VideoNote
Operator Overloading

Overloading the = Operator

As we have seen, copy constructors are designed to solve problems that arise when an object containing a pointer is initialized with the data of another object of the same class using memberwise assignment. Similar problems arise in object assignment. For example, with the `NumberArray` class of the previous section, we may have a program that has defined two objects of that class:

```
NumberArray first(3, 10.5);
NumberArray second(5, 20.5);
```

Now, because C++ allows the assignment operator to be used with class objects, we may execute the statement

```
first = second;
```

if we want to set the first object to exactly the same value as the second. At this point, C++ will once again perform a memberwise copy from the second to the first object, leaving pointers in both objects pointing to the same memory.

Because the default object assignment encounters the same problem as the default copy constructor, we might think that a programmer-defined copy constructor can be used to solve the problem caused by the default assignment, but this is not so. Copy constructors only come into play when an object is being *initialized* at creation time. In particular, copy

constructors are not called in an *assignment*. To see the difference between initialization and assignment, suppose that the object `first` has already been created. Then the statement

```
NumberArray second = first;     // copy constructor called
```

which creates `second` and initializes it with the value of `first`, is an *initialization* and causes the copy constructor to be called to perform the initialization. However, the statement

```
second = first;                 // copy constructor not called
```

which assumes that both objects have previously been created, is an *assignment*, and therefore no constructor is invoked.

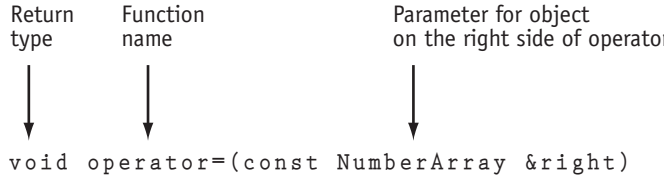
To address the problems that result from memberwise assignment of objects, we need to modify the behavior of the assignment operator so that it does something other than memberwise assignment when it is applied to objects of classes that have pointer members. In effect we are supplying a different version of the assignment operator to be used for objects of that class. In so doing, we say that we are *overloading* the assignment operator.

One way to overload the assignment operator for a given class is to define an *operator function* called `operator=` as a member function of the class. To do this for the `NumberArray` class, we would write the class declaration as follows:

```
class NumberArray
{
private:
   double *aPtr;
   int arraySize;
public:
   void operator=(const NumberArray &right); // Overloaded operator.
   NumberArray(const NumberArray &);
   NumberArray(int size, double value);
   ~NumberArray() { if (arraySize > 0) delete [ ] aPtr; }
   void print();
   void setValue(double value);
};
```

Let's take a look at the function header, or prototype, before we look at how the operator function itself is implemented. We break the header down into its main parts, as shown in Figure 11-4.

Figure 11-4



The name of the function is `operator=`. Since the operator function is an instance member of a class, it can only be called through an object of the class. The object of the class through which it is called is considered the left operand of the assignment operator, while the parameter

passed to the function is considered the right operand of the assignment operator. To illustrate, let us suppose that two objects, `left` and `right`, have been defined in a program:

```
NumberArray left(3,10.5);
NumberArray right(5, 20.5);
```

To assign the value of `right` to `left`, we call the member function `operator=` through the `left` object, and pass it the `right` object as parameter:

```
left.operator=(right);
```

While you can call operator functions this way, the compiler will also let you use the more conventional notation

```
left = right;
```



NOTE: Parameters to operator functions do not have to be passed by reference, nor do they have to be declared `const`. In this example we have used a reference parameter for efficiency reasons: Reference parameters avoid the overhead of copying the object being passed as parameter. The `const` is used to protect the parameter from change.

Let us now consider the implementation of the above operator function. The function starts out by deleting memory allocated to pointers in the object being assigned to, then makes a copy of the other object in pretty much the same way as the copy constructor for the class. Here is the code for the function (we have used the name `right` for the object that is the right parameter of the assignment, but any other name could have been used).

```
void NumberArray::operator=(const NumberArray &right)
{
   if (arraySize > 0) delete [] aPtr;
   arraySize = right.arraySize;
   aPtr = new double[arraySize];
   for (int index = 0; index < arraySize; index++)
      aPtr[index] = right.aPtr[index];
}
```

In general, the assignment operator should be overloaded whenever a nondefault copy constructor is used. In particular, classes allocating dynamic memory to a pointer member in any constructor should define both a copy constructor and an overloaded assignment operator. In addition, they should also provide a destructor to deallocate the storage allocated in the constructor.

The class `NumberArray`, with modifications to include both a copy constructor and an overloaded assignment operator, is demonstrated in Program 11-9.

Program 11-9

Contents of `overload.h`

```
1 #include <iostream>
2 using namespace std;
3
4 class NumberArray
5 {
6 private:
```

(program continues)

Program 11-9 *(continued)*

```
  double *aPtr;
  int arraySize;
public:
  // Overloaded operator function.
  void operator=(const NumberArray &right);

  // Constructors and other member functions.
  NumberArray(const NumberArray &);
  NumberArray(int size, double value);
  ~NumberArray() { if (arraySize > 0) delete [ ] aPtr; }
  void print();
  void setValue(double value);
};
```

Contents of `overload.cpp`

```
#include <iostream>
#include "overload.h"
using namespace std;

//*************************************************
//The overloaded operator function for assignment.  *
//*************************************************
void NumberArray::operator=(const NumberArray &right)
{
   if (arraySize > 0) delete [] aPtr;
   arraySize = right.arraySize;
   aPtr = new double[arraySize];
   for (int index = 0; index < arraySize; index++)
    aPtr[index] = right.aPtr[index];
}

//*********************************************
//Copy constructor.                           *
//*********************************************
NumberArray::NumberArray(const NumberArray &obj)
{
   arraySize = obj.arraySize;
   aPtr = new double[arraySize];
   for(int index = 0; index < arraySize; index++)
     aPtr[index] = obj.aPtr[index];
}

//*********************************************
//Constructor.                                *
//*********************************************
NumberArray::NumberArray(int size1, double value)
{
   arraySize = size1;
   aPtr = new double[arraySize];
   setValue(value);
}

```

(program continues)

Program 11-9 *(continued)*

```
38 //*************************************************
39 //Sets the value stored in all entries of the array. *
40 //*************************************************
41 void NumberArray::setValue(double value)
42 {
43    for(int index = 0; index < arraySize; index++)
44       aPtr[index] = value;
45 }
46
47 //*************************************
48 //Print out all entries in the array.   *
49 //*************************************
50 void NumberArray::print()
51 {
52    for(int index = 0; index < arraySize; index++)
53      cout << aPtr[index] << "  ";
54 }
```

Contents of `Pr11-9.cpp`

```
1  // This program demonstrates overloading of
2  // the assignment operator.
3  #include <iostream>
4  #include <iomanip>
5  #include "overload.h"
6  using namespace std;
7
8  int main()
9  {
10    NumberArray first(3, 10.5);
11    NumberArray second(5, 20.5);
12
13    // Display the values of the two objects.
14    cout << setprecision(2) << fixed << showpoint;
15    cout << "First object's data is ";
16    first.print();
17    cout << endl << "Second object's data is ";
18    second.print();
19
20    // Call the overloaded operator.
21    cout << "\nNow we will assign the second object "
22         << "to the first." << endl;
23    first = second;
24
25    // Display the new values of the two objects.
26    cout << "First object's data is ";
27    first.print();
28    cout << endl << "The second object's data is ";
29    second.print();
30
31    return 0;
32 }
```

(program continues)

Program 11-9 *(continued)*

Program Output

```
First object's data is 10.50  10.50  10.50
Second object's data is 20.50  20.50  20.50  20.50  20.50
Now we will assign the second object to the first.
First object's data is 20.50  20.50  20.50  20.50  20.50
The second object's data is 20.50  20.50  20.50  20.50  20.50
```

The = Operator's Return Value

There is only one problem with the overloaded = operator shown in Program 11-8: It has a `void` return type. C++'s built-in assignment operator allows multiple assignment statements such as

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

Multiple assignment statements work because the built-in assignment operator is implemented so that it returns the value of its left operand *after* the assignment has been performed. Thus in this statement, the expression `b = c` causes `c` to be assigned to `b` and then returns the value of `b`. The return value is then stored in `a`.

To make an overloaded assignment operator behave similarly, we must redefine the operator function so that it also returns the value of its left operand after the assignment has been performed. In particular, we need to declare the operator function to have a return type of the same type as the class. This is shown in our final modification of the `NumberArray` class:

```
class NumberArray
{
private:
   double *aPtr;
   int arraySize;
public:
   NumberArray operator=(const NumberArray &right);
   NumberArray(const NumberArray &);
   NumberArray(int size, double value);
   ~NumberArray() { if (arraySize > 0) delete [] aPtr; }
   void print();
   void setValue(double value);
};
```

The only modification we need to make to the assignment operator function is to add a statement at the very end returning the value of its left operand. Since the assignment is equivalent to the statement

```
left.operator=(right);
```

returning the value of the left operand is accomplished by the operator function returning the value of the object through which it is called. Recall that C++ makes available to each call of a non-static member function, the address of the object through which the call is being made, and that the address of that object is accessed through the pointer `this`. The value of the object itself can be obtained and returned by dereferencing the `this` pointer:

```
return *this;
```

The code for the modified assignment operator function is

```
NumberArray NumberArray::operator=(const NumberArray &right)
{
   if (arraySize > 0)  delete [] aPtr;
   arraySize = right.arraySize;
   aPtr = new double[arraySize];
   for (int index = 0; index < arraySize; index++)
      aPtr[index] = right.aPtr[index];
   return *this;
}
```

Overloading Other Operators

C++ allows the programmer to overload other operators besides assignment. There are many times when it is natural to overload some of C++'s built-in operators to make them work with classes that the programmer has defined. For example, assume that a class named `Date` exists, and that objects of the `Date` class hold the day, month, and year in member variables. Suppose the `Date` class has a member function named `add`. The `add` member function adds a number of days to the date and adjusts the member variables if the date goes to another month or year. For example, the following statement adds five days to the date stored in the `today` object:

```
today.add(5);
```

Although it might be obvious that the statement is adding five days to the date stored in `today`, the use of an operator might be more intuitive. For example, look at the following statement:

```
today += 5;
```

This statement uses the standard += operator to add 5 to `today`. This behavior does not happen automatically, however. The += operator must be overloaded for this action to occur. In this section, you will learn to overload many of C++'s operators to perform specialized operations on class objects.



NOTE: You have already experienced the behavior of an overloaded operator. The / operator performs two types of division: floating-point and integer. If one of operator's operands is a floating-point type, the result will be a floating-point value. If both of the / operator's operands are integers, however, a different behavior occurs: the result is an integer, and the fractional part is thrown away.

Some General Issues of Operator Overloading

Now that you have had an introduction to operator overloading, let's look at some of the general issues involved in this programming technique.

First, you can change an operator's entire meaning, if that's what you wish to do. There is nothing to prevent you from changing the = symbol from an assignment operator to a "display" operator. For instance, the following class does just that:

```
class Weird
{
private:
   int value;
```

```
public:
   Weird(int v)
      {value = v; }
   void operator=(const Weird &right)
      { cout << right.value << endl; }
};
```

Although the `operator=` function overloads the assignment operator, the function doesn't perform an assignment. All the overloaded operator does is display the contents of `right.value`. Consider the following program segment:

```
Weird a(5), b(10);
a = b;
```

Although the statement `a = b` looks like an assignment statement, it actually causes the contents of `b`'s `value` member to be displayed on the screen:

```
10
```

Another operator overloading issue is that you cannot change the number of operands taken by an operator. The = symbol must always be a binary operator. Likewise, ++ and -- must always be unary operators.

The last issue is that although you may overload most of the C++ operators, you cannot overload all of them. Table 11-1 shows all of the C++ operators that may be overloaded.



NOTE: Some of the operators in Table 11-1 are beyond the scope of this book and are not covered.

Table 11-1 Operators That Can Be Overloaded

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

The only operators that cannot be overloaded are

```
?:   .   .*   ::   sizeof
```

Approaches to Operator Overloading

There are two approaches you can take to overload an operator:

1. *Make the overloaded operator a member function of the class.* This allows the operator function access to private members of the class. It also allows the function to use the implicit `this` pointer parameter to access the calling object.
2. *Make the overloaded member function a separate, stand-alone function.* When overloaded in this manner, the operator function must be declared a friend of the class to have access to the private members of the class.

Some operators, such as the stream input and output operators >> and <<, must be overloaded as stand-alone functions. Other operators may be overloaded either as member functions or stand-alone functions. Consider a class

```
class Length
{
private:
    int len_inches;
public:
    Length(int feet, int inches)
    {
        setLength(feet, inches);
    }
    Length(int inches){ len_inches = inches; }
    int getFeet(){ return len_inches / 12; }
    int getInches() { return len_inches % 12; }
    void setLength(int feet, int inches)
    {
        len_inches = 12 *feet + inches;
    }

};
```

designed to represent length measurements. The class internally represents the length of an item in inches, but allows its clients to specify measurements in feet and inches via a `setLength()` function. The class also provides member functions `getFeet()` and `getInches()` to allow the feet and inch components of a measurement to be separately retrieved.

Overloading the Arithmetic and Relational Operators

Clients of the class must be able to add and subtract measurements. In addition, they should be able to compare two measurements to see if they are equal, or if one of them is less or greater than the other. We will provide all these capabilities by overloading the operators +, -, <, and == as stand-alone functions. We start by adding the following declarations to the `Length` class:

```
friend Length operator+(Length a, Length b);
friend Length operator-(Length a, Length b);
friend bool operator<(Length a, Length b);
friend bool operator==(Length a, Length b);
```

To see how we arrive at these declarations, consider the *addition* and *less-than* operators. Addition needs to take two `Length` objects `a` and `b` as parameters and produce a third `Length` object that is the sum of `a` and `b`. Similarly, the less-than operator needs to take two `Length` objects as parameters and return a Boolean value.

To see how to write the code that implements these functions, consider again the addition operator. Given two input parameters `a` and `b`, it needs to return a `Length` object whose `len_inches` member is the sum of the `len_inches` members of `a` and `b`. This can be done by writing

```
Length operator+(Length a, Length b)
{
    Length result(a.len_inches + b.len_inches);
    return result;
}
```

or more succinctly:

```
Length operator+(Length a, Length b)
{
    return Length(a.len_inches + b.len_inches);
}
```

We can reason in a similar manner to work out the definitions of the operator functions –, <, and ==. Here is a complete program showing the class and its overloaded operators and illustrating their use.

Contents of `Length.h`

```
#ifndef _LENGTH_H
#define _LENGTH_H
#include <iostream>
using namespace std;

class Length
{
private:
    int len_inches;
public:
    Length(int feet, int inches)
    {
        setLength(feet, inches);
    }
    Length(int inches){ len_inches = inches; }
    int getFeet(){ return len_inches / 12; }
    int getInches() { return len_inches % 12; }
    void setLength(int feet, int inches)
    {
        len_inches = 12 *feet + inches;
    }
    friend Length operator+(Length a, Length b);
    friend Length operator-(Length a, Length b);
    friend bool operator< (Length a, Length b);
    friend bool operator== (Length a, Length b);
};
#endif
```

Contents of `Length.cpp`

```
#include "Length.h"

//************************************
// Overloaded operator +              *
//************************************
Length operator+(Length a, Length b)
{
    return Length(a.len_inches + b.len_inches);
}

//************************************
// Overloaded  operator -             *
//************************************
Length operator-(Length a, Length b)
{
```

```
    return Length(a.len_inches - b.len_inches);
}

//***********************************
// Overloaded operator ==           *
//***********************************
bool operator==(Length a, Length b)
{
    return a.len_inches == b.len_inches;
}

//***********************************
// Overloaded operator <            *
//***********************************
bool operator<(Length a, Length b)
{
    return a.len_inches < b.len_inches;
}
```

Program 11-10

```
// This program demonstrates the Length class's overloaded
// +, -, ==, and < operators.
#include <iostream>
#include "Length.h"
using namespace std;

int main()
{
    Length first(0), second(0), third(0);
    int f, i;
    cout << "Enter a distance in feet and inches: ";
    cin  >> f >> i;
    first.setLength(f, i);
    cout << "Enter another distance in feet and inches: ";
    cin  >> f >> i;
    second.setLength(f, i);

    // Test the + and - operators
    third = first + second;
    cout << "first + second = ";
    cout << third.getFeet() << " feet, ";
    cout << third.getInches() << " inches.\n";
    third = first - second;
    cout << "first - second = ";
    cout << third.getFeet() << " feet, ";
    cout << third.getInches() << " inches.\n";

    // Test the relational operators
    cout << "first == second = ";
    if (first == second) cout << "true"; else cout << "false";
    cout << "\n";
```

(program continues)

Program 11-10 *(continued)*

```
32      cout << "first < second = ";
33      if (first < second) cout << "true"; else cout << "false";
34      cout << "\n";
35
36      return 0;
37  }
```

Program Output with Example Input Shown in Bold

```
Enter a distance in feet and inches: 6 5[Enter]
Enter another distance in feet and inches: 3 10[Enter]
first + second = 10 feet, 3 inches.
first - second = 2 feet, 7 inches.
first == second = false
first < second = false
```

Choosing Between Stand-Alone and Member-Function Operators

Given the stand-alone overloads we have written, the code

```
Length a(4, 2), b(1, 8), c(0);
c = a + b;
```

is interpreted by the compiler as being

```
Length a(4, 2), b(1, 8), c(0);
c = operator+(a, b);
```

The compiler allows the programmer to use the friendly infix notation. Internally, however, it sees the operator as just an ordinary function whose name is `operator+`. This has an implication that is not immediately obvious. The statement

```
c = 2 + a;
```

is equivalent to

```
c = operator+(2, b);
```

Both of these statements compile and execute correctly because the *convert constructor* of the `Length` class is able to create a `Length` object out of the integer parameter 2. You will learn about convert constructors in Section 11.8.

We could just as easily have overloaded the arithmetic and relational operators as member functions. Here is how to do so for the addition operator. First, modify the in-class declaration to make the operator a member function:

```
class Length
{
private:
    int len_inches;
public:
    // Modified declaration of operator+
    Length operator+(Length b);
    // Rest of class not shown
};
```

Notice that the operator is now declared as taking a single operator of type `Length`. This is because as a member function, the operator is automatically passed a `Length` object through the implicit parameter `this`. When we write

```
Length a(4, 2), b(1, 8), c(0);
c = a + b;
```

The compiler sees this as

```
Length a(4, 2), b(1, 8), c(0);
c = a.operator+(b);
```

When you write `a + b`, the left operand of the overloaded + operator becomes the object through which the member function is called, and the right operand becomes the explicit parameter. With these changes, the body of the operator is written as follows:

```
Length Length::operator+(Length b)
{
    return Length(this->len_inches + b.len_inches);
}
```

To sum up, the addition operator (as well as other arithmetic and relational operators) can be overloaded equally well as member functions or as stand-alone functions. It is generally better to overload binary operators that take parameters of the same type as stand-alone functions. This is because, unlike stand-alone operator overloading, member-function overloading introduces an artificial distinction between the two parameters by making the left parameter implicit. This allows convert constructors to apply to the right parameter but not to the left, creating situations where changing the order of parameters causes a compiler error in an otherwise correct program:

```
Length a(4, 2), c(0);
c = a + 2;  // Compiles, equivalent to c = a.operator+(2)
c = 2 + a;  // Does not compile: equivalent to c = 2.operator+(a);
```

Overloading the Prefix ++ Operator

We want to overload the prefix operator for the `Length` class so that the expression `++b` increments the object `b` by adding 1 inch to its length and returns the resulting object. We overload this operator as a member function. This makes its single parameter implicit, so the overloaded operator needs no parameters. Here is the portion of the `Length` class that shows the operator declaration:

```
class Length
{
private:
    int len_inches;
public:
    // Declaration of prefix ++
    Length operator++();
    // Rest of class not shown
};
```

Here is the implementation of the operator—it increases the number of inches by 1 and returns the modified object:

```
Length Length::operator++()
{
   len_inches ++;
   return *this;
}
```

Given this overload, the user-friendly notation `++b` is equivalent to the call `b.operator++()`. Either notation may be used in your program.

Overloading the Postfix ++ Operator

The postfix increment operator `b++` also increments the length of `b`, but differs from the prefix version in that it returns the value that the object had *prior* to being incremented. Overloading the postfix operator is only slightly different from overloading the prefix version. Here is the function that overloads the postfix operator for the `Length` class:

```
Length Length::operator++(int)
{
    Length temp = *this;
    len_inches ++;
    return temp;
}
```

The first difference you will notice is that the function has a *dummy parameter* of type `int` that is never used in the body of the function. This is a convention that tells the compiler that the increment operator is being overloaded in postfix mode. The second difference is the use of a temporary local variable `temp` to capture the value of the object before it is incremented. This value is saved and is later returned by the function.

Overloading the Stream Insertion and Extraction Operators

Overloading the stream insertion operator `<<` is convenient because it allows values of objects to be converted into text and output to `cout`, to a file object, or to any object of a class that derives from `ostream`. In the presence of appropriate overloads, the statements

```
Length b(4, 8), c(2, 5);
cout << b;
cout << b + c;
```

appear to the compiler as

```
Length b(4, 8), c(2, 5);
operator<<(cout, b);
operator(cout, b + c);
```

This equivalence has the following implications:

1. The overloaded operator `<<` takes two parameters, the first of which is an `ostream` object and the second of which is an object of the class for which the operator is being overloaded. For the `Length` class, the prototype would be `operator<<(ostream &strm, Length a)`.

2. To allow expressions (such as `b + c`) in the second parameter, the second parameter should be passed by value. The first parameter should be passed by reference because `ostream` parameters should never be passed by value.

In addition, the stream insertion operator should return its stream parameter so that several output expressions can be chained together, as in

```
Length b(4, 8), c(2, 5);
cout << b << "  " << b + c;
```

Putting all of this together, we see that the stream insertion operator should be written as

```
ostream &operator<<(ostream& out, Length a)
{
    out << a.getFeet() << " feet, " << a.getInches() << " inches";
    return out;
}
```

Overloading the stream output operator is useful because it allows for the various fields of a complex class to be labeled during output.

Overloading the stream input operator is similar, except that the class parameter signifying the object to be read into must be passed by reference. Thus the header for the stream input operator looks like this:

```
istream &operator>>(istream &in, Length &a);
```

The full implementation of this function can be found in lines 3–18 of the following listing of `Length1.cpp`. At first glance, the function appears to be useful in that it relieves the programmer of the necessity of issuing prompts for the different parts of the object when the user is entering data at the screen and keyboard. Notice, however, that the prompts become an irritating distraction if the operator is being used to read `Length` objects from a non-keyboard source such as a file or network connection.

Contents of `Length1.h`

```
 1 #ifndef _LENGTH1_H
 2 #define _LENGTH1_H
 3 #include <iostream>
 4 using namespace std;
 5
 6 class Length
 7 {
 8 private:
 9     int len_inches;
10 public:
11     Length(int feet, int inches)
12     {
13         setLength(feet, inches);
14     }
15     Length(int inches){ len_inches = inches; }
16     int getFeet(){ return len_inches / 12; }
17     int getInches() { return len_inches % 12; }
18     void setLength(int feet, int inches)
```

```
    {
        len_inches = 12 *feet + inches;
    }
    // Overloaded arithmetic and relational operators
    friend Length operator+(Length a, Length b);
    friend Length operator-(Length a, Length b);
    friend bool operator<(Length a, Length b);
    friend bool operator==(Length a, Length b);
    Length operator++();
    Length operator++(int);

    // Overloaded stream input and output operators
    friend ostream &operator<<(ostream &out, Length a);
    friend istream &operator>>(istream &in, Length &a);
};
#endif
```

Contents of `Length1.cpp`

```
#include "Length1.h"

//*********************************************
// Overloaded stream extraction operator >>    *
//*********************************************
istream &operator>>(istream &in, Length &a)
{
    // Prompt for and read the object data
    int feet, inches;
    cout << "Enter feet: ";
    in >> feet;
    cout << "Enter inches: ";
    in >> inches;

    // Modify the object a with the data and return
    a.setLength(feet, inches);
    return in;
}

//*********************************************
// Overloaded stream insertion operator <<    *
//*********************************************
ostream &operator<<(ostream& out, Length a)
{
    out << a.getFeet() << " feet, " << a.getInches() << " inches";
    return out;
}

//***********************************
// Overloaded prefix ++ operator    *
//***********************************
Length Length::operator++()
{
    len_inches ++;
    return *this;
}
```

```
37
38 //***********************************
39 // Overloaded postfix ++ operator   *
40 //***********************************
41 Length Length::operator++(int)
42 {
43     Length temp = *this;
44     len_inches ++;
45     return temp;
46 }
47
48 //***********************************
49 // Overloaded operator -             *
50 //***********************************
51 Length operator+(Length a, Length b)
52 {
53     return Length(a.len_inches + b.len_inches);
54 }
55
56 //***********************************
57 // Overloaded  operator -            *
58 //***********************************
59 Length operator-(Length a, Length b)
60 {
61     return Length(a.len_inches - b.len_inches);
62 }
63
64 //***********************************
65 // Overloaded operator ==            *
66 //***********************************
67 bool operator==(Length a, Length b)
68 {
69     return a.len_inches == b.len_inches;
70 }
71
72 //***********************************
73 // Overloaded operator <             *
74 //***********************************
75 bool operator<(Length a, Length b)
76 {
77     return a.len_inches < b.len_inches;
78 }
```

Program 11-11

```
1 // This program demonstrates the Length class's overloaded
2 // prefix ++, postfix ++, and stream operators.
3 #include <iostream>
4 #include "Length1.h"
5 using namespace std;
6
7 int main()
```

(program continues)

Program 11-11 *(continued)*

```
{
    Length first(0), second(1, 9), c(0);

    cout << "Demonstrating prefix ++ operator and output operator.\n";
    for (int count = 0; count < 4; count++)
    {
        first = ++second;
        cout << "First: " << first <<  ". Second: " << second << ".\n";
    }
    cout << "\nDemonstrating postfix ++ operator and output operator.\n";
    for (int count = 0; count < 4; count++)
    {
        first = second++;
        cout << "First: " << first <<  ". Second: " << second << ".\n";
    }

    cout << "\nDemonstrating input and output operators.\n";
    cin >> c;
    cout << "You entered " << c << "." << endl;
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Demonstrating prefix ++ operator and output operator.
First: 1 feet, 10 inches. Second: 1 feet, 10 inches.
First: 1 feet, 11 inches. Second: 1 feet, 11 inches.
First: 2 feet, 0 inches. Second: 2 feet, 0 inches.
First: 2 feet, 1 inches. Second: 2 feet, 1 inches.

Demonstrating postfix ++ operator and output operator.
First: 2 feet, 1 inches. Second: 2 feet, 2 inches.
First: 2 feet, 2 inches. Second: 2 feet, 3 inches.
First: 2 feet, 3 inches. Second: 2 feet, 4 inches.
First: 2 feet, 4 inches. Second: 2 feet, 5 inches.

Demonstrating input and output operators.
Enter feet: 3[Enter]
Enter inches: 4[Enter]
You entered 3 feet, 4 inches.
```

Overloading the [] Operator

In addition to the traditional operators, C++ allows you to change the way the `[]` symbols work. This gives you the ability to write classes that have array-like behaviors. For example, the `string` class overloads the `[]` operator so you can access the individual characters stored in `string` class objects. Assume the following definition exists in a program:

```
string name = "William";
```

The first character in the string, `W`, is stored at `name[0]`, so the following statement will display `W` on the screen.

```
cout << name[0];
```

Program 11-12 further demonstrates the `string` class's overloaded `[]` operator.

Program 11-12

```
1  // This program demonstrates the string class's
2  // overloaded [] operator.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string name = "William";
10
11    cout << "Here are the letters in your name: \n";
12    for (int x=0; x < name.length(); x++)
13      cout << name[x] << ' ';
14    cout << "\nEnter a character and press Enter: ";
15    cin >> name[2];
16    cout << "Now, here are the letters in your name:\n";
17    for (int x=0; x < name.length(); x++)
18      cout << name[x] << ' ';
19
20    return 0;
21 }
```

Program Output with Output Shown in Bold

```
Here are the letters in your name:
W i l l i a m
Enter a character and press Enter: x[Enter]
Now, here are the letters in your name:
W i x l i a m
```

You can use the `overloaded[]`operator to create an array class, like the following one. The class behaves like a regular array but performs the bounds checking that C++ lacks. It also has several other enhancements over regular integer arrays.

```
class IntArray
{
   private:
      int *aptr;
      int arraySize;
      void subError();            // Handles subscripts out of range
   public:
      IntArray(int);              // Constructor
      IntArray(const intArray &); // Copy constructor
      ~IntArray();                // Destructor
      int size()
          { return arraySize; }
      int &operator[](int);       // Overloaded [] operator
};
```

Before focusing on the overloaded operator, let's look at the constructors and the destructor. The code for the first constructor is

```
IntArray::IntArray(int s)
{
   arraySize = s;
   aptr = new int [s];
   for (int count = 0; count < size; count++)
      *(aptr + count) = 0;
}
```

When an instance of the class is defined, the number of elements the array is to have is passed into the constructor's parameter `s`. This value is copied to the `arraySize` member, then used to dynamically allocate enough memory for the array. The constructor's final step is to store zeros in all of the array's elements:

```
for (int count = 0; count < size; count++)
   *(aptr + count) = 0;
```

The class also has a copy constructor, which is used when a class object is initialized with another object's data:

```
IntArray::IntArray(const IntArray &obj)
{
   arraySize = obj.arraySize;
   aptr = new int [arraySize];
   for(int count = 0; count < arraySize; count++)
      *(aptr + count) = *(obj.aptr + count);
}
```

A reference to the initializing object is passed into the parameter `obj`. Once the memory is successfully allocated for the array, the constructor copies all the values in `obj`'s array into the calling object's array.

The destructor simply frees the memory allocated by the class's constructors. First, however, it checks the value in `arraySize` to be sure the array has at least one element:

```
IntArray::~IntArray()
{
   if (arraySize > 0)
      delete [] aptr;
}
```

The [] operator is overloaded similarly to other operators. Here is the definition of the `operator[]` function for the `IntArray` class:

```
int &IntArray::operator[](int sub)
{
   if (sub < 0 || sub >= arraySize)
      subError();
   return aptr[sub];
}
```

The `operator[]` function can only have a single parameter. The one shown here uses an integer parameter. This parameter holds the value placed inside the brackets in an expression. For example, if `table` is an `IntArray` object, the number 12 will be passed into the `sub` parameter in the following statement:

```
cout << table[12];
```

Inside the function, the value in the `sub` parameter is tested by the following `if` statement:

```
if (sub < 0 || sub >= arraySize)
   subError();
```

This statement determines whether `sub` is within the range of the array's subscripts. If `sub` is less than zero or greater than or equal to `arraySize`, it's not a valid subscript, so the `subError` function is called. If `sub` is within range, the function uses it as an offset into the array, and returns a reference to the value stored at that location.

One critically important aspect of the function shown is its return type. It's crucial that the function not simply return an integer, but a *reference* to an integer. The reason for this is that expressions such as the following must be possible:

```
table[5] = 27;
```

Remember, the built-in = operator requires the object on its left to be an lvalue. An lvalue must represent a modifiable memory location, such as a variable. The integer return value of a function is not an lvalue. If the `operator[]` function merely returns an integer, it cannot be used to create expressions placed on the left side of an assignment operator.

A reference to an integer, however, is an lvalue. If the `operator[]` function returns a reference, the statement above causes the `operator[]` function to be called with 5 being passed as its argument. Assuming 5 is within range, the function returns a reference to the integer stored at `(aptr + 5)`. In essence, the statement is equivalent to

```
*(aptr + 5) = 27;
```

Because the `operator[]` function returns actual integers stored in the array, it is not necessary for math or relational operators to be overloaded. Even the stream operators << and >> will work just as they are with the `IntArray` class.

Here is the complete listing of `intarray.h` and `intarray.cpp`:

Contents of `intarray.h`

```
1  #ifndef INTARRAY_H
2  #define INTARRAY_H
3  #include <iostream>
4  using namespace std;
5
6  class IntArray
7  {
8  private:
9     int *aptr;
10    int arraySize;
11    void subError();   // Handles subscripts out of range
12 public:
13    IntArray(int);              // Constructor
14    IntArray(const IntArray &); // Copy constructor
15    ~IntArray();                // Destructor
16    int size(){ return arraySize; }
17    int &operator[](int);       // Overloaded [] operator
18 };
19 #endif
```

Contents of intarray.cpp

```
#include "intarray.h"

//***************************************************
// Constructor for IntArray class. Sets the size of  *
// the array and allocates memory for it.            *
//***************************************************
IntArray::IntArray(int s)
{
   arraySize = s;
   aptr = new int [s];
   for (int count = 0; count < arraySize; count++)
      *(aptr + count) = 0;
}

//***************************************************
// Copy constructor for IntArray class.              *
//***************************************************
IntArray::IntArray(const IntArray &obj)
{
   arraySize = obj.arraySize;
   aptr = new int [arraySize];
   for(int count = 0; count < arraySize; count++)
      *(aptr + count) = *(obj.aptr + count);
}

 //***************************************************
// Destructor for IntArray class.                    *
//***************************************************
IntArray::~IntArray()
{
   if (arraySize > 0)
      delete [] aptr;
}

//***************************************************
// subError function. Displays an error message and   *
// exits the program when a subscript is out of range.*
//***************************************************
void IntArray::subError()
{
   cout << "ERROR: Subscript out of range.\n";
   exit(0);
}

//***************************************************
// Overloaded [] operator. The argument is a subscript *
// This function returns a reference to the element    *
// in the array indexed by the subscript.              *
//***************************************************
int &IntArray::operator[](int sub)
```

```
{
   if (sub < 0 || sub >= arraySize)
      subError();
   return aptr[sub];
}
```

Program 11-13 demonstrates how the class works.

Program 11-13

```
// This program demonstrates a class that behaves
// like an array.
#include <iostream>
#include "intarray.h"
using namespace std;

int main()
{
   IntArray table(10);

   // Store values in the array.
   for (int x = 0; x < table.size(); x++)
      table[x] = (x * 2);

   // Display the values in the array.
   for (int x = 0; x < table.size(); x++)
      cout << table[x] << " ";
   cout << endl;

   // Use the built-in + operator on array elements.
   for (int x = 0; x < table.size(); x++)
      table[x] = table[x] + 5;

   // Display the values in the array.
   for (int x = 0; x < table.size(); x++)
      cout << table[x] << " ";
   cout << endl;

   // Use the built-in ++ operator on array elements.
   for (int x = 0; x < table.size(); x++)
      table[x]++;

   // Display the values in the array.
   for (int x = 0; x < table.size(); x++)
      cout << table[x] << " ";

   cout << endl;
   return 0;
}
```

Program Output

```
0 2 4 6 8 10 12 14 16 18
5 7 9 11 13 15 17 19 21 23
6 8 10 12 14 16 18 20 22 24
```

Program 11-14 demonstrates the `IntArray` class's bounds-checking capability.

Program 11-14

```
1  // This program demonstrates the bounds-checking
2  // capabilities of the IntArray class.
3  #include <iostream>
4  #include "intarray.h"
5  using namespace std;
6
7  int main()
8  {
9     IntArray table(10);
10
11    // Store values in the array.
12    for (int x = 0; x < table.size(); x++)
13       table[x] = x;
14
15    // Display the values in the array.
16    for (int x = 0; x < table.size(); x++)
17       cout << table[x] << " ";
18    cout << endl;
19
20    cout << "Attempting to store outside the array bounds:\n";
21    table[table.size()] = 0;
22
23    return 0;
24 }
```

Program Output

```
0 1 2 3 4 5 6 7 8 9
Attempting to store outside the array bounds:
ERROR: Subscript out of range.
```



Checkpoint

11.14 Assume there is a class named `Pet`. Write the prototype for a member function of `Pet` that overloads the = operator.

11.15 Assume that `dog` and `cat` are instances of the `Pet` class, which has overloaded the = operator. Rewrite the following statement so it appears in function call notation instead of operator notation:

```
dog = cat;
```

11.16 What is the disadvantage of an overloaded = operator returning `void`?

11.17 Describe the purpose of the `this` pointer.

11.18 The `this` pointer is automatically passed to what type of functions?

11.19 Assume there is a class named `Animal`, which overloads the = and + operators. In the following statement, assume `cat`, `tiger`, and `wildcat` are all instances of the `Animal` class:

```
wildcat = cat + tiger;
```

Of the three objects, `wildcat`, `cat`, and `tiger`, which is calling the `operator+` function? Which object is passed as an argument into the function?

11.20 What does the use of a dummy parameter in a unary operator function indicate to the compiler?

11.21 Describe the values that should be returned from functions that overload relational operators.

11.22 What is the advantage of overloading the << and >> operators?

11.23 What type of object should an overloaded << operator function return?

11.24 What type of object should an overloaded >> operator function return?

11.25 If an overloaded << or >> operator accesses a private member of a class, what must be done in that class's declaration?

11.26 Assume the class `NumList` has overloaded the [] operator. In the expression below, `list1` is an instance of the `NumList` class:

```
list1[25]
```

Rewrite this expression to explicitly call the function that overloads the [] operator.

11.27 When overloading a binary operator such as + or – as an instance member function of a class, what object is passed into the operator function's parameter?

11.28 Explain why overloaded prefix and postfix ++ and -- operator functions should return a value.

11.29 How does C++ tell the difference between an overloaded prefix and postfix ++ or -- operator function?

11.30 Overload the function call `operator () (int i, int j)` for the `IntArray` class of Program 11-13 to return the sum of all array entries in positions `i` through `j`.

11.7 Type Conversion Operators

CONCEPT: Special operator functions may be written to convert a class object to any other type.

As you've already seen, operator functions allow classes to work more like built-in data types. Another capability that operator functions can give classes is automatic type conversion.

Data type conversion happens "behind the scenes" with the built-in data types. For instance, suppose a program uses the following variables:

```
int i;
double d;
```

The following statement automatically converts the value in `i` to a double and stores it in `d`:

```
d = i;
```

Likewise, the following statement converts the value in `d` to an integer (truncating the fractional part) and stores it in `i`:

```
i = d;
```

The same functionality can also be given to class objects. For example, assuming `distance` is a `Length` object and `d` is a `double`, the following statement would conveniently store distance into a floating-point number stored in `d`, if `Length` is properly written:

```
d = distance;
```

To be able to use a statement such as this, an operator function must be written to perform the conversion. Here is an operator function for converting a `Length` object to a `double`:

```
Length::operator double()
{
   return len_inches /12 + (len_inches %12) / 12.0;
}
```

This function computes the real decimal equivalent of a length measurement in feet. For example, a measurement of 4 feet 6 inches would be converted to the real number 4.5.



NOTE: No return type is specified in the function header because the return type is inferred from the name of the operator function. Also, because the function is a member function, it operates on the calling object and requires no other parameters.

Program 11-15 demonstrates a modified version of the `Length` class with both a `double` and an `int` conversion operator. The `int` operator simply returns the number of inches of the `Length` object.

Contents of `Length2.h`

```
#ifndef _LENGTH1_H
#define _LENGTH1_H
#include <iostream>
using namespace std;

class Length
{
private:
  int len_inches;
public:
  Length(int feet, int inches)
  {
      setLength(feet, inches);
  }
  Length(int inches){ len_inches = inches; }
  int getFeet(){ return len_inches / 12; }
  int getInches() { return len_inches % 12; }
  void setLength(int feet, int inches)
  {
      len_inches = 12 *feet + inches;
  }
  // Type conversion operators
  operator double();
  operator int() { return len_inches;  }

  // Overloaded stream output operator
  friend ostream &operator<<(ostream &out, Length a);
};
#endif
```

Contents of Length2.cpp

```
#include "Length2.h"

//*********************************************
// Operatord double converts Length to a double *
//*********************************************
Length::operator double()
{
   return len_inches /12 + (len_inches %12) / 12.0;
}

//*****************************************
// Overloaded stream insertion operator <<  *
//*****************************************
ostream &operator<<(ostream& out, Length a)
{
   out << a.getFeet() << " feet, " << a.getInches() << " inches";
   return out;
}
```

Program 11-15

```
// This program demonstrates the type conversion operators for
// the Length class.
#include "Length2.h"

#include <iostream>
#include <string>
using namespace std;

int main()
{
   Length distance(0);
   double feet;
   int inches;
   distance.setLength(4, 6);
   cout << "The Length object is " << distance <<  "." << endl;

   // Convert and print
   feet = distance;
   inches = distance;
   cout << "The Length object measures " << feet << " feet." << endl;
   cout << "The Length object measures " << inches << " inches."
        << endl;
   return 0;
 }
```

Program Output

```
The Length object is 4 feet, 6 inches.
The Length object measures 4.5 feet.
The Length object measures 54 inches.
```

11.8 Convert Constructors

CONCEPT: **In addition to providing a means for the creation of objects, convert constructors provide a way for the compiler to convert a value of a given type to an object of the class.**

A constructor that takes a single parameter of a type other than its class type can be regarded as converting its parameter into an object of its class. Such a constructor is called a *convert constructor.*

In addition to the function of creating objects of its class, a convert constructor provides the compiler with a way of performing implicit type conversions. Such type conversions will be performed by the compiler whenever a value of the constructor's parameter type is given where a value of the class type is expected.

As a simple example, consider the class

```
class IntClass
{
private:
   int value;
public:
   // Convert constructor from int
   IntClass(int intValue)
   {
      value = intValue;
   }
   int getValue(){ return value; }
};
```

Since the constructor `IntClass(int)` takes a single parameter of a type other than `IntClass`, it is a convert constructor.

Convert constructors are automatically invoked by the compiler whenever the context demands a class object but a value of constructor's parameter type is provided. This occurs in four different contexts:

1. An object of the class is initialized with a value of the convert constructor's parameter type: for example

   ```
   IntClass intObject = 23;
   ```

2. An object of the class is assigned a value of the convert constructor's parameter type: for example

   ```
   intObject = 24;
   ```

3. A function expecting a value parameter of the class type is instead passed a value of the constructor's parameter type. For example, we may define a function

   ```
   void printValue(IntClass x)
   {
       cout << x.getValue();
   }
   ```

and then pass it an `int` when we call it:

```
printValue(25);
```

The compiler will use the convert constructor to convert the integer `25` into an object of the `IntClass` class, and will then pass the object to the function. The compiler will not invoke the convert constructor if the formal parameter is a pointer or a reference to an `IntClass` object: *convert constructors are only invoked when the formal parameter uses pass by value.*

4. A function that declares a return value of the class type actually returns a value of the convert constructor's parameter type. For example, the compiler will accept the following function:

```
IntClass f(int intValue)
{
    return intValue;
}
```

Note that the function returns a value of type integer, even though `IntClass` is declared as its return type. Again, the compiler will implicitly call the convert constructor to convert the integer `intValue` to an `IntClass` object. It is this object which is returned from the function.

The following program illustrates the action of convert constructors.

Contents of `Convert.h`

```
#include <iostream>
using namespace std;

class IntClass
{
private:
  int value;
public:
  // Convert constructor from int.
  IntClass(int intValue)
  {
    value = intValue;
  }
  int getValue(){ return value; }
};
```

Contents of `Convert.cpp`

```
#include "Convert.h"
//*****************************************
// This function returns an int even though *
// an IntClass object is declared as the    *
// return type.                             *
//*****************************************
IntClass f(int intValue)
{
   return intValue;
}

```

```
12 //*****************************************
13 // Prints the int value inside an IntClass  *
14 // object.                                  *
15 //*****************************************
16 void printValue(IntClass x)
17 {
18    cout << x.getValue();
19 }
```

Program 11-16

```
1  // This program demonstrates the action of
2  // convert constructors.
3  #include "Convert.h"
4
5  // Function prototypes.
6  void printValue(IntClass);
7  IntClass f(int);
8
9  int main()
10 {
11    // Initialize with an int.
12    IntClass intObject = 23;
13    cout << "The value is " << intObject.getValue() << endl;
14
15    // Assign an int.
16    intObject = 24;
17    cout  << "The value is " << intObject.getValue()  << endl;
18
19    // Pass an int to a function expecting IntClass.
20    cout << "The value is ";
21    printValue(25);
22    cout << endl;
23
24    // Demonstrate conversion on a return.
25    intObject = f(26);
26    cout << "The value is ";
27    printValue(intObject);
28
29    return 0;
30 }
```

Program Output

```
The value is 23
The value is 24
The value is 25
The value is 26
```

You should consider the use of a convert constructor whenever it makes sense to have automatic conversions from some type to the class type. A practical example of the use of

convert constructors can be found in the C++ `string` class. That class provides a convert constructor from C-strings:

```
class string
{
   // Only the convert constructor is shown
   public:
      string(char *);
};
```

The presence of this convert constructor allows programmers to pass C-strings to functions that expect `string` object parameters, assign C-strings to `string` objects, and use C-strings as initial values of `string` objects:

```
string str = "Hello";
str = "Hello There!";
```

In a way, convert constructors work in a way that is the opposite of the type conversion operators covered in a previous section: whereas type conversion operators convert an object to a value of another type, convert constructors convert a value of a given type to an object of the class.



Checkpoint

11.31 What are the benefits of having operator functions that perform object conversion?

11.32 Why is it not necessary to specify a return type for an operator function that performs data type conversion?

11.33 Assume that there is a class named `BlackBox`. Write a prototype for a member function that converts `BlackBox` to `int`.

11.34 Assume there are two classes, `Big` and `Small`.Write a prototype for the convert constructor that converts objects of type `Small` to objects of type `Big`.

11.9 Aggregation and Composition

CONCEPT: Class aggregation occurs when an object of one class owns an object of another class. Class composition is a form of aggregation where the owner class controls the lifetime of objects of the owned class.



VideoNote
Aggregation and Composition

In Chapter 7, you learned that a class can contain members that are themselves objects of other classes. When a class C contains a member that is an object of another class D, every object of C will have inside it an object of the class D. This creates a *has-a* relationship between C and D. In this type of relationship, every instance of C *has*, or *owns*, an instance of the class D. In C++, such ownership usually occurs as result of C having a member of type D, but it can also occur as result of C having a pointer to an object of D. The term *aggregation* is often broadly used to describe situations in which objects of one class own objects of other classes.

Member Initialization Lists

Consider the following `Person` and `Date` classes.

```
class Date
{
   string month;
   int day, year;
public:
   Date(string m, int d, int y)
   {
      month = m;
      day = d;
      year = y;
   }
};

class Person
{
   String name;
   Date dateOfBirth;
public:
   Person(string name, string month, int day, int year)
   {
      // Pass month, day and year to the
      // dateOfBirth constructor
      this->name = name;
   }
};
```

The `Person` constructor receives parameters `month`, `day`, and `year` that it needs to pass to the `Date` constructor of its `dateOfBirth` member. C++ provides a special notation, called a *member initialization list*, that allows constructors of classes to pass arguments to constructors of member objects. A member initialization list is a list of comma-separated calls to member object constructors. It is prefixed with a colon and is placed just after the header, but before the body, of the constructor of the containing class:

```
class Person
{
   String name;
   Date dateOfBirth;
public:
   Person(string name, string month, int day, int year):
   dateOfBirth(month, day, year) // Member initialization list
   {
      this->name = name;
   }
};
```

Notice the colon at the end of the constructor header, and notice that in invoking the constructor of the contained `Date` object, it is the *name of the object* (`dateOfBirth`) rather than the *class of the object* (`Date`) that is used. This allows constructors of different objects of the same class to be invoked in the same initialization list.

Although the member initialization list is usually used to invoke constructors on member objects, it can be used to initialize member variables of any type. Thus, the `Person` and `Date` class can be written as follows:

```
class Date
{
   string month;
   int day, year;
```

```
public:
   Date(string m, int d, int y):
   month(m), day(d), year(y)  // Member Initialization list
   {
   }
};

class Person
{
   String name;
   Date dateOfBirth;
public:
   Person(string name, string month, int day, int year):
   name(name),
   dateOfBirth(month, day, year)
   {
   }
};
```

Notice that the bodies of the `Date` and `Person` constructors are now empty. This is because the assignment of values to member variables normally performed there is now accomplished by the initialization lists. Many programmers prefer the use of member initialization lists to assignment inside of the body of the constructor because it allows the compiler to generate more efficient code in certain situations. When using member initialization lists, it is good programming practice to list the members in the initialization list in the same order that they are declared in the class.

Finally, notice the occurrence of `name(name)` in the initialization list of the `Person` constructor. The compiler is able to determine that the first occurrence of `name` refers to the member variable, and that its second occurrence refers to the parameter.

Aggregation Through Pointers

Now let's suppose that in addition to having a date of birth, each person has a country of residence. A country has a name, and possibly many other attributes:

```
class Country
{
   string name;
   // Additional fields
};
```

Because many people will "have" the same country, the has-a relationship between `Person` and `Country` should not be implemented by embedding an instance of the `Country` class inside every `Person` object. Because many people share the same country of residence, implementing the has-a relation by containment will result in unnecessary duplication of data and waste memory. In addition, it would require many `Person` objects to be updated whenever a country has a change in any of its data. Using a pointer to implement the has-a relation avoids these problems. Here is a version of the `Person` class, modified to include a pointer to the country of residence:

```
class Person
{
   string name;
   Date dateOfBirth;
   Country *pCountry;  // Pointer to country of residence
```

```
public:
   Person(string name, string month, int day, int year, Country *pC):
   dateOfBirth(month, day, year), name(name), pCountry(pC)
   {
   }
};
```

Aggregation, Composition, and Object Lifetimes

Composition is a term used to describe special cases of aggregation in which the lifetime of the owned object coincides with the lifetime of its owner. A good example of composition is when a class C contains a member that is an object of another class D. The contained D object is created at the same time that the C object is created, and is destroyed when the containing C object is destroyed or goes out of scope. Another example of composition is when a class C contains a pointer to a D object, and the D object is created by the C constructor and destroyed by the C destructor.

The following program features modified versions of the above classes designed to illustrate aggregation, composition, and object lifetimes. Each class has a constructor to announce the creation of its objects and a destructor to announce their demise. The `Person` class has a static member

```
int Person::uniquePersonID;
```

that is used to generate numbers assigned to `Person` objects as they are created. These numbers serve as a sort of universal personal identification number, much as social security numbers are used to identify people in the United States. The numbers are stored in a `personID` field of the `Person` and `Date` classes, and are used to identify objects being created or destroyed. Each `dateOfBirth` object carries the same `personID` number as the Person object that contains it.

Program 11-17

```
1  // This program illustrates aggregation, composition
2  // and object lifetimes.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  class Date
8  {
9     string month;
10    int day, year;
11    int personID;  //ID of person whose birthday this is.
12 public:
13    Date(string m, int d, int y, int id):
14    month(m), day(d), year(y), personID(id)
15    {
16       cout << "Date-Of-Birth object for person "
17            << personID  << " has been created.\n";
18    }
19    ~Date()
```

(program continues)

Program 11-17 *(continued)*

```
20    {
21       cout << "Date-Of-Birth object for person "
22            << personID  << " has been destroyed.\n";
23    }
24 };
25
26 class Country
27 {
28    string name;
29 public:
30    Country(string name) : name(name)
31    {
32       cout << "A Country object has been created.\n";
33    }
34    ~Country()
35    {
36       cout << "A Country object has been destroyed.\n";
37    }
38 };
39
40 class Person
41 {
42    string name;
43    Date dateOfBirth;
44    int personID;       // Person identification number (PID)
45    Country *pCountry;
46 public:
47    Person(string name, string month, int day, int year, Country *pC):
48    name(name),
49    dateOfBirth(month, day, year, Person::uniquePersonID),
50    personID(Person::uniquePersonID),
51    pCountry(pC)
52    {
53       cout << "Person object "
54            << personID << " has been created.\n";
55       Person::uniquePersonID ++;
56    }
57    ~Person()
58    {
59       cout << "Person object "
60            << personID << " has been destroyed.\n";
61    }
62    static int uniquePersonID;  // Used to generate PIDs
63 };
64
65 // Define the static class variable
66 int Person::uniquePersonID = 1;
67
68 int main()
69 {
70    // Create a Country object
```

(program continues)

Program 11-17 *(continued)*

```
71    Country usa("USA");
72    // Create a Person object
73    Person *p = new Person("Peter Lee", "January", 1, 1985, &usa);
74    // Create another Person object
75    Person p1("Eva Gustafson", "May", 15, 1992, &usa);
76    cout << "Now there are two people.\n";
77    // Delete the first person
78    delete p;
79    cout << "Now there is only one.\n";
80    // The second person will go out of scope when main returns
81    return 0;
82 }
```

Program Output

```
A Country object has been created.
Date-Of-Birth object for person 1 has been created.
Person object 1 has been created.
Date-Of-Birth object for person 2 has been created.
Person object 2 has been created.
Now there are two people.
Person object 1 has been destroyed.
Date-Of-Birth object for person 1 has been destroyed.
Now there is only one.
Person object 2 has been destroyed.
Date-Of-Birth object for person 2 has been destroyed.
A Country object has been destroyed.
```

The relationship between the `dateOfBirth` objects and the `Person` objects that contain them is an example of composition. As you can see from the program output, those `Date` objects are created at the same time, and die at the same time, as the `Person` objects that own them. Aggregation in its more general form is exemplified by the has-a relationship between `Person` and `Country`.

By looking at the `print` member function, you can see an example of how the member functions of the enclosing class can access the member functions of the contained class.

The Has-A Relation

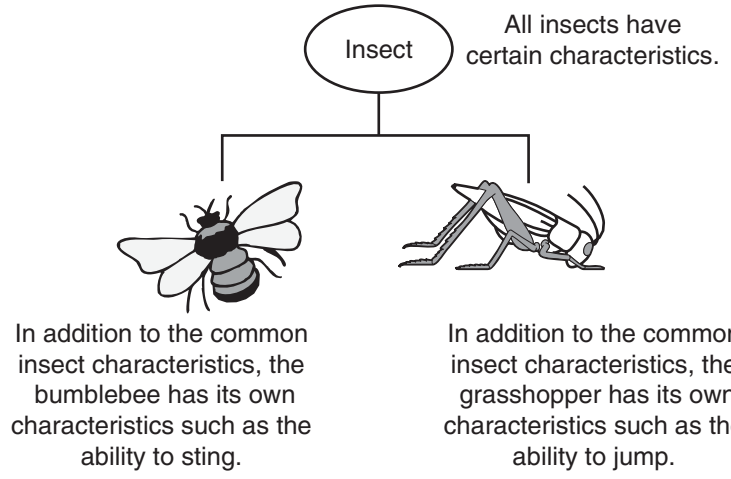
When one class contains an instance of a second class, the first class is said to sustain a *has-a* relation to the second. For example, the `Acquaintance` class has-a `Date` class in the form of its `dob` member, while the `Date` class has-a `string` object in the form of its `month` member. The has-a relation is important in modeling relationships among classes and objects during the design of an object-oriented system. Another important relation between classes in a program is the is-a relation, which we will discuss in a later section after we have discussed the concept of inheritance. Thus object composition realizes the has-a relation, while, as we will see later, inheritance is a way of realizing the is-a relation.

11.10 Inheritance

Generalization and Specialization

In the real world you can find many objects that are specialized versions of other more general objects. For example, the term "insect" describes a very general type of creature with numerous characteristics. Because grasshoppers and bumblebees are insects, they have all the general characteristics of an insect. In addition, they have special characteristics of their own. For example, the grasshopper has its jumping ability, and the bumblebee has its stinger. Grasshoppers and bumblebees are specialized versions of an insect. This is illustrated in Figure 11-5.

Figure 11-5



Inheritance and the Is-a Relationship

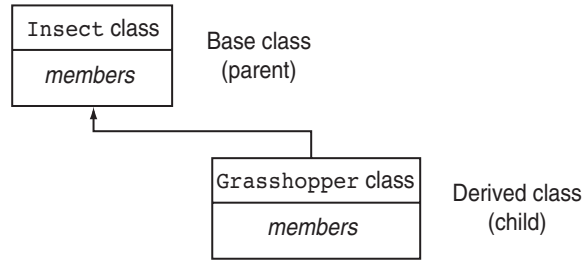
When one object is a specialized version of another object, there is an *is-a relationship* between them. For example, a grasshopper *is an* insect. Here are a few other examples of the is-a relationship.

- A poodle *is a* dog.
- A car *is a* vehicle.
- A rectangle *is a* shape.

When an is-a relationship exists between objects, it means that the specialized object has all of the characteristics of the general object, plus additional characteristics that make it special. In object-oriented programming, *inheritance* is used to create an is-a relationship between classes.

Inheritance involves a base class and a derived class. The *base class* is the general class and the *derived class* is the specialized class. The derived class is based on, or derived from, the base class. You can think of the base class as the parent and the derived class as the child. This is illustrated in Figure 11-6.

Figure 11-6



The derived class inherits the member variables and member functions of the base class without any of them being rewritten. Furthermore, new member variables and functions may be added to the derived class to make it more specialized than the base class. To take a specific example, consider a college or university environment where there are both students and faculty personnel. Suppose that we have a class `Person` with a `name` data member and member functions for working with the `name` member:

```
class Person{
private:
   string name;
public:
   Person(){ setName(""); }
   Person(string pName) { setName(pName);}
   void setName(string pName) { name = pName; }
   string getName() { return name;}
};
```

Assuming the enumerated types

```
enum Discipline {
   ARCHEOLOGY, BIOLOGY, COMPUTER_SCIENCE
};
enum Classification {
   FRESHMAN, SOPHOMORE, JUNIOR, SENIOR
};
```

to define the range of disciplines in which studies are offered and the classification of students, we can define both the `Student` and `Faculty` classes as classes that inherit from the `Person` class. This makes sense because a `Student` is a `Person`, and a `Faculty` member is also a `Person`.

To define a class by inheritance, we need to specify the base class plus the additional members that the derived class adds to the base class. Let's say that in addition to having all the characteristics of a `Person`, a `Student` must declare a major in some discipline and have an academic advisor who is a `Person`. The `Student` class can be defined as follows:

```
class Student : public Person
{
private:
   Discipline major;
   Person *advisor;
```

```
public:
   void setMajor(Discipline d) { major = d; }
   Discipline getMajor() { return major;  }
   void setAdvisor(Person *p) { advisor = p; }
   Person *getAdvisor() { return advisor; }
};
```

We assume that many different students may have the same advisor. Having each `Student` object store a copy of the advisor data would lead to unnecessary duplication, and would force us to update each student object whenever the advisor data is updated. The `Student` object stores a pointer to the advisor to avoid these and other problems.

The first part of the first line of the class declaration specifies `Student` as the name of the class being defined, and specifies the existing class `Person` as its base class:

```
class Student : public Person
```

Derived class — Base class

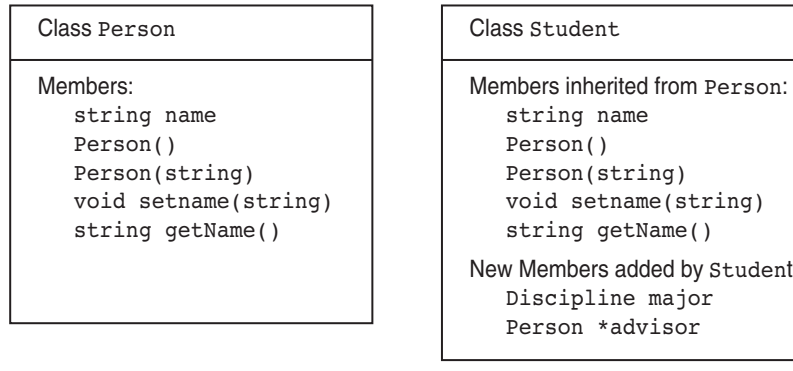
The key word `public` that precedes the name of the base class is the *base class access specification*. It affects how the members of the base class will be accessed by the member functions of the derived class, and by code outside of the two classes. We will discuss the base class access specification in greater detail in a later section.

A class can be used as the base class for more than one derived class. In particular, a `Faculty` class can also be derived from the `Person` class as follows:

```
class Faculty : public Person
{
private:
  Discipline department;
public:
  void setDepartment(Discipline d) { department = d; }
  Discipline getDepartment( ) { return department;  }
};
```

Thus a `Faculty` object is a `Person` object that has a home department in some discipline. It is also important to note that each object of the derived class will contain all members of the base class. This is illustrated by Figure 11-7 in the case of the `Student` and `Person` classes.

Figure 11-7



Program 11-18 demonstrates the creation and use of an object of a derived class by creating a `Faculty` object. The program uses arrays of strings that map values of enumerated types to strings to enable printing of values of enumerated types in the form of strings. The included "`inheritance.h`" file contains declarations of the `Person`, `Student`, and `Faculty` classes, as well as the enumerated types `Discipline` and `Classification`.

Contents of `Inheritance.h`

```
#include <string>
using namespace std;

enum Discipline { ARCHEOLOGY, BIOLOGY, COMPUTER_SCIENCE };
enum Classification { FRESHMAN, SOPHOMORE, JUNIOR, SENIOR };

class Person
{
private:
  string name;
public:
  Person() { setName(""); }
  Person(string pName) { setName(pName); }
  void setName(string pName) { name = pName; }
  string getName() { return name; }
};

class Student:public Person
{
private:
  Discipline major;
  Person *advisor;
public:
  void setMajor(Discipline d) { major = d; }
  Discipline getMajor() { return major; }
  void setAdvisor(Person *p) { advisor = p; }
  Person *getAdvisor() { return advisor; }
};

class Faculty:public Person
{
private:
  Discipline department;
public:
  void setDepartment(Discipline d) { department = d; }
  Discipline getDepartment( ) { return department; }
};
```

Program 11-18

```
// This program demonstrates the creation and use
// of objects of derived classes.
#include <iostream>
#include "inheritance.h"

using namespace std;
```

(program continues)

Program 11-18 *(continued)*

```
 7
 8 // These arrays of string are used to print the
 9 // enumerated types.
10 const string dName[] = {
11   "Archeology", "Biology", "Computer Science"
12 };
13
14 const string cName[] = {
15   "Freshman", "Sophomore", "Junior", "Senior"
16 };
17
18 int main()
19 {
20    // Create a  Faculty object.
21    Faculty prof;
22
23    // Use a Person member function.
24    prof.setName("Indiana Jones");
25
26    // Use a Faculty member function.
27    prof.setDepartment(ARCHEOLOGY);
28    cout << "Professor " << prof.getName()
29         << " teaches in the " << "Department of ";
30
31    // Get department as an enumerated type.
32    Discipline dept = prof.getDepartment();
33
34    // Print out the department in string form.
35    cout << dName[dept] << endl;
36
37    return 0;
38 }
```

Program Output

```
Professor Indiana Jones teaches in the Department of Archeology
```

Superclasses and Subclasses

We can think of a class as describing the set of all objects that have certain characteristics. An object of a derived class inherits all the characteristics of the base class, so it can be regarded as belonging to the base class. Thus objects of the derived class are just specialized objects of the base class. For this reason, the derived class is often called a *subclass* of the base class, and the base class is called a *superclass* of the derived class.

Multiple Inheritance

C++ supports *multiple inheritance*, in which a derived class simultaneously derives from two or more base classes. Although interesting, multiple inheritance can lead to programs that are very difficult to understand, and is rarely useful in practice. You can find a discussion of it in Appendix K of the student CD.



Appendix D on the student CD shows how to represent inheritance using UML.

11.11 Protected Members and Class Access

CONCEPT: Protected members of a base class are like private members, except they may be accessed by derived classes. The base class access specification determines how private, protected, and public base class members are accessed when they are inherited by the derived class.

Until now you have used two access specifications within a class: `private` and `public`. C++ provides a third access specification, `protected`. Protected members of a base class are like private members, except they may be accessed by member functions of a derived class. Protected members are inaccessible to all other code in the program.*

Let us suppose we want to add to the `Faculty` class a constructor that takes as parameter the name and department of a professor. The best way to accomplish this is to have the constructor call the `setName()` member function inherited from the `Person` class. To illustrate the use of protected members, however, we will change the access specification of the `name` field of the `Person` class to `protected` and have the `Faculty` constructor access it directly. We make similar changes to the `Student` class, adding a constructor that takes parameters and sets the protected member `name`. The resulting code is stored in the `inheritance1.h` file:

Contents of `Inheritance1.h`

```
1  #include <string>
2  using namespace std;
3
4  enum Discipline { ARCHEOLOGY, BIOLOGY, COMPUTER_SCIENCE };
5  enum Classification { Freshman, Sophomore, Junior, Senior };
6
7  class Person
8  {
9  protected:
10   string name;
11 public:
12   Person() { setName("");}
13   Person(string pName) { setName(pName);}
14   void setName(string pName) { name = pName; }
15   string getName() { return name; }
16 };
17
18 class Student:public Person
19 {
20 private:
21   Discipline major;
22   Person *advisor;
```

* Friend functions and friend classes have access to both private and protected members.

```
public:
  // Constructor.
  Student(string sname, Discipline d, Person *adv);

  void setMajor(Discipline d) { major = d; }
  Discipline getMajor(){return major; }
  void setAdvisor(Person *p){advisor = p;}
  Person *getAdvisor() { return advisor; }
 };

class Faculty:public Person
{
private:
  Discipline department;
public:
  // Constructor.
  Faculty(string fname, Discipline d)
  {
     // Access the protected base class member.
     name = fname;
     department = d;
   }
   // Other member functions.
   void setDepartment(Discipline d) { department = d; }
   Discipline getDepartment( ) { return department; }
};
```

Contents of inheritance1.cpp

```
#include "inheritance1.h"
//*****************************************
// Constructor for the Student class.         *
//*****************************************
Student::Student(string sname, Discipline d, Person *adv)
{
   // Access the  protected member name.
   name = sname;

   // Access the other members.
   major = d;
   advisor = adv;
}
```

Program 11-19 demonstrates the use of these classes.

Program 11-19

```
//This program demonstrates the use of
//objects of derived classes.
#include "inheritance1.h"
#include <iostream>
using namespace std;

// These arrays of string are used to print
// values of enumerated types.
```

(program continues)

Program 11-19 *(continued)*

```
 9 const string dName[] =
10       {"Archeology", "Biology", "Computer Science"};
11
12 const string cName[] =
13   {"Freshman", "Sophomore", "Junior", "Senior"};
14
15 int main()
16 {
17 // Create Faculty and Student objects.
18    Faculty prof("Indiana Jones", ARCHEOLOGY);
19    Student st("Sean Bolster", ARCHEOLOGY, &prof);
20    cout << "Professor " << prof.getName() << " teaches "
21         << dName[prof.getDepartment()] << "." << endl;
22
23    //Get student's advisor
24    Person *pAdvisor = st.getAdvisor();
25    cout << st.getName() <<"\'s advisor is "
26         << pAdvisor->getName() << ".";
27
28    return 0;
29 }
```

Program Output

```
Professor Indiana Jones teaches Archeology.
Sean Bolster's advisor is Indiana Jones.
```

Although our example does not show it, member functions of a base class can be declared protected as well. Protected member functions can be called by member functions of derived classes, and by friend functions and friend classes.

Base Access Specifications

In addition to public, C++ permits the use of protected and private as base access specifications, as illustrated in the following (incompletely specified) examples

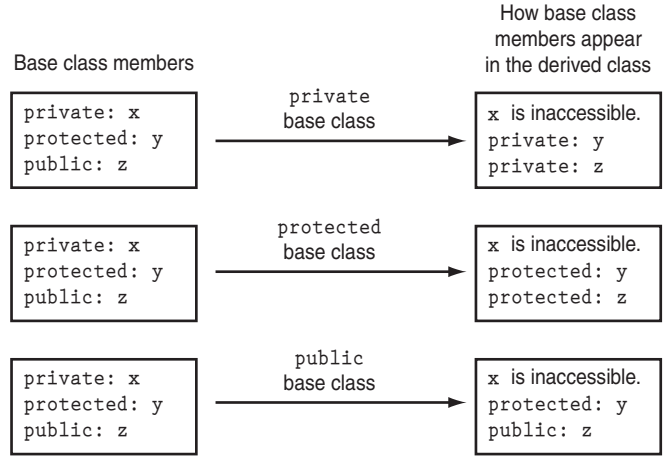
```
class Cat : protected Feline
{

};

class Dog : private Canine
{

};
```

Be careful not to confuse base access specification with member access specification. Member access specification determines the type of access for members *defined* in the class, whereas base access specification determines the type of access for *inherited* members. Table 11-2 and Figure 11-8 show the interplay between member access specification in the base class and base class specification that determines access to the inherited member.

Table 11-2 Base Class Access Specifications

Base Class Access Specification	How Members of the Base Class Appear in the Derived Class
`private`	• Private members of the base class are inaccessible to the derived class. • Protected members of the base class become private members of the derived class. • Public members of the base class become private members of the derived class.
`protected`	• Private members of the base class are inaccessible to the derived class. • Protected members of the base class become protected members of the derived class. • Public members of the base class become protected members of the derived class.
`public`	• Private members of the base class are inaccessible to the derived class. • Protected members of the base class become protected members of the derived class. • Public members of the base class become public members of the derived class.

Figure 11-8



Checkpoint

11.35 What type of relationship between classes is realized by inheritance?

11.36 Why does it make sense to think of a base class as a superclass of its derived class?

11.37 What is a base class access specification?

11.38 Think of an example of two classes where one class is a special case of the other, and write declarations for both classes, with the special case being written as a derived class.

11.39 What is the difference between private members and protected members?

11.40 What is the difference between member access specification and base class access specification?

11.41 Suppose a program has the following class declaration:

```
class CheckPoint
{
   private:
      int a;
   protected:
      int b;
      int c;
      void setA(int x) { a = x;}
   public:
      void setB(int y) { b = y;}
      void setC(int z) { c = z;}
};
```

Answer the following questions.

A) Suppose another class, `Quiz`, is derived from the `CheckPoint` class. Here is the first line of its declaration:

```
class Quiz : private CheckPoint
```

Indicate whether each member of the `CheckPoint` class is `private`, `protected`, `public`, or inaccessible:

```
a
b
c
setA
setB
setC
```

B) Suppose the `Quiz` class, derived from the `CheckPoint` class, is declared as

```
class Quiz : protected Checkpoint
```

Indicate whether each member of the `CheckPoint` class is `private`, `protected`, `public`, or inaccessible:

```
a
b
c
setA
setB
setC
```

C) Suppose the `Quiz` class, derived from the `CheckPoint` class, is declared as

```
class Quiz : public Checkpoint
```

Indicate whether each member of the `CheckPoint` class is `private`, `protected`, `public`, or inaccessible:

```
a
b
c
setA
setB
setC
```

D) Suppose the `Quiz` class, derived from the `CheckPoint` class, is declared as

```
class Quiz : Checkpoint
```

Is the `CheckPoint` class a `private`, `public`, or `protected` base class?

11.12 Constructors, Destructors, and Inheritance

CONCEPT: When an object of a derived class is being instantiated, the base class constructor is called before the derived class constructor. When the object is destroyed, the derived class destructor is called before the base class destructor.

Recall that constructors are automatically called by the compiler whenever an object of a class is being created. Because every object of a derived class can be regarded as having an object of the base class embedded within it, the creation of a derived class object involves the creation of the embedded base class object. The compiler will always call the base class constructor before it calls the derived class constructor. This order is reversed upon destruction of a derived class object; the destructor in the derived class is called before the destructor in the base class. This order permits the derived class constructors and destructors to use data or member functions of the base class in doing their work.

Program 11-20 illustrates this behavior in a simple program.

Program 11-20

```
1  // This program demonstrates the order in which base and
2  // derived class constructors and destructors are called.
3  // For the sake of simplicity, all the class declarations
4  // are in this file.
5  #include <iostream>
6  using namespace std;
7
8  // Base class.
9  class BaseDemo
10 {
11 public:
12    BaseDemo()  // Constructor
13       { cout << "This is the BaseDemo constructor.\n"; }
14    ~BaseDemo() // Destructor
15       { cout << "This is the BaseDemo destructor.\n"; }
16 };
```

(program continues)

Program 11-20 *(continued)*

```
17
18 // Derived class.
19 class DeriDemo : public BaseDemo
20 {
21 public:
22    DeriDemo()  //Constructor
23       { cout << "This is the DeriDemo constructor.\n"; }
24    ~DeriDemo()  // Destructor
25       { cout << "This is the DeriDemo destructor.\n"; }
26 };
27
28 int main()
29 {
30    cout << "We will now create a DeriDemo object.\n";
31    DeriDemo object;
32    cout << "The program is now going to end.\n";
33    return 0;
34 }
```

Program Output

```
We will now create a DeriDemo object.
This is the BaseDemo constructor.
This is the DeriDemo constructor.
The program is now going to end.
This is the DeriDemo destructor.
This is the BaseDemo destructor.
```

Passing Arguments to Base Class Constructors

As already mentioned, the compiler will automatically call a base class constructor before executing the derived class constructor. The compiler's default action is to call the default constructor in the base class. Some classes, however, may not have a default constructor. Also, the programmer may want to specify which of several base class constructors should be called during the creation of a derived class object.

In these cases, the programmer must explicitly specify which base class constructor should be called by the compiler. This is done by specifying the arguments to the selected base class constructor in the definition of the derived class constructor.

The syntax for passing arguments to base class constructors is simple: the header for the derived class constructor is followed by a colon, an indication of which base class constructor to call, and the arguments to be passed. To illustrate, we will modify the constructor for the `Faculty` class so that it invokes a constructor in the `Person` class.

The constructor in its previous form was

```
Faculty(string fname, Discipline d)
{
   name = fname;
   department = d;
}
```

It now becomes

```
Faculty(string fname, Discipline d) : Person(fname)
{
   department = d;
}
```

Notice that one of the arguments passed to the derived class constructor is passed to the base class constructor. In general, the argument passed to the base class constructor may be any expression, and may involve any variables that are in scope at the point of the call to the derived class constructor. For example, a string literal, or even a global string variable, could have been passed as the argument to the `Person` constructor. If for example, it was desired that the name of a faculty member default to that of the ubiquitous "Dr. Staff," the following constructor would be just what we want:

```
Faculty(Discipline d) : Person("Staff")
{
   department = d;
}
```

In general, the base class constructor may take any number of parameters.

Contents of `inheritance2.h`

```
#include <string>
using namespace std;

enum Discipline { ARCHEOLOGY, BIOLOGY, COMPUTER_SCIENCE };
enum Classification { FRESHMAN, SOPHOMORE, JUNIOR, SENIOR };

class Person
{
protected:
  string name;
public:
  Person() {setName(""); }
  Person(string pName) { setName(pName); }
  void setName(string pName) { name = pName; }
  string getName() { return name; }
};

class Student:public Person
{
private:
  Discipline major;
  Person *advisor;
public:
  // Constructor.
  Student(string sname, Discipline d, Person *adv);

  void setMajor(Discipline d) { major = d; }
  Discipline getMajor() { return major; }
  void setAdvisor(Person *p) { advisor = p; }
  Person *getAdvisor() { return advisor; }
```

```
31 };
32
33 class Faculty:public Person
34 {
35 private:
36   Discipline department;
37 public:
38   // Constructor.
39   Faculty(string fname, Discipline d) : Person(fname)
40   {
41      department = d;
42    }
43
44   void setDepartment(Discipline d) { department = d; }
45   Discipline getDepartment( ) { return department; }
46 };
```

Contents of `inheritance2.cpp`

```
1  #include "inheritance2.h"
2  //*****************************************
3  // Constructor for the Student class.         *
4  //*****************************************
5  Student::Student(string sname, Discipline d, Person *adv)
6  : Person(sname) // Base constructor initialization
7  {
8     major = d;
9     advisor = adv;
10 }
```

The new constructors are demonstrated in Program 11-21App on the student CD, which is virtually the same as Program 11-19. The only difference is that Program 11-21App includes `inheritance2.h` rather than `inheritance1.h`, and must be compiled and linked with `inheritance2.cpp`. It can be found in the file Pr11-21App.cpp on the student CD.

It is important to remember that the arguments to the base class constructor must be specified in the *definition* of the derived class constructor, and not in its *declaration*. In the case of the `Student` class, the declaration of the constructor occurs at line 25 of the `inheritance.h` file. The corresponding definition starts at line 5 of the `inheritance.cpp` file, and specifies the argument to pass to the `Person` superclass in line 6.



Checkpoint

11.42 What is the reason that base class constructors are called before derived class constructors?

11.43 Why do you think the arguments to a base class constructor are specified in the definition of the derived class constructor rather than in the declaration?

11.44 Passing arguments to base classes constructors solves the problem of selecting a base class constructor in inheritance. Can the same problem arise with composition? That is, might there be a case where a constructor of a class might have to pass arguments to the constructor of a contained class? If so, guess the syntax that

would be used to pass the parameters, and construct a simple example to verify your guess.

11.45 What will the following program display?

```
#include <iostream>
using namespace std;
class Base
{
    public:
       Base() { cout << "Entering the base.\n"; }
       ~Base() { cout << "Leaving the base.\n"; }
};

class Camp : public Base
{
public:
   Camp() { cout << "Entering the camp.\n"; }
   ~Camp() { cout << "Leaving the camp.\n"; }
};
int main()
{
   Camp outpost;
   return 0;
}
```

11.46 What will the following program display?

```
#include <iostream>
using namespace std;
class Base
{
public:
   Base() { cout << "Entering the base.\n"; }
   Base(char *str)
      { cout << "This base is"
             << str  << ".\n";
      }
   ~Base() { cout << "Leaving the base.\n"; }
};
class Camp : public Base
{
public:
   Camp() { cout << "Entering the camp.\n"; }
   Camp(char * str1, char *str2) : Base(str1)
      { cout << "The camp is " << str2 << ".\n"; }
   ~Camp() { cout << "Leaving the camp.\n"; }
};

int main()
{
   Camp outpost("secure", "secluded");
   return 0;
}
```

11.13 Overriding Base Class Functions

CONCEPT: **A derived class can override a member function of its base class by defining a derived class member function with the same name and parameter list.**



VideoNote
Overriding Base Class Functions

It is often useful for a derived class to define its own version of a member function inherited from its base class. This may be done to specialize the member function to the needs of the derived class. When this happens, the base class member function is said to be *overridden*, or *redefined*, by the derived class.

As a simple example, suppose that we want to have a class `Tfaculty` that will allow us to associate with each member of the faculty a title such as "Dr.", "Professor", or "Dean." To accomplish this, we derive the new class from the `Faculty` class by adding a `title` data member, an appropriate constructor, a member function to set the title, and then overriding the inherited `getName()` member function to return a "titled" name.

```
class TFaculty: public Faculty
{
private:
  string title;
public:
  // This Constructor allows the specification of a title.
  TFaculty(string fname, Discipline d, string title)
  : Faculty(fname, d)
  {
    setTitle(title);
  }

  void setTitle(string title) { this->title = title; }

  // Override the getName function.
  string getName( ){ return title + " " + name; }
};
```

Program 11-21 illustrates the use of this class and its overridden member function. It uses the files `inheritance3.h` and `inheritance3.cpp`. The `inheritance3.h` file is just `inheritance2.h` with the class declaration of `TFaculty` added, and `inheritance3.cpp` is the same as `inheritance2.cpp`. Code listings of `inheritance2.h` and `inheritance2.cpp` can be found at the end of Section 11.12. Copies of all these files are included on the student CD.

Program 11-21

```
1 // This program illustrates member function overriding.
2 #include "inheritance3.h"
3 #include <iostream>
4 using namespace std;
5
```

(program continues)

Program 11-21 *(continued)*

```
// These arrays of string are used to output
// values of enumerated types.
const string dName[] =
    { "Archeology", "Biology", "Computer Science" };

const string cName[] =
    { "Freshman", "Sophomore", "Junior", "Senior" };

int main()
{
   // New constructor allows specification of title.
   TFaculty prof("Indiana Jones", ARCHEOLOGY, "Dr.");
   Student st("Sean Bolster", ARCHEOLOGY, &prof);

   // Use the new TFaculty version of getName.
   cout << prof.getName() << " teaches "
        << dName[prof.getDepartment()] << "." << endl;

   // This call uses the Person version of getName.
   Person *pAdvisor = st.getAdvisor();
   cout << st.getName() <<"\'s advisor is "
        << pAdvisor->getName() << ".";

   return 0;
}
```

Program Output

```
Dr. Indiana Jones teaches Archeology.
Sean Bolster's advisor is Indiana Jones.
```

Choosing Between Base and Derived Class Versions of an Overriden Function

An object of a derived class that has overridden a base class member function contains more than one version of the member function. The compiler will determine which of the several versions to call by using type information in the expression used to make the call to the member function. For example, in Program 11-21, there are two calls to `getName()`:

1. The call `prof.getName()` returns `Dr. Indiana Jones` because the function is called through `prof`, which has type `TFaculty`. The compiler calls the `TFaculty` version `getName()`.
2. The call `pAdvisor->getName()` returns `Indiana Jones` without the "Dr." because the function is called through the pointer `pAdvisor`, which is a pointer to `Person`. The compiler calls the Person version of `getName()`.

The Difference Between Overloading and Overriding

Both overloading and overriding involve the definition of different functions with the same name. There are differences between the two concepts, however. Overriding can only be done in the context of inheritance, and refers to the defining of a member function by a

derived class when the base class already has a member function of the same name and parameter list. Overloading refers to the definition of different functions within the same class with the same name and *different* parameter lists. Overloading can also refer to the definition of different functions with different parameter lists at the global level.

Gaining Access to an Overridden Member Function

If a derived class overrides a base class member function, member functions of the derived class that would have otherwise called the overridden base class member function will now call the version in the derived class. It is occasionally useful to be able to call the overridden version. In fact, the new member function of the derived class may want to call the base class member function that it is overriding. This is done by using the scope resolution operator to specify the class of the overridden member function being accessed. For example, a member function of `TFaculty` that is to call the `getName` function of `Person` can do so in this fashion:

```
Person::getName();
```

Thus, a better version of the `TFaculty` class is the following. Note that the overriding function does not need to access any protected members of `Person`, but instead calls the public member function `getName`.

```
class TFaculty : public Faculty
{
private:
   string title;
public:
   TFaculty(string fname, Discipline d, string title)
   : Faculty(fname, d)
   {
      setTitle(title);
   }
   void setTitle(string title) { this->title = title; }
   // Override getName() by calling Person::getName
   string getName( )
   {
      return title + " " + Person::getName();
   }
};
```

Code for the program demonstrating this can be found in files `Pr11-22App.cpp`, `inheritance4.h`, and `inheritance4.cpp` on the student CD.

11.14 Tying It All Together: *Putting Data on the World Wide Web*

The ability to generate output formatted in HTML (Hypertext Markup Language) is important to programs that interact with users via the World Wide Web. These applications include Web servers and Web-based E-commerce applications such as Amazon and eBay. Often the information displayed by these programs must be formatted using *HTML tables*.

HTML tables are quite simple. They consist of *rows* of cells where each cell holds a unit of information referred to as *table data*. The information comprising the table is marked with HTML *tags* as shown in Table 11-3.

Table 11-3 HTML Tags for Formatting Tables

`<table>`	Marks the beginning of the table
`</table>`	Marks the end of the table
`<tr>`	Marks the beginning of a row in the table
`</tr>`	Marks the end of a row in the table
`<td>`	Marks the beginning of data in a single cell of the table
`</td>`	Marks the end of data in a single cell of the table
`<th>`	Marks the beginning of the header for a single column of the table
`</th>`	Marks the end of the header for a single column of the table

The `<table>` tag normally causes a browser to display tables with no borders. To display tables with borders, the *border* attribute can be used. For example, the data table shown in Table 11-4 can be displayed using the following HTML markup, identified here as the contents of a file named `table.html`. The file can be found on the student CD in the Programs folder for this chapter.

Table 11-4 Sample Input Data for the HTML Table Program

Name	Address	Phone
Mike Sane	1215 Mills St	630-555-1293
Natasha Upenski	513 Briarcliff Ln	412-555-1004

Contents of `table.html`

```
<table border = "1">
  <tr>
       <th> Name    </th>
       <th> Address </th>
       <th> Phone   </th>
  </tr>
  <tr>
       <td> Mike Sane     </td>
       <td> 1215 Mills St </td>
       <td> 630-728-1293  </td>
  </tr>
  <tr>
      <td> Natasha Upenski   </td>
      <td> 513 Briarcliff Ln </td>
      <td> 412-672-1004      </td>
  </tr>
</table>
```

If you use Microsoft Windows, you can display this table in your browser by double-clicking on the file `table.html` in Windows Explorer, or by using *Open* in the *File* menu of your browser.

Let's write a program that converts a two-dimensional array of strings into an HTML table capable of being displayed in a Web browser. The centerpiece of our program is an `HTMLTable` class with two member variables

```
vector<string> headers;
vector<vector<string> > rows;
```

that represent the *headers* and the *rows* of the table. The headers constitute a single vector of strings, while the rows of the table are represented by a vector of vectors. Note the blank character between the closing > > characters at the end of the definition of a vector of vectors: it is needed to avoid confusion with the >> operator. The `HTMLTable` class has a member function `setHeaders()` for setting the headers and a member function `addRow()` for adding rows to the table. The class also has an overloaded stream output operator

```
ostream & operator <<(ostream &out, HTMLTable hTable);
```

that is used to convert the table data stored in the `headers` and `rows` vectors into HTML markup and write that markup onto an output stream. If the stream receiving the markup is a file, you can open the file in a browser for viewing. Alternatively, if your system has established a default browser for opening files with an `.html` extension, you can use the C++ library function

```
system("file_location.html");
```

to make the operating system open the HTML file using the default browser. This, of course, assumes that the HTML markup is stored in a file named `"file_location.html."`

We learned in this chapter that a derived class object can be used wherever a base class object is expected. We use this fact to allow for more flexibility in the data that is passed as parameters to functions. We build this flexibility into an overloaded stream output operator, and into a convert constructor for a class derived from the STL `vector` class. The overloaded stream output operator is able to write the standard output object `cout` because `cout` is an `ostream` object. It is also able to write to an `ofstream` object because `ofstream` inherits from `ostream`.

C++ programmers often face a dilemma: Although vectors are preferable to arrays in many ways, arrays are more convenient to initialize. The solution we adopt is to use vectors as parameters to functions, but use arrays for initialization. To bridge the gap, we write a `StringVector` class that has a convert constructor that takes an array of strings as parameter. Because `StringVector` derives from a vector of `string`, it can be used wherever a vector of `string` is expected. At the same time, its convert constructor allows for the use of arrays of strings, as shown in lines 91–100 of Program 11-22.

Program 11-22

```
// This program demonstrates the use of classes to
// to put tabular data on the World Wide Web.
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
using namespace std;

// This is a convenience class used to convert
// an array of strings into a vector of strings.
// The array of strings must be (sentinel)-terminated
// by a string of length 0.

class StringVector : public vector<string>
{
public:
     StringVector(string s[])
     {
         int k = 0;
         while (s[k].length() != 0)
         {
             this->push_back(s[k]);
             k++;
         }
     }
 };

// This class allows a 2-dimensional table expressed as
// a vectors of vector of strings to be transformed into
// HTML form.

class HTMLTable
{
private:
     vector<string> headers;
     vector<vector<string> > rows;
     // Helper method for writing an HTML row in a table
     void writeRow(ostream &out, string tag, vector<string> row);
public:
     // Set headers for the table columns
     void setHeaders(const vector<string> &headers)
     {  this->headers = headers; }
     // Add rows to the table
     void addRow(const vector<string> &row)
     {  rows.push_back(row);  }
     // Write the table into HTML form onto an output stream
    friend ostream & operator<<(ostream & out, HTMLTable htmlTable);
 };

```

(program continues)

Program 11-22 *(continued)*

```
50 //*************************************************************
51 // Writes a row of the table, using the given tag for the table    *
52 // data. The tag may be td for table data or th for table header.  *
53 //*************************************************************
54 void HTMLTable::writeRow(ostream &out, string tag, vector<string> row)
55 {
56     out << "<tr>\n";
57     for (unsigned int k = 0; k < headers.size(); k++)
58      {
59          out << "<" << tag << "> "
60              << row[k] << " </" << tag << "> ";
61      }
62     out << "\n</tr>\n";
63  }
64
65 //*****************************************************
66 // Overloaded stream output operator <<                *
67 //*****************************************************
68 ostream & operator<<(ostream &out, HTMLTable htmlTable)
69 {
70      out << "<table border = \"1\">\n";
71      // Write the headers
72      htmlTable.writeRow(out, "th", htmlTable.headers);
73      // Write the rows of the table
74      for (unsigned int r = 0; r < htmlTable.rows.size(); r++)
75      {
76          htmlTable.writeRow(out, "td", htmlTable.rows[r]);
77      }
78      // Write end tag for table
79      out << "</table>\n";
80      return out;
81 }
82
83 int main()
84 {
85      // Hard-coded data for table column headers
86      // The arrays must have empty string sentinels
87      string headers [] = {"Name", "Address", "Phone", ""};
88
89      // Hard-coded data for the two rows of the table
90      // The arrays must have empty string sentinels
91      string person1 [] =
92         {"Mike Sane", "1215 Mills St", "630-728-1293", ""};
93      string person2 [] =
94         {"Natasha Upenski", "513 Briarcliff Ln", "412-672-1004", ""};
95
96      // Create the HTML table object and set its members
97      HTMLTable hTable;
98      hTable.setHeaders(StringVector(headers));
```

(program continues)

Program 11-22 *(continued)*

```
99      hTable.addRow(StringVector(person1));
100     hTable.addRow(StringVector(person2));
101
102     // Open a file and write the HTML code to the file
103     ofstream outFile("c:\\temp\\table.html");
104     outFile << hTable;
105     outFile.close();
106
107     // Write the same HTML code to the screen for ease of viewing
108     cout << hTable;
109     // Use the default browser to view generated HTML table
110     system("c:\\temp\\table.html");
111
112     return 0;
113 }
```

Program Output as Displayed in Browser

Name	Address	Phone
Mike Sane	1215 Mills St	630-555-1293
Natasha Upenski	513 Briarcliff Ln	412-555-1004

Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. If a member variable is declared _____, all objects of that class share that variable.
2. Static member variables are defined _____ the class.
3. A(n) _____ member function cannot access any nonstatic member variables in its own class.
4. A static member function may be called _____ any instances of its class are defined.
5. A(n) _____ function is not a member of a class, but has access to the private members of the class.
6. A(n) _____ tells the compiler that a specific class will be declared later in the program.
7. _____ is the default behavior when an object is assigned the value of another object of the same class.
8. A(n)_____ is a special constructor, called whenever a new object is initialized with another object's data.
9. _____ is a special built-in pointer that is automatically passed as a hidden argument to all nonstatic member functions.

10. An operator may be _____ to work with a specific class.
11. When the _____ operator is overloaded, its function must have a dummy parameter.
12. Making an instance of one class a member of another class is called _____.
13. Object composition is useful for creating a(n) _____ relationship between two classes.
14. A constructor that takes a single parameter of a type different from the class type is a _____ constructor.
15. The class `Stuff` has both a copy constructor and an overloaded = operator. Assume that `blob` and `clump` are both instances of the `Stuff` class. For each of the statements, indicate whether the copy constructor or the overloaded = operator will be called.

```
Stuff blob = clump;
clump = blob;
blob.operator=(clump);
showValues(blob);   // Blob is passed by value.
```

16. Explain the programming steps necessary to make a class's member variable static.
17. Explain the programming steps necessary to make a class's member function static.
18. Consider the following class declaration:

```
class Thing
{
   private:
     int x;
     int y;
     static int z;
   public:
     Thing()
         { x = y = z; }
     static void putThing(int a)
         { z = a; }
};
int Thing:: z = 0:
```

Assume a program containing the class declaration defines three `Thing` objects with the following statement:

```
Thing one, two, three;
```

A) How many separate instances of the `x` member exist?
B) How many separate instances of the `y` member exist?
C) How many separate instances of the `z` member exist?
D) What value will be stored in the `x` and `y` members of each object?
E) Write a statement that will call the `putThing` member function *before* the `Thing` objects are defined.

19. Describe the difference between making a class a member of another class (object composition) and making a class a friend of another class.
20. What is the purpose of a forward declaration of a class?

21. Explain why memberwise assignment can cause problems with a class that contains a pointer member.
22. Explain why a class's copy constructor is called when an object of that class is passed by value into a function.
23. Explain why the parameter of a copy constructor must be a reference.
24. Assume a class named `Bird` exists. Write the header for a member function that overloads the = operator for that class.
25. Assume a class named `Dollars` exists. Write the headers for member functions that overload the prefix and postfix ++ operators for that class.
26. Assume a class named `Yen` exists. Write the header for a member function that overloads the < operator for that class.
27. Assume a class named `Length` exists. Write the header for a member function that overloads the stream insertion << operator for that class.
28. Assume a class named `Collection` exists. Write the header for a member function that overloads the [] operator for that class.
29. Explain why a programmer would want to overload operators rather than use regular member functions to perform similar operations.

Find the Error

30. Each of the following class declarations has errors. Locate as many as you can.

A)
```
class Box
{
    private:
       double width;
       double length;
       double height;
    public:
       Box(double w, l, h)
          { width = w; length = l; height = h; }
       Box(Box b) // Copy constructor
          { width = b.width;
            length = b.length;
            height = b.height; }

  ... Other member functions follow ...
};
```

B)
```
class Circle
{
    private:
        double diameter;
        int centerX;
        int centerY;
    public:
        Circle(double d, int x, int y)
            { diameter = d; centerX = x; centerY = y;   }
        // Overloaded = operator
        void Circle=(Circle &right)
```

```
            { diameter = right.diameter;
              centerX = right.centerX;
              centerY = right.centerY; }

        ... Other member functions follow ...
};
```

C)
```
class Point
  {
      private:
          int xCoord;
          int yCoord;
      public:
          Point (int x, int y)
              { xCoord = x; yCoord = y; }
          // Overloaded + operator
          void operator+(const &Point Right)
             { xCoord += right.xCoord;
               yCoord += right.yCoord;
             }

       ... Other member functions follow ...
};
```

D)
```
class Box
  {
      private:
         double width;
         double length;
         double height;
      public:
         Box(double w, l, h)
             { width = w; length = l; height = h; }
         // Overloaded prefix ++ operator
         void operator++()
            {   ++width; ++length; }
         // Overloaded postfix ++ operator
         void operator++()
            {width++; length++; }

       ... Other member functions follow ...
  };
```

E)
```
class Yard
   {
      private:
         double length;
      public:
          Yard(double l)
             { length = l; }
          // double conversion function
          void operator double()
             { return length; }

          ... Other member functions follow ...
   };
```

Fill-in-the-Blank

31. A derived class inherits the _____ of its base class.
32. The base class named in the following line of code is _____ .

```
class Pet : public Dog
```

33. The derived class named in the following line of code is _____ .

```
class Pet : public Dog
```

34. In the following line of code, the class access specification for the base class is _____.

```
class Pet : public Dog
```

35. In the following line of code, the class access specification for the base class is _____.

```
class Pet : Fish
```

36. Protected members of a base class are like _____ members, except they may be accessed by derived classes.
37. Complete the following table by filling in private, protected, public, or inaccessible in the right-hand column:

In a private base class, this base class MEMBER access specification…	…becomes this access specification in the derived class.
private	
protected	
public	

38. Complete the following table by filling in private, protected, public, or inaccessible in the right-hand column:

In a protected base class, this base class MEMBER access specification…	…becomes this access specification in the derived class.
private	
protected	
public	

39. Complete the following table by filling in private, protected, public, or inaccessible in the right-hand column:

In a public base class, this base class MEMBER access specification…	…becomes this access specification in the derived class.
private	
protected	
public	

40. When both a base class and a derived class have constructors, the base class's constructor is called _____ (first/last).
41. When both a base class and a derived class have destructors, the base class's destructor is called _____ (first/last).
42. An overridden base class function may be called by a function in a derived class by using the _____ operator.

Find the Errors

43. Each of the following class declarations and/or member function definitions has errors. Find as many as you can.

A)
```
class Car, public Vehicle
{
   public:
      Car();
      ~Car();
    protected:
      int passengers;
}
```

B)
```
class Truck, public : Vehicle, public
{
   private:
      double cargoWeight;
   public:
      Truck();
      ~Truck();
};
```

Soft Skills

44. Your company's software is a market leader, but is proving difficult to maintain because it was written in C without using object-oriented concepts. Customers have identified problems with the software that must be fixed immediately, and have pointed out features in competitors' products that they want you to support. The best solution will require a complete OOP redesign and subsequent implementation, but will take three years. Write a memo to company management outlining your recommendation for the course of action the company should pursue.

Programming Challenges



1. Check Writing

Design a class `Numbers` that can be used to translate whole dollar amounts in the range 0 through 9999 into an English description of the number. For example, the number 713 would be translated into the string *seven hundred thirteen*, and 8203 would be translated into *eight thousand two hundred three*.

The class should have a single integer member variable

```
int number;
```

and a collection of static string members that specify how to translate key dollar amounts into the desired format. For example, you might use static strings such as

```
string lessThan20[ ] =
    {"zero", "one", …, "eighteen", "nineteen" };
string hundred = "hundred";
string thousand = "thousand";
```

The class should have a constructor that accepts a nonnegative integer and uses it to initialize the `Numbers` object. It should have a member function `print()` that prints the English description of the `Numbers` object. Demonstrate the class by writing a main program that asks the user to enter a number in the proper range and then prints out its English description.

2. Day of the Year

Assuming that a year has 365 days, write a class named `DayOfYear` that takes an integer representing a day of the year and translates it to a string consisting of the month followed by day of the month. For example,

Day 2 would be *January 2*

Day 32 would be *February 1*

Day 365 would be *December 31*.

The constructor for the class should take as parameter an integer representing the day of the year, and the class should have a member function `print()` that prints the day in the month-day format. The class should have an integer member variable to represent the day, and should have static member variables of type `string` to assist in the translation from the integer format to the month–day format.

Test your class by inputting various integers representing days and printing out their representation in the month–day format.

3. Day of the Year Modification

Modify the `DayOfYear` class, written in an earlier Programming Challenge, to add a constructor that takes two parameters: a string representing a month and an integer in the range 0 through 31 representing the day of the month. The constructor should then initialize the integer member of the class to represent the day specified by the month and day of month parameters. The constructor should terminate the program with an appropriate error message if the number entered for a day is outside the range of days for the month given.

Add the following overloaded operators:

++ prefix and postfix increment operators. These operators should modify the `DayOfYear` object so that it represents the next day. If the day is already the end of the year, the new value of the object will represent the first day of the year.

-- prefix and postfix decrement operators. These operators should modify the `DayOfYear` object so that it represents the previous day. If the day is already the first day of the year, the new value of the object will represent the last day of the year.



VideoNote
Solving the Number of Days Worked Problem

4. Number of Days Worked

Design a class called `NumDays`. The class's purpose is to store a value that represents a number of work hours and convert it to a number of days. For example, 8 hours would

be converted to 1 day, 12 hours would be converted to 1.5 days, and 18 hours would be converted to 2.25 days. The class should have a constructor that accepts a number of hours, as well as member functions for storing and retrieving the hours and days. The class should also have the following overloaded operators:

- *The addition operator* `+`. The number of hours in the sum of two objects is the sum of the number of hours in the individual objects.
- *The subtraction operator* `-`. The number of hours in the difference of two objects X and Y is the number of hours in X minus the number of hours in Y.
- *Prefix and postfix Increment operators* `++`. The number of hours in an object is incremented by 1.
- *Prefix and postfix decrement operators* `--`. The number of hours in an object is decremented by 1.

myCodeMate

5. Palindrome Testing

A palindrome is a string that reads the same backward as forward. For example, the words *mom*, *dad*, *madam* and *radar* are all palindromes. Write a class `Pstring` that is derived from the STL `string` class. The `Pstring` class adds a member function

```
bool isPalindrome( )
```

that determines whether the string is a palindrome. Include a constructor that takes an STL `string` object as parameter and passes it to the `string` base class constructor. Test your class by having a main program that asks the user to enter a string. The program uses the string to initialize a `Pstring` object and then calls `isPalindrome()` to determine whether the string entered is a palindrome.

You may find it useful to use the subscript operator `[]` of the string class: if `str` is a string object and `k` is an integer, then `str[k]` returns the character at position `k` in the string.

6. String Encryption

Write a class `EncryptableString` that is derived from the STL `string` class. The `Encryptable` string class adds a member function

```
void encrypt( )
```

That encrypts the string contained in the object by replacing each letter with its successor in the ASCII ordering. For example, the string *baa* would be encrypted to *cbb*. Assume that all characters that are part of an `EncryptableString` object are letters `a`, .., `z` and `A`, .., `Z`, and that the successor of `z` is a and the successor of `Z` is `A`. Test your class with a program that asks the user to enter strings that are then encrypted and printed.

7. Corporate Sales

A corporation has six divisions, each responsible for sales to different geographic locations. Design a `DivSales` class that keeps sales data for a division, with the following members:

- An array with four elements for holding four quarters of sales figures for the division
- A private static variable for holding the total corporate sales for all divisions for the entire year.

- A member function that takes four arguments, each assumed to be the sales for a quarter. The value of the arguments should be copied into the array that holds the sales data. The total of the four arguments should be added to the static variable that holds the total yearly corporate sales.
- A function that takes an integer argument within the range of 0 to 3. The argument is to be used as a subscript into the division quarterly sales array. The function should return the value of the array element with that subscript.

Write a program that creates an array of six `DivSales` objects. The program should ask the user to enter the sales for four quarters for each division. After the data is entered, the program should display a table showing the division sales for each quarter. The program should then display the total corporate sales for the year.

Input Validation: Only accept positive values for quarterly sales figures.

8. HTML Table of Names and Scores

Write a class whose constructor takes a vector of `Student` objects, where the each `Student` has a name of type `string` and a score of type `int`. The class internally stores the data passed to it in its constructor. The class should have an overloaded output operator that outputs its data in the form of an HTML table. Make up suitable input and use it to test your class.

CHAPTER 12

More About Characters, Strings, and the `string` Class

TOPICS

12.1 C-Strings
12.2 Library Functions for Working with C-Strings
12.3 Conversions Between Numbers and Strings
12.4 Character Testing
12.5 Character Case Conversion
12.6 Writing Your Own C-String Handling Functions
12.7 More About the C++ `string` Class
12.8 Creating Your Own `String` Class
12.9 Advanced Software Enterprises Case Study
12.10 Tying It All Together: *Program Execution Environments*

12.1 C-Strings

CONCEPT: A C-string is a sequence of characters stored in consecutive memory locations and terminated by a null character.

In C++, a C-string is a sequence of characters stored in consecutive memory locations and terminated by a null character. Recall that the null character is the character whose ASCII code 0. In a program, the null character is usually written '\0'. It is also common to use the integer 0 or the constant `NULL` to denote the null character in a program. Thus, all of the following statements store the null character into a character variable:

```
char ch1, ch2, ch3;
ch1 = '\0';
ch2 = 0;
ch3 = NULL;
```

Because an array is a sequence of consecutive memory locations that store values of the same type, a C-string is really a null-terminated array of characters. C-strings can appear in a program in one of three forms:

- "Hard-coded" string literals
- Programmer-defined arrays of character
- Pointers to character

Regardless of which one of the three forms a C-string appears in a program, a C-string is always a null-terminated array of characters, and is represented in the program by a pointer to the first character in the array. In other words, the type of a C-string is

```
char *
```

that is, the type of a C-string is *pointer to char.*

String Literals

String literals, also called *string constants*, are written directly into the program as a sequence of characters enclosed in double quotes: For example,

```
"What is your name?"
"Bailey"
```

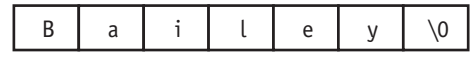
are both string literals. Here is an example of a program that uses a string literal.

Program 12-1

```
1 // This program illustrates string literals.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7    cout << "Bailey";
8    return 0;
9 }
```

Let us look at Program 12-1 to see how the compiler handles C-strings that come in the form of string literals. When the compiler encounters the literal string `"Bailey"`, it allocates an array of seven characters in the data portion of the program. The compiler then stores the six characters of the string `"Bailey"` in the first six entries of the array, storing the null character in the last entry, as shown in Figure 12-1. Finally, the compiler passes a value of type `char *`, the address of the first character of the array, to the `cout` object.

Figure 12-1



Program 12-2 illustrates the fact that a string literal is regarded by the compiler as a value of type `char *`, that is, it is a pointer to a character.

Program 12-2

```
1  // This program demonstrates that string literals
2  // are pointers to char.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // Define variables that are pointers to char.
9     char *p, *q;
10
11    // Assign string literals to the pointers to char.
12    p = "Hello ";
13    q = "Bailey";
14
15    // Print the pointers as C-strings!
16    cout << p << q << endl;
17
18    // Print the pointers as C-strings and as addresses.
19    cout << p  << " is stored at " << int(p) << endl;
20    cout << q  << " is stored at " << int(q) << endl;
21
22    // A string literal can be treated as a pointer!
23    cout << int ("Another string literal");
24    return 0;
25 }
```

Program Output

```
Hello Bailey
Hello  is stored at 4309447
Bailey is stored at 4309454
4309461
```

The first two assignments of Program 12-2 show that string literals are pointers to `char` by assigning them to variables of type `char *`. The pointers `p` and `q` then hold the addresses of the two string literals. By casting the pointers to `int`, we can see where in memory the string literals are stored. Notice that in this case, the compiler has stored all string literals in the program in consecutive memory locations.

Programmer-Defined Arrays of Character

String literals can only hold C-strings that are hard-coded into the program. To have a C-string whose characters are read from the keyboard, or from a file, you must explicitly define an array to hold the characters of the C-string. In doing this, you should make sure that you allocate an additional entry in the array for the terminating null character. For example, if your C-string will be at most 19 characters long, you will need to allocate an array of 20 characters, as in

```
char company[20];
```

As in the case of literals, the compiler will represent the C-string by the address of the first character of the string, in this case, the array identifier. Recall from chapter 8 that an array

identifier without the brackets is interpreted by the compiler to be the address of the first entry of the array.

A C-string defined as an array can be given its value by initializing it with a literal string, or by reading characters into it from the keyboard or a file, or by copying characters into the array one character at a time. Here are some examples of initialization:

```
char company[20] = "Robotic Systems, inc.";
char corporation[ ] = "C. K. Graphics";
```

When initializing an array with a string literal in this manner, the size of the array in the array definition is optional. If not specified, the compiler will set the size to one more than the number of characters in the initializing literal string (thus allowing room for the null terminator).

As described in Chapter 3, C-strings defined as arrays can be read and written using the various objects, operators, and member functions of the input and output stream classes. A C-string stored as a programmer-defined array can be processed using standard subscript notation. Program 12-3 is an example. It outputs a string one character at a time, and stops when it finds the null terminator. It uses the `getline` member function, covered in Chapter 3, to read the string to be output.

Program 12-3

```
1  // This program cycles through a character array, displaying
2  // each element until a null terminator is encountered.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int LENGTH = 80;   // Maximum length for string
9     char line[LENGTH];       // Array of char
10
11    // Read a string into the character array.
12    cout  << "Enter a sentence of no more than "
13          <<  LENGTH-1 << " characters:\n";
14    cin.getline(line, LENGTH);
15    cout  << "The sentence you entered is:\n";
16
17    // Loop through the array printing each character.
18    for(int index = 0; line[index] != '\0'; index++)
19    {
20       cout << line[index];
21    }
22    return 0;
23 }
```

Program Output with Example Input Shown in Bold

```
Enter a sentence of no more than 79 characters:
C++ is challenging but fun![Enter]
The sentence you entered is:
C++ is challenging but fun!
```

Pointers to char

As we have seen, C-strings can be represented as string literals, or as arrays of characters. Both of these methods allocate an array and then use the address of the array as a pointer to char to actually represent the string. The difference between the two is that in the first case, the array used to store the string is allocated implicitly by the compiler, whereas in the second, the array is explicitly allocated by the programmer.

The third method of representing a C-string uses a pointer to char to point to a C-string whose storage has already been allocated by one of the other two methods. Here are some examples of using C-strings in this way:

```
char name[20] = "John Q. Public";
char *p;
p = name;               // Point to an existing C-string
cout << p << endl;      // Print
p = "Jane Doe";         // Point to another C-string
cout << p << endl;      // Print
```

A major advantage in using a pointer variable to represent a C-string is the ability to make the pointer point to different C-strings.

Another way to use a pointer to a char as a C-string is to define the pointer and then set it to point to dynamically allocated storage returned by new operator. This is illustrated in Program 12-4.

Program 12-4

```
1  // This program illustrates dynamic allocation
2  // of storage for C-strings.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int NAME_LENGTH = 50;  // Maximum length
9     char *pname;                 // Address of array
10
11    // Allocate the array.
12    pname = new char[NAME_LENGTH];
13
14    // Read a string.
15    cout << "Enter your name: ";
16    cin.getline(pname, NAME_LENGTH);
17
18    // Display the string.
19    cout << "Hello " << pname;
20    return 0;
21 }
```

Program Output with Example Input Shown in Bold

```
Enter your name: George[Enter]
Hello George
```

A common mistake when using pointers to char as C-strings is using the pointer when it does not point to a properly allocated C-string. For example, the code

```
char *pname;
cout << "Enter your name: ";
cin  >> pname;
```

is erroneous because the program tries to read a string into the memory location pointed to by pname, when pname has not been properly initialized.

12.2 Library Functions for Working with C-Strings

CONCEPT: The C++ library provides many functions for working with C-strings.

The C++ library provides many functions that can be used to work with C-strings. There are functions for determining the length of a string, for concatenating two strings, for comparing two strings, and for searching for the occurrence of one string within another. You must include the cstring header file to use these functions.

The strlen Function

The strlen function is passed a C-string as its argument, and returns the length of the string. This is the number of characters up to, but not including, the null terminator. For example, in the code segment

```
char str[20] = "Hello";
int length = strlen(str);
```

the variable length will have the number 5 stored in it.

The length of a string should not be confused with the size of the array holding it. Remember, the only information passed to strlen is the beginning of the C-string. It doesn't know where the array ends, so it looks for the null terminator to indicate the end of the string.

Passing C-String Arguments

Because C-strings are pointers to char, C-string handling functions take parameters that are arrays of char, or equivalently, pointers to char. The C-string can be passed to the function in any one of the three forms that a C-string can take:

- A literal string
- The name of an array that stores the C-string
- A pointer variable holding the address of the C-string

The strcat Function

Another example of a C-string handling function is strcat. The strcat function takes two strings as parameters and concatenates them, returning a single string that consists of all the characters of the first string followed by the characters of the second string. Here is an example of its use:

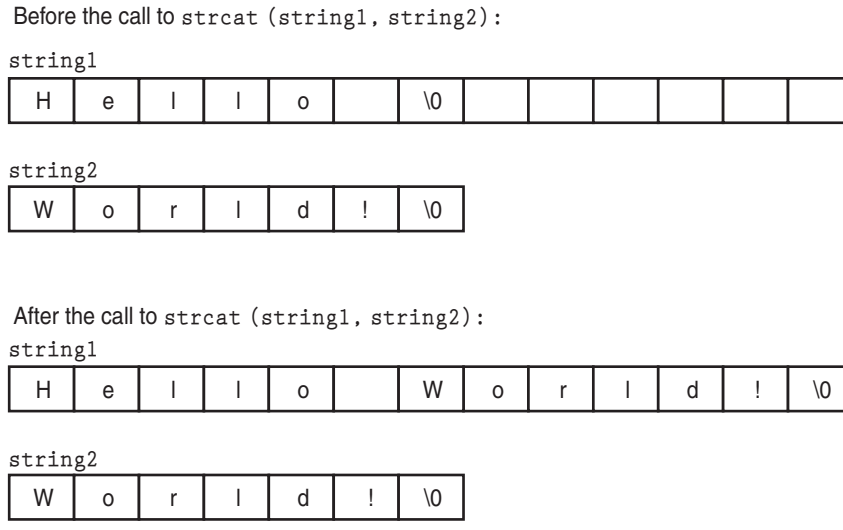
```
char string1[13] = "Hello ";
char string2[7] = "World!";
cout << string1 << endl;
cout << string2 << endl;
strcat(string1, string2);
cout << string1 << endl;
```

These statements will produce the following output:

```
Hello
World!
Hello World!
```

The `strcat` function copies the contents of `string2` to the end of `string1`. In this example, `string1` contains the string "Hello " before the call to `strcat`. After the call, it contains the string "Hello World!". Figure 12-2 shows the contents of both arrays before and after the function call.

Figure 12-2



Notice the last character in `string1` (before the null terminator) is a space. The `strcat` function doesn't insert a space, so it's the programmer's responsibility to make sure one is already there, if needed. It's also the programmer's responsibility to make sure the array holding `string1` is large enough to hold `string1` plus `string2` plus a null terminator. Here is a program segment that uses the `sizeof` operator to test an array's size before `strcat` is called:

```
if (sizeof(string1) >= (strlen(string1) + strlen(string2) + 1))
    strcat(string1, string2);
else
    cout << "String1 is not large enough for both strings.\n";
```



WARNING! If the array holding the first string isn't large enough to hold both strings, `strcat` will overflow the boundaries of the array.

The strcpy Function

Recall from Chapter 8 that one array cannot be assigned to another with the = operator. Each individual element must be assigned, usually inside a loop. The strcpy function, which you have already encountered in Chapter 3, can be used to copy one string to another. Here is an example of its use:

```
char name[20];
strcpy(name, "Albert Einstein");
```

The strcpy function's arguments are both C-strings. The second C-string is copied to the address specified by the first C-string argument.

If anything is already stored in the location referenced by the first argument, it is overwritten, as shown in the following program segment:

```
char string1[10] = "Hello", string2[10] = "World!";
cout << string1 << endl;
cout << string2 << endl;
strcpy(string1, string2);
cout << string1 << endl;
cout << string2 << endl;
```

Here is the output:

```
Hello
World!
World!
World!
```



WARNING! Being true to C++'s nature, strcpy performs no bounds checking. The array specified by the first argument will be overflowed if it isn't large enough to hold the string specified by the second argument.

Comparing C-Strings

The assignment and relational operators work with the C++ string class because they have been overloaded to work with that class. However, just as the assignment operator cannot be used to assign to C-strings, the relational operators <=, <, >, >=, !=, and == cannot be used to compare C-strings. This is because when used with C-strings, these operators compare the addresses at which the C-strings are stored instead of comparing the actual sequence of characters that comprise the C-strings. Program 12-5 shows the incorrect result of trying to compare C-strings using the equality operator.

Program 12-5

```
1 // This program illustrates that you cannot compare
2 // C-strings with relational operators. Although it
3 // appears to test the strings for equality, that is
4 // not what happens.
5 #include <iostream>
6 using namespace std;
7
```

(program continues)

Program 12-5 *(continued)*

```
 8 int main()
 9 {
10    // Two arrays for holding two strings.
11    const int LENGTH = 40;
12    char firstString[LENGTH], secondString[LENGTH];
13
14    // Read two strings.
15    cout << "Enter a string: ";
16    cin.getline(firstString, LENGTH);
17    cout << "Enter another string: ";
18    cin.getline(secondString, LENGTH);
19
20    // Attempt to compare the two strings using ==.
21    if    (firstString == secondString)
22          cout << "You entered the same string twice.\n";
23    else
24          cout << "The strings are not the same.\n";
25
26    return 0;
27 }
```

Program Output with Example Input Shown in Bold

```
Enter a string: Alfonso[Enter]
Enter another string: Alfonso[Enter]
The strings are not the same.
```

Although two identical strings may be entered, Program 12-5 will always report that they are not equal. This is because the expression

```
firstString == secondString
```

used in the `if` statement compares the memory addresses of the two arrays instead of comparing the strings of characters stored at those addresses. Because these addresses are different, the comparison yields a result of false. In fact, in C++, even the comparison

```
"abc" == "abc"
```

will usually yield a result of false. This is because most compilers do not check to see if a string literal has been encountered before, and will store the two strings at different memory addresses. The compiler will then compare the two different addresses, giving a value of false for the result.

The `strcmp` Function

To properly compare C-strings, you should use the library function `strcmp`. This function takes two C-strings as parameters and returns an integer that indicates how the two strings compare to each other. Its prototype,

```
int strcmp(char *string1, char *string2);
```

indicates that the function takes two C-strings as parameters (recall that `char *` is the type of C-string) and returns an integer result. The value of the result is set according to the following convention:

- The result is *zero* if the two strings are *equal* on a character by character basis
- The result is *negative* if `string1` comes *before* `string2` in alphabetical order
- The result is *positive* if `string1` comes *after* `string2` in alphabetical order

Here is an example of the use of `strcmp` to determine if two strings are equal:

```
if (strcmp(string1, string2) == 0)
    cout << "The strings are equal";
else
    cout << "The strings are not equal";
```

Program 12-5, which incorrectly tested two C-strings with a relational operator, can be correctly rewritten with the `strcmp` function, as shown in Program 12-6.

Program 12-6

```
1  // This program correctly tests two C-strings for
2  // equality with the strcmp function.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     // Two arrays for two strings.
10    const int LENGTH = 40;
11    char firstString[LENGTH], secondString[LENGTH];
12
13    // Read two strings.
14    cout << "Enter a string: ";
15    cin.getline(firstString, LENGTH);
16    cout << "Enter another string: ";
17    cin.getline(secondString, LENGTH);
18
19    // Compare the strings for equality with strcmp.
20    if  (strcmp(firstString, secondString) == 0)
21        cout << "You entered the same string twice.\n";
22    else
23        cout << "The strings are not the same.\n";
24
25    return 0;
26 }
```

Program Output with Example Input Shown in Bold

```
Enter a string: Alfonso[Enter]
Enter another string: Alfonso[Enter]
You entered the same string twice.
```

The function `strcmp` is case sensitive when it compares strings. If the user enters “Dog” and “dog” in Program 12-6, it will report they are not the same. Most compilers provide

nonstandard versions of `strcmp` that perform case-insensitive comparisons. For instance, Borland C++ has the `stricmp` function. It works identically to `strcmp` except the case of the characters is ignored.

Program 12-7 is a more practical example of how `strcmp` can be used. It asks the user to enter the part number of the stereo they wish to purchase. The part number contains digits, letters, and a hyphen, so it must be stored as a string. Once the user enters the part number, the program displays the price of the stereo.

Program 12-7

```
1  // This program uses strcmp to compare the string entered
2  // by the user with the valid stereo part numbers.
3  #include <iostream>
4  #include <cstring>
5  #include <iomanip>
6  using namespace std;
7
8  int main()
9  {
10    // Price of parts.
11    const double A_PRICE = 249.0,
12                 B_PRICE = 299.0;
13
14    // Character array for part number.
15    const int PART_LENGTH = 8;
16    char partNum[PART_LENGTH];
17
18    // Instruct the user to enter a part number.
19    cout << "The stereo part numbers are:\n";
20    cout << "\tBoom Box, part number S147-29A\n";
21    cout << "\tShelf Model, part number S147-29B\n";
22    cout << "Enter the part number of the stereo you\n";
23    cout << "wish to purchase: ";
24
25    // Read a part number of at most 8 characters.
26    cin >> setw(9);
27    cin >> partNum;
28
29    // Determine what user entered using strcmp
30    // and print its price.
31    cout << showpoint << fixed << setprecision(2);
32    if (strcmp(partNum, "S147-29A") == 0)
33        cout << "The price is $" << A_PRICE << endl;
34    else if (strcmp(partNum, "S147-29B") == 0)
35        cout << "The price is $" << B_PRICE << endl;
36    else
37        cout << partNum << " is not a valid part number.\n";
38
39    return 0;
40 }
```

(program continues)

Program 12-7 *(continued)*

Program Output with Example Input Shown in Bold

```
The stereo part numbers are:
     Boom Box, part number S147-29A
     Shelf Model, part number S147-29B
Enter the part number of the stereo you
wish to purchase: S147-29B[Enter]
The price is $299.00
```

Using ! with strcmp

Some programmers prefer to use the logical NOT operator with `strcmp` when testing strings for equality. Since 0 is considered logically `false`, the `!` operator converts that value to `true`. The expression `!strcmp(string1, string2)` will return `true` when both strings are the same, and `false` when they are different. The two following statements perform the same operation:

```
if (strcmp(firstString, secondString) == 0)
if (!strcmp(firstString, secondString))
```

Sorting Strings

Programs are frequently written to print alphabetically sorted lists of items. For example, consider a department store computer system that keeps customers' names and addresses in a file. The names do not appear in the file alphabetically but in the order the operator entered them. If a list were to be printed in this order, it would be very difficult to locate any specific name. The list would have to be sorted before it was printed.

Because the value returned by `strcmp` is based on the relative alphabetic order of the two strings being compared, it can be used in programs that sort strings. Program 12-8 asks the user to enter two names, which are then printed in alphabetic order.

Program 12-8

```
1  // This program uses the return value of strcmp to
2  // alphabetically sort two strings entered by the user.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     // Two arrays to hold two strings.
10    const int NAME_LENGTH = 30;
11    char name1[NAME_LENGTH], name2[NAME_LENGTH];
12
13    // Read two strings.
14    cout << "Enter a name (last name first): ";
15    cin.getline(name1, NAME_LENGTH);
16    cout << "Enter another name: ";
17    cin.getline(name2, NAME_LENGTH);
18
```

(program continues)

Program 12-8 *(continued)*

```
19    // Print the two strings in alphabetical order.
20    cout << "Here are the names sorted alphabetically:\n";
21    if  (strcmp(name1, name2) < 0)
22        cout << name1 << endl << name2 << endl;
23    else if (strcmp(name1, name2) > 0)
24        cout << name2 << endl << name1 << endl;
25    else
26        cout << "You entered the same name twice!\n";
27
28    return 0;
29 }
```

Program Output with Example Input Shown in Bold

```
Enter a name (last name first): Smith, Richard[Enter]
Enter another name: Jones, John[Enter]
Here are the names sorted alphabetically:
Jones, John
Smith, Richard
```

Table 12-1 summarizes the string-handling functions discussed here, as well as others. (All the functions listed require the `cstring` header file.)

Table 12-1 (See your C++ reference manual for more information on these functions.)

Function	Description
`strlen`	Accepts a C-string as an argument. Returns the length of the C-string (not including the null terminator. *Example Usage:* `len = strlen(name);`
`strcat`	Accepts two C-strings as arguments. The function appends the contents of the second string to the first C-string. (The first string is altered, the second string is left unchanged.) *Example Usage:* `strcat(string1, string2);`
`strcpy`	Accepts two C-strings as arguments. The function copies the second C-string to the first C-string. The second C-string is left unchanged. *Example Usage:* `strcpy(string1, string2);`
`strncpy`	Copies at most `n` characters of `string2` to `string1`. If `string2` has fewer than `n` characters, then `string1` is padded with '\0' characters until a total of `n` characters have been written to it. If `string2` has `n` or more characters, then the first `n` characters are copied and `string1` is not null-terminated. *Example Usage:* `strncpy(string1, string2, n);`
`strcmp`	Accepts two C-string arguments. If `string1` and `string2` are the same, this function returns 0. If `string2` is alphabetically greater than `string1`, it returns a negative number. If `string2` is alphabetically less than `string1`, it returns a positive number. *Example Usage:* `if (strcmp(string1, string2))`
`strstr`	Searches for the first occurrence of `string2` in `string1`. If an occurrence of `string2` is found, the function returns a pointer to it. Otherwise, it returns a `NULL` pointer (address 0). *Example Usage:* `cout << strstr(string1, string2);`

The strstr Function

The last function in Table 12-1 is strstr, which searches for a string inside of a string. For instance, it could be used to search for the string "seven" inside the larger string "Four score and seven years ago." The function's first argument is the string to be searched, and the second argument is the string to look for. If the function finds the second string inside the first, it returns the address of the occurrence of the second string within the first string. Otherwise it returns the address 0, or the NULL address. Here is an example:

```
char array[] = "Four score and seven years ago";
char *strPtr;
cout << array << endl;
strPtr = strstr(array, "seven");  // search for "seven"
cout << strPtr << endl;
```

In the preceding program segment strstr will locate the string "seven" inside the string "Four score and seven years ago." It will return the address of the first character in "seven", which will be stored in the pointer variable strPtr. If run as part of a complete program, the segment will display the following:

```
Four score and seven years ago
seven years ago
```

The strstr function can be useful in any program that must locate information inside one or more strings. Program 12-9, for example, stores a database of product numbers and descriptions in an array of C-strings. It allows the user to look up a product description by entering all or part of its product number.

Program 12-9

```
1  // This program uses the strstr function to search an array
2  // of strings for a name.
3  #include <iostream>
4  #include <string> // For strstr
5  using namespace std;
6
7  int main()
8  {
9     const int   N_ITEMS = 5,   // Maximum number of items
10                S_LENGTH = 31; // maximum length of description
11
12    // Array of product descriptions.
13    char prods[5][S_LENGTH] = {"TV327  31 inch Television",
14                               "CD257  CD Player",
15                               "TA677  Answering Machine",
16                               "CS109  Car Stereo",
17                               "PC955  Personal Computer"};
18
19
```

(program continues)

Program 12-9 *(continued)*

```
20    char lookUp[S_LENGTH];  // For user input
21    char *strPtr = NULL;    // Result from strstr
22
23    // Get user input.
24    cout << "\tProduct Database\n\n";
25    cout << "Enter a product number to search for: ";
26    cin.getline(lookUp, S_LENGTH);
27
28    // Search for the string.
29    int index = 0;
30    while(index < N_ITEMS)
31    {
32       strPtr = strstr(prods[index], lookUp);
33       if (strPtr != NULL)
34          break;
35       index++;
36    }
37
38    // Output the result of the search.
39    if (strPtr == NULL)
40       cout << "No matching product was found.\n";
41    else
42       cout << prods[index] << endl;
43
44    return 0;
45 }
```

Program Output with Example Input Shown in Bold

```
      Product Database

Enter a product to search for: CD257[Enter]
CD257  CD Player
```

Program Output with Other Example Input Shown in Bold

```
      Product Database

Enter a product to search for: CS[Enter]
CS109  Car Stereo
```

Program Output with Other Example Input Shown in Bold

```
      Product Database

Enter a product to search for: AB[Enter]
No matching product was found.
```

In Program 12–9, the loop in lines 29–36 cycles through each C-string in the array calling the following statement:

```
strPtr = strstr(prods[index], lookUp);
```

The `strstr` function searches the string referenced by `prods[index]` for the name entered by the user, which is stored in `lookUp`. If `lookUp` is found inside `prods[index]`,

the function returns its address. In that case, the following `if` statement causes the loop to terminate:

```
if (strPtr != NULL)
       break;
```

Outside the loop, the following `if-else` statement determines if the string entered by the user was found in the array. If not, it informs the user that no matching product was found. Otherwise, the product number and description are displayed:

```
if (strPtr == NULL)
   cout << "No matching product was found.\n";
else
   cout << prods[index] << endl;
```



Checkpoint

12.1 Write a short description of each of the following functions:

A) `strlen`
B) `strcat`
C) `strcpy`
D) `strncpy`
E) `strcmp`
F) `strstr`

12.2 What will the following program segment display?

```
char dog[] = "Fido";
cout << strlen(dog) << endl;
```

12.3 What will the following program segment display?

```
char string1[16] = "Have a ";
char string2[9] = "nice day";
strcat(string1, string2);
cout << string1 << endl;
cout << string2 << endl;
```

12.4 Write a statement that will copy the string "Beethoven" to the array `composer`.

12.5 When complete, the following program skeleton will search for the string "`Windy`" in the array `place`. If `place` contains "`Windy`" the program will display the message "`Windy found.`" Otherwise it will display "`Windy not found.`"

```
#include <iostream>
// include any other necessary header files
int main()
{
   char place[] = "The Windy City";
   // Complete the program. It should search the array place
   // for the string "Windy" and display the message "Windy
   // found" if it finds the string. Otherwise, it should
   // display the message "Windy not found."
}
```

12.6 Indicate whether the following `strcmp` function calls will return 0, a negative number, or a positive number. Refer to the ASCII table in Appendix A if necessary.

A) `strcmp("ABC", "abc");`
B) `strcmp("Jill", "Jim");`
C) `strcmp("123", "ABC");`
D) `strcmp("Sammy", "Sally");`

12.7 Complete the `if` statements in the following program skeleton.

```
#include <iostream>
   using namespace std;

   int main()
   {
      const int LENGTH = 20;
      char iceCream[LENGTH];
      cout << "What flavor of ice cream do you like best? ";
      cout << "Chocolate, Vanilla, or Pralines and Pecan? ";
      cin.getline(iceCream, LENGTH);
      cout << "Here is the number of fat grams for a half ";
      cout << "cup serving:\n";
      //
      // Finish the following if-else statement
      // so the program will select the ice cream entered
      // by the user.
      //
      if (/* insert your code here */)
         cout << "Chocolate: 9 fat grams.\n";
      else if (/* insert your code here */)
         cout << "Vanilla: 10 fat grams.\n";
      else if (/* insert your code here */)
         cout << "Pralines and Pecan: 14 fat grams.\n";
      else
         cout << "That's not one of our flavors!\n";
      return 0;
   }
```

12.3 Conversions Between Numbers and Strings

CONCEPT: The C++ libraries provide classes that can be used to convert a string representation of a number to numeric form and vice versa.

There is a difference between a number that is stored as a string and one stored as a numeric value. The string "2679" isn't a number: it is a sequence of ASCII codes of the characters that form the individual digits of the number 2679. Because the string "2679" is not a number, the compiler will not allow arithmetic operations such as addition, multiplication, and division to be applied to it. Strings that represent numbers must first be converted to numeric form before they can be used with arithmetic operators.

String representations of numbers arise naturally in computing, most often during input and output of numbers. When a user enters a number at a keyboard, the number is entered in its string form as a sequence of characters (digits) typed by the user. In C++, such a number is usually read via the stream extraction operator >>. This operator automatically performs conversions as needed before storing into a variable of numeric type. During output, the reverse conversion from numeric to string is performed by the stream output operator <<.

There are times when a string such as "2679" that is already stored in memory as a C-string or string object needs to be converted to numeric form. There are also times when a number already stored in numeric form in memory needs to be converted into its string representation. C++ has two classes, `ostringstream` and `istringstream`, that can be used to perform string/numeric conversions. The class `ostringstream` is a subclass of `ostream` (the class that `cout` belongs to) and uses the stream insertion operator << to convert numeric values to string. Objects of type `ostringstream` work the same way that `cout` and file objects do, except that instead of writing to the screen or to a file, `ostringstream` writes its data to a string object contained inside it. Each time you use << on the `ostringstream` object, it performs any numeric-to-string conversions necessary and appends the result to the end of its string. In addition to supporting all the member functions and operators of the `ostream` class, `ostringstream` objects support the `str` member functions shown in Table 12-2.



VideoNote
Converting Strings to Numbers

The `istringstream` class derives from `istream`. It contains a string object inside it that functions as an input stream that can be "read" from. The input stream can be set by the `istringstream` constructor when the object is created, or can be set by calling the `str(string s)` function after the object has been created. The stream extraction operator >> reads from the enclosed string and converts from string to numeric where necessary. Member functions of `istringstream` are also listed in Table 12-2. You must include the `sstream` header file in your programs to use these classes.

Table 12-2 Member Functions of `ostringstream` and `istringstream` Classes

Member Function	Description
`istringstream(string s)`	Constructor for `istringstream`: sets the initial value of the input stream for the object. Example: `istringstream istr("50 64 28");`
`ostringstream(string s)`	Constructor for `ostringstream`: sets the initial value of the output stream for the object. Example: `ostringstream ostr("50 64 28");`

`string str()`	Returns the string contained in the `ostringstream` or `istringstream` object.
	Example:
	`string is = istr.str();`
	`string os = ostr.str();`

`void str(string &s)`	Sets the string that serves as the input or output stream for the object.
	Example:
	`ostr.str("50 64 28");`
	`istr.str("50 64 28");`

Program 12-10 demonstrates the use of these classes.

Program 12-10

```
1  // This program illustrates the use of sstream objects.
2  #include <sstream>
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9      string str = "John  20 50";  // String to read from
10     char *cstr = "Amy 30 42";    // Cstring to read from
11     istringstream istr1(str);    // istr1 will read from str
12     istringstream istr2;         // istr2 will read from cstr
13     ostringstream ostr;          // The ostringstream object
14
15     string name;
16     int score1, score2, average_score;
17
18     // Read name and scores and compute average then write to ostr
19     istr1 >> name >> score1 >> score2;
20     average_score = (score1 + score2)/2;
21     ostr <<  name << " has average score " << average_score << "\n";
22
23     // Set istr2 to read from the C string and repeat the above
24     istr2.str(cstr);
25     istr2 >> name >> score1 >> score2;
26     average_score = (score1 + score2)/2;
27     ostr <<  name << " has average score " << average_score << "\n";
28
29     // Switch to hexadecimal output on ostr
30     ostr << hex;
31
32     // Write Amy's scores in hexadecimal
33     ostr << name << "'s scores in hexadecimal are: " << score1
34          << " and " << score2 << "\n";
35
36     // Extract the string from ostr and print it to the screen.
37     cout << ostr.str();
38
39     return 0;
40 }
```

Program Output

```
John has average score 35
Amy has average score 36
Amy's scores in hexadecimal are: 1e and 2a
```

Notice that these classes have the full power of `ostream` and `istream` objects, including the ability to convert numbers to string using different bases such as octal and hexadecimal. Although these classes are the preferred way to convert between numeric and string forms,

you should also be aware of the C++ library functions shown in Table 12-3. These functions work on C-strings only.

Table 12-3 (See your C++ reference manual for more information on these functions.)

Function	Description
`atoi`	Accepts a C-string as an argument. The function converts the C-string to an integer and returns that value. *Example Usage:* `num = atoi("4569");`
`atol`	Accepts a C-string as an argument. The function converts the C-string to a `long` integer and returns that value. *Example Usage:* `lnum = atol("500000");`
`atof`	Accepts a C-string as an argument. The function converts the C-string to a `double` and returns that value. Use this function to convert a C-string to a `float` or `double`. *Example Usage:* `fnum = atof("3.14159");`
`itoa`	Converts an integer to a C-string. The first argument, `value`, is the integer. The result will be stored at the location pointed to by the second argument, `string`. The third argument, `base`, is an integer. It specifies the numbering system that the converted integer should be expressed in (8 = octal, 10 = decimal, 16 = hexadecimal, etc.). *Example Usage:* `itoa(value, string, base);`

The `atoi` function converts a string to an integer. It accepts a C-string argument and returns the converted integer value. Here is an example of how to use it:

```
int num;
num = atoi("1000");
```

In these statements, `atoi` converts the string "1000" into the integer 1000. Once the variable `num` is assigned this value, it can be used in mathematical operations or any task requiring a numeric value.



NOTE: The `atoi` function as well as the others discussed in this section require that the `cstdlib` header file be included.

The `atol` function works just like `atoi`, except the return value is a `long` integer. Here is an example:

```
long bigNum;
bigNum = atol("500000");
```

As expected, the `atof` function accepts a C-string argument and converts it to a `double`. The numeric `double` value is returned, as shown here:

```
double fnum;
fnum = atof("12.67");
```



NOTE: If a string that cannot be converted to a numeric value is passed to any of these functions, the function's behavior is undefined by C++. Many compilers, however, will perform the conversion process until an invalid character is encountered. For example, `atoi("123x5")` might return the integer 123. It is possible that these functions will return 0 if they cannot successfully convert their argument.

The itoa function is similar to atoi, but it works in reverse. It converts a numeric integer into a string representation of the integer. The itoa function accepts three arguments: the integer value to be converted, a pointer to the location in memory where the string is to be stored, and a number that represents the base of the converted value. Here is an example:

```
char numArray[10];
itoa(1200, numArray, 10);
cout << numArray << endl;
```

This program segment converts the integer value 1200 to a string. The string is stored in the array numArray. The third argument, 10, means the number should be written in decimal, or base 10 notation. The output of the cout statement is

```
1200
```



WARNING! As always, C++ performs no array bounds checking. Make sure the array whose address is passed to itoa is large enough to hold the converted number, including the null terminator.

Now let's look at Program 12-11, which uses a string-to-number conversion function, atoi. It allows the user to enter a series of values, or the letters Q or q to quit. The average of the numbers is then calculated and displayed.

Program 12-11

```
1  // This program demonstrates the strcmp and atoi functions.
2  #include <iostream>
3  #include <cstring> // For strcmp
4  #include <cstdlib> // For atoi
5  using namespace std;
6
7  int main()
8  {
9     // Array used to read numbers in string form.
10    const int LENGTH = 20;
11    char input[LENGTH];
12
13    int   total = 0,  // Running total
14          count = 0;  // Number of numbers read
15    double average;   // Average
16
17    // Read numbers and computer total of numbers.
18    cout << "This program will average a series of numbers.\n";
19    cout << "Enter the first number or Q to quit: ";
20    cin.getline(input, LENGTH);
21    while((strcmp(input,"Q")!= 0)&&(strcmp(input,"q")!= 0))
22    {
23       // Keep a running total.
24       total += atoi(input);
25
26       // Keep track of how many numbers are entered.
27       count++;
28
```

(program continues)

Program 12-11 *(continued)*

```
29       // Are there more?
30       cout << "Enter the next number or Q to quit: ";
31    cin.getline(input,LENGTH);}
32    }
33
34    // Compute and print average.
35    if (count != 0)
36    {
37       average = double(total) / count;
38       cout << "average: " << average << endl;
39    }
40
41    return 0;
42 }
```

Program Output with Example Input Shown in Bold

```
This program will average a series of numbers.
Enter the first number or Q to quit: 74[Enter]
Enter the next number or Q to quit: 98[Enter]
Enter the next number or Q to quit: 23[Enter]
Enter the next number or Q to quit: 54[Enter]
Enter the next number or Q to quit: Q[Enter]
Average: 62.25
```

Recall that `strcmp` compares two C-strings. If they are identical, it returns 0. Otherwise a nonzero value is returned. The following `while` statement uses `strcmp` to determine if the string in `input` is either "`Q`" or "`q`".

```
while ((strcmp(input, "Q") != 0)&&(strcmp(input, "q") != 0))
```

If the user hasn't entered "`Q`" or "`q`" the program uses `atoi` to convert the string in `input` to an integer and adds its value to `total` with the following statement:

```
total += atoi(input);  // Keep a running total
```

The user is then asked for the next number. When all the numbers are entered, the user terminates the loop by entering "`Q`" or "`q`". If one or more numbers are entered, their average is displayed.



Checkpoint

12.8 Write a short description of each of the following functions.
- A) `atoi`
- B) `atol`
- C) `atof`
- D) `itoa`

12.9 Write a statement that will convert the C-string "`10`" to an integer and store the result in the variable `num`.

12.10 Write a statement that will convert the C-string "`100000`" to a `long` and store the result in the variable `num`.

12.11 Write a statement that will convert the string "7.2389" to a double and store the result in the variable num.

12.12 Write a statement that will convert the integer 127 to a string, stored in base 10 notation in the array value.

12.4 Character Testing

CONCEPT: The C++ library provides several functions for testing characters.

The C++ library provides several functions that allow you to test the value of a character. These functions test a single int argument and return either true or false.* The integer argument is the ASCII code of a character. For example, the following program segment uses the isupper function to determine if the character passed as an argument is an uppercase letter. If it is, the function returns true. Otherwise, it returns false.

```
char letter = 'a';
if (isupper(letter))
   cout << "Letter is uppercase.\n";
else
   cout << "Letter is not uppercase.\n";
```

Since the variable letter, in this example, contains a lowercase character, isupper returns false. The if statement will cause the message "Letter is not uppercase" to be displayed.

Table 12-4 lists several character-testing functions. Each of these is implemented in the cctype header file, so be sure to include that file when using the functions.

Table 12-4 Character-Testing Functions

Character Function	Description
isalpha	Returns true (a nonzero number) if the argument is a letter of the alphabet. Returns false if the argument is not a letter.
isalnum	Returns true (a nonzero number) if the argument is a letter of the alphabet or a digit. Otherwise it returns false.
isdigit	Returns true (a nonzero number) if the argument is a digit from 0 to 9. Otherwise it returns false.
islower	Returns true (a nonzero number) if the argument is a lowercase letter. Otherwise, it returns false.
isprint	Returns true (a nonzero number) if the argument is a printable character (including a space). Returns false otherwise.

(table continues)

* These functions actually return an int value. The return value is nonzero to indicate true, or zero to indicate false.

Table 12-4 Character-Testing Functions *(continued)*

Character Function	Description
`ispunct`	Returns `true` (a nonzero number) if the argument is a printable character other than a digit, letter, or space. Returns `false` otherwise.
`isupper`	Returns `true` (a nonzero number) if the argument is an uppercase letter. Otherwise, it returns `false`.
`isspace`	Returns `true` (a nonzero number) if the argument is a whitespace character, otherwise it returns `false`. Whitespace characters are any of the following: space ' ' vertical tab '\v' newline '\n' tab '\t'

Program 12-12 uses several of the functions shown in Table 12-4. It asks the user to input a character and then displays various messages, depending on the return value of each function.

Program 12-12

```
1  // This program demonstrates some of the character
2  // testing functions.
3  #include <iostream>
4  #include <cctype>
5  using namespace std;
6
7  int main()
8  {
9     char input;
10
11    cout  << "Enter any character: ";
12    cin.get(input);
13    cout  << "The character you entered is: "
14          << input << endl;
15    cout  << "Its ASCII code is: "
16          << static_cast<int>(input) << endl;
17    if (isalpha(input))
18        cout << "That's an alphabetic character.\n";
19    if (isdigit(input))
20        cout << "That's a numeric digit.\n";
21    if (islower(input))
22        cout << "The letter you entered is lowercase.\n";
23    if (isupper(input))
24        cout << "The letter you entered is uppercase.\n";
25    if (isspace(input))
26        cout << "That's a whitespace character.\n";
27    return 0;
28 }
```

(program continues)

Program 12-12 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter any character: A[Enter]
The character you entered is: A
Its ASCII code is: 65
That's an alphabetic character.
The letter you entered is uppercase.
```

Program Output with Other Example Input Shown in Bold

```
Enter any character: 7[Enter]
The character you entered is: 7
Its ASCII code is: 55
That's a numeric digit.
```

Program 12-13 shows a more practical application of the character-testing functions. It tests a seven-character customer number to determine whether it is in the proper format.

Program 12-13

```
1  // This program tests a customer number to determine whether
2  // it is in the proper format.
3  #include <iostream>
4  #include <cctype>
5  using namespace std;
6
7  // Function prototype
8  bool testNum(char []);
9
10 const int NUM_LENGTH = 8;
11 const int ALPHA_LENGTH = 3;
12 int main()
13 {
14    char customer[NUM_LENGTH];   // Holds customer number
15
16    // Get the customer number from the user.
17    cout << "Enter a customer number in the form ";
18    cout << "LLLNNNN\n";
19    cout << "(LLL = letters and NNNN = numbers): ";
20    cin.getline(customer, NUM_LENGTH);
21
22    // Test the customer number and print results.
23    if (testNum(customer))
24       cout << "That's a valid customer number.\n";
25    else
26    {
27       cout << "That is not the proper format of the ";
28       cout << "customer number.\nHere is an example:\n";
29       cout << "   ABC1234\n";
30    }
31    return 0;
32 }
33
```

(program continues)

Program 12-13 *(continued)*

```
34 //*********************************************************
35 // Definition of function testNum. This function accepts a *
36 // character array as its argument and tests its contents  *
37 // for a valid customer number.                            *
38 //*********************************************************
39 bool testNum(char custNum[])
40 {
41    // Test the leading characters for alphabetic letters.
42    for (int k = 0; k < ALPHA_LENGTH; k++)
43    {
44       if (!isalpha(custNum[k]))
45          return false;
46    }
47
48    // Test the trailing characters for numeric digits.
49    for (int k = ALPHA_LENGTH; k < NUM_LENGTH - 1; k++)
50    {
51       if (!isdigit(custNum[k]))
52          return false;
53    }
54    return true;
55 }
```

Program Output with Example Input Shown in Bold

```
Enter a customer number in the form LLLNNNN
(LLL = letters and NNNN = numbers): RQS4567[Enter]
That's a valid customer number.
```

Program Output with Other Example Input Shown in Bold

```
Enter a customer number in the form LLLNNNN
(LLL = letters and NNNN = numbers): AX467T9[Enter]
That is not the proper format of the customer number.
Here is an example:
   ABC1234
```

In Program 12-13, the customer number is expected to consist of alphabetic letters followed by digits. The `testNum` function accepts an array argument and tests leading characters with the following loop:

```
for (int k = 0; k < ALPHA_LENGTH; k++)
{
      if (!isalpha(custNum[k]))
         return false;
}
```

The `isalpha` function returns `true` if its argument is an alphabetic character. The `!` operator is used in the `if` statement to determine whether the tested character is *not* alphabetic. If this is so for any of the first three characters, the function `testNum` returns false.

Likewise, the next four characters are tested to be digits with the following loop:

```
for (int k = ALPHA_LENGTH; k < NUM_LENGTH - 1; k++)
{
   if (!isdigit(custNum[k]))
      return false;
}
```

The `isdigit` function returns `true` if its argument is the character representation of any of the digits 0 through 9. Once again, the `!` operator is used to determine if the tested character is *not* a digit. If this is so for any of the last four characters, the function `testNum` returns `false`. If the customer number is in the proper format, the function will cycle through both the loops without returning `false`. In that case, the last line in the function is the `return true` statement, which indicates the customer number is valid.

12.5 Character Case Conversion

CONCEPT: The C++ library offers functions for converting a character to upper- or lowercase.

The C++ library provides two functions, `toupper` and `tolower`, for converting the case of a character. The functions are described in Table 12-5. (These functions are prototyped in the header file `cctype`, so be sure to include it.)

Table 12-5 Character Conversion Functions

Function	Description
`toupper`	Returns the uppercase equivalent of its argument.
`tolower`	Returns the lowercase equivalent of its argument.

Both of the functions in Table 12-5 accept an integer representing the ASCII code of a character to be converted and return an integer representing the ASCII code of the uppercase or lowercase equivalent. Each function simply returns its argument if the conversion cannot be made or is unnecessary. For example, if the argument to `toupper` is not a lowercase letter, then `toupper` simply returns its argument unchanged. The prototypes of these functions are

```
int toupper(int ch);
int tolower(int ch);
```

The fact that these functions return an integer means that a statement such as

```
cout << toupper('a'); // prints 65
```

will print the integer that is the ASCII code of `'A'` rather than printing the character `'A'` itself. To get it to print the character, you can cast the return value to a `char`, as in

```
cout << static_cast<char>(toupper('a')); // prints A
```

or you can assign the return value to a character variable first and then print the character:

```
char ch = toupper('a');
cout << ch;             // prints  A
```

As mentioned above, the `toupper` function returns its argument unchanged if the argument is not a letter:

```
cout << static_cast<char>(toupper('&')); // prints &
```

or is a letter but it is already uppercase:

```
cout << static_cast<char>(toupper('B');  // prints B
```

Because the arguments to `toupper` and `tolower` are passed by value, the functions do not change their actual parameters: Instead, they simply return the uppercase or lowercase equivalent of their argument. For example, in the following program segment, the variable `letter` is set to the value 'A'. The `tolower` function returns the character 'a', but `letter` still contains 'A'.

```
char letter = 'A';
char ch = tolower(letter);
cout << ch << endl;
cout << letter;
```

These statements will cause the following to be displayed:

```
a
A
```

Program 12-14 demonstrates the `toupper` function in a loop that lets the user enter either Y or N.

Program 12-14

```
1  // This program calculates the area of a circle. It
2  // asks the user if he or she wishes to continue. A
3  // loop that demonstrates the toupper function repeats
4  // until the user enters 'y', 'Y', 'n', or 'N'.
5  #include <iostream>
6  #include <cctype>
7  #include <iomanip>
8  using namespace std;
9
10 int main()
11 {
12    const double PI = 3.14159;
13    double radius;
14    char response;    // User response
15
16    // Output formatting.
17    cout << setprecision(2) << fixed;
18
19    // Instructions to the user.
20    cout << "This program calculates the area of a circle.\n";
21
```

(program continues)

Program 12-14 *(continued)*

```
22    // Keep reading radii and computing area
23    // until the user has had enough.
24    do
25    {
26       // Get radius of a circle and print its area.
27       cout << "Enter the circle's radius: ";
28       cin >> radius;
29       cout << "The area is " << (PI * radius * radius);
30       cout << endl;
31
32       // Are there more circles?
33       do
34       {
35          cout << "Calculate another? (Y or N) ";
36          cin >> response;
37          response = toupper(response);
38       } while (response != 'Y' && response != 'N');
39    } while (response == 'Y');
40    return 0;
41 }
```

Program Output with Example Input Shown in Bold

```
This program calculates the area of a circle.
Enter the circle's radius: 10[Enter]
The area is 314.16
Calculate another? (Y or N) b[Enter]
Calculate another? (Y or N) y[Enter]
Enter the circle's radius: 1[Enter]
The area is 3.14
Calculate another? (Y or N) n[Enter]
```



Checkpoint

12.13 Write a short description of each of the following functions:

A) `isalpha`
B) `isalnum`
C) `isdigit`
D) `islower`
E) `isprint`
F) `ispunct`
G) `isupper`
H) `isspace`
I) `toupper`
J) `tolower`

12.14 Write a statement that will convert the contents of the `char` variable `big` to lowercase. The converted value should be assigned to the variable `little`.

12.15 Write an `if` statement that will display the word "digit" if the variable `ch` contains a numeric digit. Otherwise, it should display "Not a digit."

12.16 What is the output of the following statement?

```
cout << static_cast<char>(toupper(tolower('A')));
```

12.17 Write a loop that asks the user "Do you want to repeat the loop or quit?(R/Q)". The loop should repeat as long as the user enters an R (in upper- or lowercase) and terminate when the user enters a Q (in upper- or lowercase).

12.6 Writing Your Own C-String Handling Functions

CONCEPT: You can design your own specialized functions for manipulating strings.



VideoNote
Writing a C-String Handling Function

By being able to pass arrays as arguments, you can write your own functions for processing C-strings. For example, Program 12-15 uses a function to copy a C-string from one array to another.

Program 12-15

```
1  // This program uses a function to copy
2  // a string into an array.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype.
7  void stringCopy(char [], char []);
8
9  int main()
10 {
11    // Define two arrays of char.
12    const int S_LENGTH = 30;
13    char dest[S_LENGTH], source[S_LENGTH];
14
15    // Read a string into a source array.
16    cout  << "Enter a string with no more than "
17          << S_LENGTH - 1 << " characters:\n";
18    cin.getline(source, S_LENGTH);
19
20    // Copy it into a destination array and print.
21    stringCopy(dest, source);
22    cout << "The string you entered is:\n" << dest << endl;
23    return 0;
24 }
25
26 //************************************************
27 // Definition of the stringCopy function.        *
28 // This function accepts two character arrays as  *
29 // arguments. The function assumes the two arrays *
30 // contain C-strings. The contents of the second  *
31 // array are copied to the first array.           *
32 //************************************************
```

(program continues)

Program 12-15 *(continued)*

```
33 void stringCopy(char destStr[], char sourceStr[])
34 {
35    int index = 0;
36
37    // Copy one character at a time till we come to
38    // the null terminator.
39    while (sourceStr[index] != '\0')
40    {
41       destStr[index] = sourceStr[index];
42       index++;
43    }
44    destStr[index] = '\0';
45 }
```

Program Output with Example Input Shown in Bold

```
Enter a string with no more than 29 characters:
Thank goodness it's Friday![Enter]
The string you entered is:
Thank goodness it's Friday!
```

Notice the function `stringCopy` in Program 12-15 does not accept an argument indicating the size of the arrays. It simply copies the characters from the source string to the destination until it encounters a null terminator in the source string. When the null terminator is found, the loop has reached the end of the C-string. The last statement in the function assigns a null terminator (the `'\0'` character) to the end of `string2`, so it is properly terminated.



WARNING! Since the `stringCopy` function doesn't know the size of the destination array, it's the programmer's responsibility to make sure the destination array is large enough to hold the source string array.

Program 12-16 uses another C-string handling function: `nameSlice`. The program asks the user to enter his or her first and last names, separated by a space. The function searches the string for the space and replaces it with a null terminator. In effect, this cuts off the last name of the string.

Program 12-16

```
1  // This program uses the function nameSlice
2  // to "cut" off the last name of a string that
3  // contains the user's first and last names.
4  #include <iostream>
5  using namespace std;
6
7  void nameSlice(char []);   // Function prototype
8
9  int main()
10 {
11
12 // Define array of char to hold name.
13    const int NAME_LENGTH = 41;
```

(program continues)

Program 12-16 *(continued)*

```
14    char name[NAME_LENGTH];
15
16    // Get user's first and last names.
17    cout << "Enter your first and last names, separated ";
18    cout << "by a space:\n";
19    cin.getline(name, NAME_LENGTH);
20
21    // Slice off the last name and print what is left.
22    nameSlice(name);
23    cout << "Your first name is: " << name << endl;
24    return 0;
25 }
26
27 //*****************************************************
28 // Definition of function nameSlice. This function    *
29 // accepts a character array as its argument. It      *
30 // scans the array looking for a space. When it finds *
31 // one, it replaces it with a null terminator.        *
32 //*****************************************************
33 void nameSlice(char userName[])
34 {
35    // Look for the end of the first name, indicated
36    // by a space or a null terminator.
37    int k = 0;
38    while (userName[k] != ' ' && userName[k] != '\0')
39       k++;
40
41    // Insert null terminator.
42    if (userName[k] == ' ')
43       userName[k] = '\0';
44 }
```

Program Output with Example Input Shown in Bold

```
Enter your first and last names, separated by a space:
Jimmy Jones[Enter]
Your first name is: Jimmy
```

The following loop in `nameSlice` starts at the first character in the array and scans the string, searching for either a space or a null terminator:

```
while (userName[k] != ' ' && userName[k] != '\0')
   k++;
```

If the character in `userName[k]` isn't a space or the null terminator, `k` is incremented, and the next character is examined. With the example input "`Jimmy Jones`," the loop finds the space separating "`Jimmy`" and "`Jones`" at `userName[5]`. When the loop stops, `k` is set to 5. This is illustrated in Figure 12-3.



NOTE: The loop stops if it encounters a null terminator so it will not go beyond the boundary of the array if the user didn't enter a space.

Once the loop has finished, `userName[k]` will either contain a space or a null terminator. If it contains a space, the following `if` statement whose action is illustrated in Figure 12-4, replaces it with a null terminator:

```
if (userName[k] == ' ')
   userName[k] = '\0';
```

The new null terminator now becomes the end of the string.

Figure 12-3

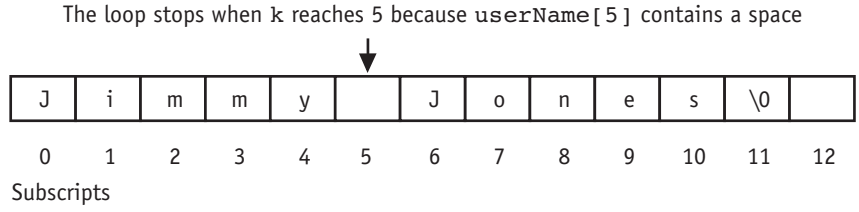
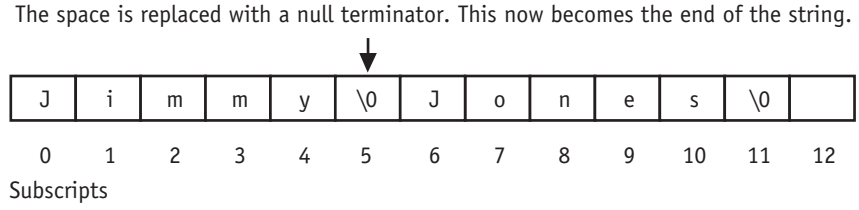


Figure 12-4



Using Pointers to Pass C-String Arguments

Pointers are extremely useful for writing functions that process C-strings. If the starting address of a string is passed into a pointer parameter variable, it can be assumed that all the characters, from that address up to the byte that holds the null terminator are part of the string. (It isn't necessary to know the length of the array that holds the string.)

Program 12-17 demonstrates a function, `countChars`, that uses a pointer to count the number of times a specific character appears in a C-string.

Program 12-17

```
1  // This program demonstrates a function, countChars,
2  // that counts the number of times a specific
3  // character appears in a string.
4  #include <iostream>
5  using namespace std;
6
7  // Function prototype
8  int countChars(char *, char);
9
```

(program continues)

Program 12-17 *(continued)*

```
10 int main()
11 {
12    // Define array to hold the string.
13    const int S_LENGTH = 51;
14    char userString[S_LENGTH];
15
16    char letter;   // User input.
17
18    // Read the string and the letter to count.
19    cout  << "Enter a string (up to "
20          << S_LENGTH-1 << " characters): ";
21    cin.getline(userString, S_LENGTH);
22    cout << "Enter a character and I will tell you how many\n;
23    cout << "times it appears in the string: ";
24    cin >> letter;
25
26    // Output the results of the letter count.
27    cout << letter << " appears ";
28    cout << countChars(userString, letter) << " times.\n";
29    return 0;
30 }
31
32 //*******************************************************
33 // Definition of countChars. The parameter strPtr is    *
34 // a pointer that points to a string. The parameter     *
35 // ch is a character that the function searches for     *
36 // in the string. The function returns the number of    *
37 // times the character appears in the string.           *
38 //*******************************************************
39 int countChars(char *strPtr, char ch)
40 {
41    int count = 0;
42    while (*strPtr != '\0')
43    {
44       if (*strPtr == ch)
45          count++;
46       strPtr++;
47    }
48    return count;
49 }
```

Program Output with Example Input Shown in Bold

```
Enter a string (up to 50 characters): Starting Out With C++[Enter]
Enter a character and I will tell you how many
times it appears in the string: t[Enter]
t appears 4 times.
```

In the function `countChars`, `strPtr` points to the C-string that is to be searched and `ch` contains the character to look for. The `while` loop repeats as long the character `strPtr` points to is not the null terminator:

```
while (*strPtr != '\0')
```

Inside the loop, the following if statement compares the character that strPtr points to with the character in ch:

```
if (*strPtr == ch)
```

If the two are equal, the variable count is incremented. (count keeps a running total of the number of times the character appears.) The last statement in the loop is

```
strPtr++;
```

This statement increments the address in strPtr. This causes strPtr to point to the next character in the string. Then the loop starts over. When strPtr finally reaches the null terminator, the loop terminates and the function returns the value in count.



Checkpoint

12.18 What is the output of the following program?

```
#include <iostream>
using namespace std;

// Function prototype
void mess(char []);

int main()
{
   char stuff[] = "Tom Talbert Tried Trains";

   cout << stuff << endl;
   mess(stuff);
   cout << stuff << endl;
   return 0;
}

// Definition of function mess
void mess(char str[])
{
   int step = 0;

   while (str[step] != '\0')
   {
      if (str[step] == 'T')
         str[step] = 'D';
      step++;
   }
}
```

12.7 More About the C++ string Class

From an ease-of-programming point of view, the standard library string class offers several advantages over C-strings. As you have seen throughout this text, the string class has several member functions and overloaded operators. These simplify tasks,

such as locating a character or string within a string, that are difficult and tedious to perform with C-strings. In this section we review some basic operations with strings, then discuss more of the string class's member functions.

Any program using the string class must #include the string header file. String objects may then be created using any of several constructors. Program 12-18 demonstrates two string class constructors: the default constructor, and, the convert constructor that converts a C-string into a string object.

Program 12-18

```
1  // This program demonstrates some C++ string class constructors.
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  int main()
7  {
8     string greeting;               // Default constructor
9     string name("William Smith"); // Convert constructor
10
11    greeting = "Hello ";
12    cout << greeting << name << endl;
13    return 0;
14 }
15
```

Program Output

```
Hello William Smith
```

Other examples of the use of string constructors are given in Table 12-6.

Table 12-6 String Constructors

Definition	Description
`string address;`	Defines an empty string object named address.
`string name("William Smith");`	Defines a string object named name, initialized with "William Smith."
`string person1(person2);`	Defines a string object named person1, which is a copy of person2. person2 may be either a string object or character array.
`string set1(set2, 5);`	Defines a string object named set1, which is initialized to the first five characters in the character array set2.
`string lineFull('z', 10);`	Defines a string object named lineFull initialized with 10 'z' characters.
`string firstName(fullName, 0, 7);`	Defines a string object named firstName, initialized with a substring of the string fullName. The substring is seven characters long, beginning at position 0.

Notice in Program 12-18 the use of the = operator to assign a value to the string object. The string class overloads several operators, which are described in Table 12-7.

Table 12-7 String Class Operators

Overloaded Operator	Description
>>	Extracts characters from a stream and inserts them into the string. Characters are copied until a whitespace or the end of the input is encountered.
<<	Inserts the string into a stream.
=	Assigns the string on the right to the string object on the left.
+=	Appends a copy of the string on the right to the string object on the left.
+	Returns a string that is the concatenation of the two string operands.
[]	Implements array-subscript notation, as in `name[x]`. A reference to the character in the `x` position is returned.
Relational Operators	Each of the relational operators are implemented: < > <= >= == !=

Program 12-19 demonstrates some of the string operators.

Program 12-19

```
1  // This program demonstrates C++ string class operators
2  #include <iostream>
3  #include <string>
4  using namespace std;
5
6  int main()
7  {
8     string str1, str2, str3;
9
10    // String Assignment.
11    str1 = "ABC";
12    str2 = "DEF";
13    // String concatenation.
14    str3 = str1 + str2;
15
16    // String Output.
17    cout << str1 << endl;
18    cout << str2 << endl;
19    cout << str3 << endl;
20
21    // Short form of concatenation.
22    str3 += "GHI";
23
24    // String Output.
25    cout << str3 << endl;
26    return 0;
27 }
```

(program continues)

Program 12-19 *(continued)*

Program Output

```
ABC
DEF
ABCDEF
ABCDEFGHI
```

The `string` class also has several member functions. For example, the `size` function returns the length of the string. It is demonstrated in the `for` loop in Program 12-20.

Program 12-20

```
1  // This program demonstrates some C++ string
2  // class member functions.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     // Define and set values of string variables.
10    string str1, str2, str3;
11    str1 = "ABC";
12    str2 = "DEF";
13    str3 = str1 + str2;
14
15    // Use the size() string member function
16    // and the overloaded string operator [ ].
17    for (int k = 0; k < str3.size(); k++)
18       cout << str3[k];
19    cout << endl;
20
21    // Use the overloaded string operator <.
22    if (str1 < str2)
23       cout << str1 << " is less than " <<  str2 << endl;
24    else
25       cout << str1 << " is not less than " << str2 << endl;
26    return 0;
27 }
```

Program Output

```
ABCDEF
ABC is less than DEF
```

Table 12-8 lists many of the `string` class member functions and their overloaded variations.

Table 12-8 `string` Class Member Functions

Member Function Example	Description
`theString.append(str);`	Appends `str` to `theString`. `str` can be a string object or character array.
`theString.append(str, x, n);`	`n` number of characters from `str`, starting at position `x`, are appended to `theString`. If `theString` is too small, the function will copy as many characters as possible.
`theString.append(str, n);`	The first `n` characters of the character array `str` are appended to `theString`.
`theString.append(n, 'z');`	Appends `n` copies of `'z'` to `theString`.
`theString.assign(str);`	Assigns `str` to `theString`. The parameter `str` can be a string object or a C-string.
`theString.assign(str, x, n);`	`n` number of characters from `str`, starting at position `x`, are assigned to `theString`. If `theString` is too small, the function will copy as many characters as possible.
`theString.assign(str, n);`	The first `n` characters of the character array `str` are assigned to `theString`.
`theString.assign(n, 'z');`	Assigns `n` copies of `'z'` to `theString`.
`theString.at(x);`	Returns the character at position `x` in the string.
`theString.begin();`	Returns an iterator pointing to the first character in the string. (For more information on iterators, see Chapter 15.)
`theString.capacity();`	Returns the size of the storage allocated for the string.
`theString.clear();`	Clears the string by deleting all the characters stored in it.
`theString.compare(str);`	Performs a comparison like the `strcmp` function with the same return values. `str` can be a string object or a character array.
`theString.compare(x, n, str);`	Compares `theString` and `str`, starting at position `x`, and continuing for `n` characters. The return value is defined as for strcmp. `str` can be a string object or character array.
`theString.copy(str, x, n);`	Copies the character array `str` to `theString`, beginning at position `x`, for `n` characters. If `theString` is too small, the function will copy as many characters as possible.
`theString.c_str():`	Returns the C-string value of the `string` object.
`theString.data();`	Returns a character array containing a null terminated string, as stored in `theString`.
`theString.empty();`	Returns true if `theString` is empty.
`theString.end();`	Returns an iterator pointing to the last character of the string in `theString`. (For more information on iterators, see Chapter 15.)

(table continues)

Table 12-8 `string` Class Member Functions *(continued)*

Member Function Example	Description
`theString.erase(x, n);`	Erases `n` characters from `theString`, beginning at position `x`.
`theString.find(str, x);`	Returns the first position at or beyond position `x` where the string `str` is found in `theString`. The parameter `str` may be either a string object or a C-string. If `str` is not found, a position beyond the end of `theString` is returned.
`theString.find('z', x);`	Returns the first position at or beyond position `x` where `'z'` is found in `theString`.
`theString.insert(x, str);`	Inserts a copy of `str` into `theString`, beginning at position `x`. `str` may be either a string object or a character array.
`theString.insert(x, n, 'z');`	Inserts `'z'` `n` times into `theString` at position `x`.
`theString.length();`	Returns the length of the string in `theString`.
`theString.replace(x, n, str);`	Replaces the `n` characters in `theString` beginning at position `x` with the characters in string object `str`.
`theString.resize(n, 'z');`	Changes the size of the allocation in `theString` to `n`. If `n` is less than the current size of the string, the string is truncated to `n` characters. If `n` is greater, the string is expanded and `'z'` is appended at the end enough times to fill the new spaces.
`theString.size();`	Returns the length of the string in `theString`.
`theString.substr(x, n);`	Returns a copy of a substring. The substring is `n` characters long and begins at position `x` of `theString`.
`theString.swap(str);`	Swaps the contents of `theString` with `str`.

12.8 Creating Your Own `String` Class

CONCEPT: **This section demonstrates some of the programming techniques used to create the C++ `string` class.**

The C++ `string` class automatically handles many of the tedious tasks involved in using strings, such as dynamic memory allocation and bounds checking. It also overloads operators such as + and =, and offers many member functions that ease the job of working with strings. In this section, we create a string data type with much of the functionality of the C++ class. In the process, we see examples of copy constructors and overloaded operators in full action, as well as examples of programming techniques that are useful in the solution of many problems.

The MyString Class

The MyString class defined in this section is an abstract data type for handling strings. It has many of the advantages possessed by the C++ string class provided by the Standard Template Library:

- Memory is dynamically allocated for any string stored in a MyString object. The programmer using this class doesn't need to be concerned with how large to make an array.
- Strings may be assigned to a MyString object with the = operator. The programmer using this class does not have to call the strcpy function.
- One string may be concatenated to another with the += operator. This eliminates the need for the strcat function.
- Strings may be tested for equality with the == operator. The programmer using this class doesn't have to call the strcmp function.

The following program listings show the class implementation.

Contents of mystring.h

```
#ifndef MYSTRING_H
#define MYSTRING_H

#include <iostream>
#include <cstring>  // For string library functions
#include <cstdlib>  // For exit() function
using namespace std;

class MyString {
private:
    char *str;
    int len;
public:
    // Constructors
    MyString() { str = ""; len = 0; }
    MyString(char *);
    MyString(const MyString &);

    // Destructor
    ~MyString() { if (len != 0) delete [] str; }

    // Various member functions and operators
    int length() { return len; }
    char *data() { return str; };
    MyString operator+=(MyString );
    MyString operator=(MyString );

    // Various overloaded operators
    friend bool operator==(MyString, MyString);
    friend bool operator!=(MyString, MyString);
    friend bool operator>(MyString, MyString);
    friend bool operator<(MyString, MyString);
    friend bool operator>=(MyString, MyString);
    friend bool operator<=(MyString, MyString);
    friend ostream & operator<<(ostream &, MyString );
    friend istream & operator>>(istream &, MyString &);
};
#endif
```

Contents of mystring.CPP

```
#include "mystring.h"

//************************************************
// Constructor to initialize the str member      *
// with a C-string constant.                     *
//************************************************
MyString::MyString(char *sptr)
{
    len = strlen(sptr);
    str = new char[len + 1];
    strcpy(str, sptr);
}

//************************************************
// Copy constructor.                             *
//************************************************
MyString::MyString(const MyString &right)
{
    str = new char[right.len + 1];
    strcpy(str, right.str);
    len = right.len;
}

//************************************************
// Overloaded = operator.                        *
//************************************************
MyString MyString::operator=(MyString right)
{
    if (len != 0) delete [] str;
    str = new char[right.len + 1];
    strcpy(str, right.str);
    len = right.len;
    return *this;
}

//************************************************
// Overloaded += operator.                       *
// Concatenates the str member of right to the   *
// str member of the calling object.             *
// Returns the calling object.                   *
//************************************************
MyString MyString::operator+=(MyString right)
{
    char *temp = str;
    str = new char[len + right.len + 1];
    strcpy(str, temp);
    strcat(str, right.str);
    if (len != 0) delete [] temp;
    len += right.len;
    return *this;
}

```

```
53 //*******************************************************
54 // Overloaded == operator.                              *
55 //*******************************************************
56 bool operator==(MyString left, MyString right)
57 {
58     return strcmp(left.str, right.str) == 0;
59 }
60
61 //*******************************************************
62 // Overloaded != operator.                              *
63 //*******************************************************
64 bool operator!=(MyString left, MyString right)
65 {
66     return strcmp(left.str, right.str) != 0;
67 }
68
69 //*******************************************************
70 // Overloaded > operator.                               *
71 //*******************************************************
72 bool operator>(MyString left, MyString right)
73 {
74     return (strcmp(left.str, right.str) > 0);
75 }
76
77 //*******************************************************
78 // Overloaded < operator.                               *
79 //*******************************************************
80 bool operator<(MyString left, MyString right)
81 {
82     return (strcmp(left.str, right.str) < 0);
83 }
84
85 //*******************************************************
86 // Overloaded >= operator.                              *
87 //*******************************************************
88 bool operator>=(MyString left, MyString right)
89 {
90     return (strcmp(left.str, right.str) >= 0);
91 }
92
93 //*******************************************************
94 // Overloaded <= operator.                              *
95 //*******************************************************
96 bool operator<=(MyString left, MyString right)
97 {
98    return  (strcmp(left.str, right.str) <= 0);
99 }
100
101 //*************************************************
102 // Overloaded stream insertion operator (<<).     *
103 //*************************************************
104 ostream &operator<<(ostream &strm, MyString obj)
```

```
105 {
106     strm << obj.str;
107     return strm;
108 }
109
110 //************************************************
111 // Overloaded stream extraction operator (>>).    *
112 //************************************************
113 istream &operator>>(istream &strm, MyString &obj)
114 {
115    // Read the string
116    char buffer[256];
117    strm.getline(buffer, 256);
118    // Invoke the convert constructor and overloaded assignment
119    obj = buffer;
120    return strm;
121 }
```

Constructors and Destructors

Because the `MyString` class has a pointer that is used to dynamically allocate memory, it requires a programmer-defined copy constructor, assignment operator, and destructor. In addition to the copy constructor, there is a default constructor that initializes a `MyString` object to an empty string, and a convert constructor that allows C-strings to be used in place of `MyString` objects in assignment statements. The convert constructor also allows C-strings to be passed to functions and operators that would otherwise require `MyString` objects as parameters. An example of this is line 119 in the `MyString.cpp` file, where in the statement

```
obj = buffer;
```

`obj` is a `MyString` object being assigned the value of the C-string variable `buffer`.

The Overloaded = Operator

The overloaded assignment operator is invoked whenever one `MyString` object is being assigned to another, as in

```
MyString x("Hello");
MyString y;
y = x;
```

It is also called whenever a C-string is assigned to a `MyString` object, as in

```
MyString x;
x = "hello";
```

In this second example, the convert constructor will automatically be called to convert the C-string `"hello"` to a `MyString` object, which is then assigned to the object `x`.

The Overloaded += Operator

This operator concatenates the `Mystring` object on its right to the object on its left. The right operand may be a `MyString` object, or it may be a C-string:

```
MyString x("Hello ");
x += "World";
```

The Overloaded Relational Operators ==, !=, <, <=, >, >=

These operators are overloaded as nonmember functions to preserve the symmetry between the left and right operands and to allow the convert constructor to be called when a C-string is passed for the left operand (parameter). These functions return a Boolean value that is exactly the Boolean value that is obtained when the contained C-strings are passed to `strcmp` and the result is compared to 0. As an example, here is code for operator < :

```
bool operator<(MyString left, MyString right)
{
    return (strcmp(left.str, right.str) < 0);
}
```

Program 12-21 demonstrates the use of the += operator and various other features of the `MyString` class.

Program 12-21

```
1  // This program demonstrates the MyString class. Be sure to
2  // compile this program with mystring.cpp.
3  #include <iostream>
4  #include "mystring.h"
5  using namespace std;
6
7  int main()
8  {
9     MyString object1("This"), object2("is");
10    MyString object3("a test.");
11    MyString object4 = object1;  // Call copy constructor.
12    MyString object5("is only a test.");
13    char string1[] = "a test.";
14
15    cout << "Object1: " << object1 << endl;
16    cout << "Object2: " << object2 << endl;
17    cout << "Object3: " << object3 << endl;
18    cout << "Object4: " << object4 << endl;
19    cout << "Object5: " << object5 << endl;
20    cout << "String1: " << string1 << endl;
21    object1 += " ";
22    object1 += object2;
23    object1 += " ";
24    object1 += object3;
25    object1 += " ";
26    object1 += object4;
27    object1 += " ";
28    object1 += object5;
29    cout << "object1: " << object1 << endl;
30    return 0;
31 }
```

(program continues)

Program 12-21 *(continued)*

Program Output

```
Object1: This
Object2: is
Object3: a test.
Object4: This
Object5: is only a test.
String1: a test.
object1: This is a test. This is only a test.
```

Program 12-22 shows how MyString's relational operators can be used to compare strings with the same ease that numeric data types are compared.

Program 12-22

```
1  // This program demonstrates the MyString class. Be sure to
2  // compile this program with mystring.cpp.
3  #include <iostream>
4  #include "mystring.h"
5  using namespace std;
6
7  int main()
8  {
9     MyString name1("Billy"), name2("Sue");
10    MyString name3("joe");
11    MyString string1("ABC"), string2("DEF");
12
13    cout << "name1: " << name1.getValue() << endl;
14    cout << "name2: " << name2.getValue() << endl;
15    cout << "name3: " << name3.getValue() << endl;
16    cout << "string1: " << string1.getValue() << endl;
17    cout << "string2: " << string2.getValue() << endl;
18    if (name1 == name2)
19        cout << "name1 is equal to name2.\n";
20    else
21        cout << "name1 is not equal to name2.\n";
22    if (name3 == "joe")
23        cout << "name3 is equal to joe.\n";
24    else
25        cout << "name3 is not equal to joe.\n";
26    if (string1 > string2)
27        cout << "string1 is greater than string2.\n";
28    else
29        cout << "string1 is not greater than string2.\n";
30    if (string1 < string2)
31        cout << "string1 is less than string2.\n";
32    else
33        cout << "string1 is not less than string2.\n";
34    if (string1 >= string2)
35        cout << "string1 is greater than or equal to "
36             << "string2.\n";
```

(program continues)

Program 12-22 *(continued)*

```
37      else
38          cout << "string1 is not greater than or equal to "
39               << "string2.\n";
40      if (string1 >= "ABC")
41          cout << "string1 is greater than or equal to "
42               << "ABC.\n";
43      else
44          cout << "string1 is not greater than or equal to "
45               << "ABC.\n";
46      if (string1 <= string2)
47          cout << "string1 is less than or equal to "
48               << "string2.\n";
49      else
50          cout << "string1 is not less than or equal to "
51              << "string2.\n";
52      if (string2 <= "DEF")
53          cout << "string2 is less than or equal to "
54               << "DEF.\n";
55      else
56          cout << "string2 is not less than or equal to "
57          << "DEF.\n";
58      return 0;
59  }
```

Program Output

```
name1: Billy
name2: Sue
name3: joe
string1: ABC
string2: DEF
name1 is not equal to name2.
name3 is equal to joe.
string1 is not greater than string2.
string1 is less than string2.
string1 is not greater than or equal to string2.
string1 is greater than or equal to ABC.
string1 is less than or equal to string2.
string2 is less than or equal to DEF.
```

12.9 Advanced Software Enterprises Case Study

You are a summer intern at Advanced Software Enterprises, and your boss has asked you to develop a function that can format string representations of dollar amounts. Specifically, he wants you to add commas and a dollar sign ($) at appropriate places in the string. For example, when the function is given the string object or C-string with a value of "1084567.89", it should return the string "$1,084,567.89".

After reviewing the `String` class members listed in Table 12-8 and giving the problem some thought, you decide to use the `find` method. This will let you find the index of the

decimal point in the input string. Beginning at that point, you can back up in the string, inserting a comma at every third location. You then finish it off by inserting a $ sign at the beginning. In no time at all, you have the solution and the demonstration program shown in the listing for Program 12-23.

Program 12-23

```
1  // This program demonstrates the use of the string find
2  // and insert member functions.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  string dollarFormat(string );  // Prototype
8
9  int main(void)
10 {
11    string input;    // User input
12
13    // Get the dollar amount from the user.
14    cout << "Enter a dollar amount in the form nnnnn.nn : ";
15    cin >> input;
16
17    // Display the formatted dollar amount.
18    cout << "Formatted amount:   " <<  dollarFormat(input) << endl;
19    return 0;
20 }
21
22 //*************************************************
23 // Returns a $-formatted version of the input string *
24 //*************************************************
25 string dollarFormat(string original)
26 {
27     string formatted = original;
28     int dp = formatted.find('.');  // Position of decimal point
29     int pos = dp;                  // Search for comma position
30     while (pos > 3)
31     {
32        pos = pos - 3;
33        formatted.insert(pos, ",");
34     }
35     formatted.insert(0, "$");
36     return formatted;
37 }
```

Program Output with Example Input Shown in Bold

```
Enter a dollar amount in the form nnnnn.nn : 1084567.89[Enter]
Here is the amount formatted:   $1,084,567.89
```

12.10 Tying It All Together: *Program Execution Environments*

Most operating systems provide every executing program with an *execution environment* consisting of a set of strings of the form

name=value

The *name* part of this equation is called an *environment variable* and the *value* part is used to specify a string value for that particular environment variable. As an example, look at this partial listing of the execution environment of a C++ program running on one of the authors' machines:

```
 1 COMPUTERNAME=GCM-VISTA1
 2 ComSpec=C:\Windows\system32\cmd.exe
 3 HOMEDRIVE=C:
 4 HOMEPATH=\Users\gcm
 5 LOGONSERVER=\\GCM-VISTA1
 6 NUMBER_OF_PROCESSORS=2
 7 OS=Windows_NT
 8 SESSIONNAME=Console
 9 SystemDrive=C:
10 SystemRoot=C:\Windows
11 USERDOMAIN=gcm-Vista1
12 USERNAME=gcm
13 windir=C:\Windows
14 WindowsSdkDir=C:\Program Files\\Microsoft SDKs\Windows\v6.0A\
```

In line 1, `COMPUTERNAME` is the environment variable and `GCM-VISTA1` is the associated value.

A program that examines its execution environment can obtain information about the user currently logged in and about the machine on which it is running. Depending on the operating system, the program can also gather information about the network to which the machine is connected. For example, by examining the above listing, we can tell that the user's login name is `gcm` (line 12), that the user's home folder is `\Users\gcm` (line 4), and that the machine's operating system is a version of Microsoft Windows (line 7). Furthermore, we can tell that the computer has two central processing units (line 6) and that the network name of the machine is `GCM-VISTA1` (line 1).

The operating system stores the program's environment as an array of pointers to C-strings. To mark the end of the array, the system sets the last entry in the array to 0. It then passes the base address of this array to the program. When the program starts executing, the C++ runtime system sets a variable

```
char **environ;
```

to point to the beginning of the environment array.

Basically, the `environ` variable is a global variable defined in library code that is linked with the executable code of your program. In C++, a function can access a global variable defined in a separate file by declaring the variable and prefixing the declaration with the

keyword extern. This means that you can gain access to the environment by including this declaration in your program:

```
extern char ** environ;
```

In Chapter 10 you learned that if environ is a pointer to the beginning of an array of items, you can use the notation

```
environ[k]
```

to access the various components of that array. By starting a variable k at 0 and repeatedly incrementing k, you can step through the array and examine each environment string, as shown in Program 12-24.

Program 12-24

```
1  // This program prints its environment variables.
2  #include <iostream>
3  using namespace std;
4
5  int main(int argc, char** argv)
6  {
7      extern char **environ;   // Needed to access the environment
8
9      int k = 0;
10     while(environ[k] != 0)   // Is  it last C-string in environment?
11     {
12         // Print the string
13         cout << environ[k] << "\n";
14         k++;
15     }
16     return 0;
17 }
```

The output from this program will vary depending on the user running the program and the machine on which the program is running. It will be similar to what is shown at the beginning of this section.

Review Questions and Exercises

Fill-in-the-Blank

1. The _____ function returns true if the character argument is uppercase.
2. The _____ function returns true if the character argument is a letter of the alphabet.
3. The _____ function returns true if the character argument is a digit.
4. The _____ function returns true if the character argument is a whitespace character.
5. The _____ function returns the uppercase equivalent of its character argument.
6. The _____ function returns the lowercase equivalent of its character argument.

7. The _____ file must be included in a program that uses character testing functions.
8. The _____ function returns the length of a string.
9. To _____ two strings means to append one string to the other.
10. The _____ function concatenates two strings.
11. The _____ function copies one string to another.
12. The _____ function searches for a string inside of another one.
13. The _____ function compares two strings.
14. The _____ function copies, at most, `n` number of characters from one string to another.
15. The _____ function returns the value of a string converted to an integer.
16. The _____ function returns the value of a string converted to a `long` integer.
17. The _____ function returns the value of a string converted to a `double`.
18. The _____ function converts an integer to a string.

Algorithm Workbench

19. Write a function whose prototype is

```
char lastChar(char *str)
```

that takes a nonempty C-string as parameter and returns the last character in the string. For example, the call `lastChar("abc")` will return the character `c`.

Predict the Output

For some of the questions, you may need to consult a table of ASCII codes.

20.
```
#include <iostream>
using namespace std;
int main()
{
    cout << ("hello")[1];
    return 0;
}
```

21.
```
#include <iostream>
using namespace std;
int main()
{
    cout << *("hello");
    return 0;
}
```

22.
```
#include <iostream>
using namespace std;
int main()
{
    cout << *("C++ is fun" + 5);
    return 0;
}
```

23.
```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    cout << string("fantastic").size();
    return 0;
}
```

24.
```
#include <iostream>
#include <cctype>
using namespace std;
int main()
{
    cout << toupper('B');
    return 0;
}
```

25.
```
#include <iostream>
using namespace std;
int main()
{
    if ("a" == "a")
       cout << "equal";
    else
       cout << "not equal";
    return 0;
}
```

26.
```
#include <iostream>
#include <cctype>
using namespace std;
int main()
{
    char ch = toupper(65);
    cout << ch;
    return 0;
}
```

27.
```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char name[20] = "abracadabra";
    strcpy(name+4, "sion");
    cout << name;
  return 0;
}
```

28.
```
#include <iostream>
#include <cstring>
using namespace std;
int main()
```

```
{
    char name[20] = "John ";
    *name = '\0';
    strcat(name, "Smith");
    cout << name;
    return 0;
}
```

Find the Errors

29. Each of the following programs or program segments has errors. Find as many as you can.

A)
```
char string[] = "Stop";
if (isupper(string) == "STOP")
   exit(0);
```

B)
```
char numeric[5];
int x = 123;
numeric = atoi(x);
```

C)
```
char string1[] = "Billy";
char string2[] = " Bob Jones";
strcat(string1, string2);
```

D)
```
char x = 'a', y = 'a';
if (strcmp(x, y) == 0)
    exit(0);
```

Soft Skills

30. You are a member of a standardization committee for a new C++ standard, and there is a proposal on the table to drop C-strings from the language and support only the C++ string class. State whether you would oppose or support the proposal and explain why.

Programming Challenges



1. String Length

Write a function that accepts a C-string as an argument and returns the length of the C-string as a result. The function should count the number of characters in the string and return that number. Demonstrate the function in a simple program that asks the user to input a string, passes it to the function, and then displays the function's return value.



VideoNote
Solving the Backward String Problem

2. Backward String

Write a function that accepts a C-string as an argument and displays its contents backwards. For instance, if the string argument is "`Gravity`" the function should display "`ytivarG`". Demonstrate the function in a program that asks the user to input a string and then passes it to the function.

3. Word Counter

Write a function that accepts a C-string as an argument and returns the number of words contained in the string. For instance, if the string argument is "Four score and seven years ago" the function should return the number 6. Demonstrate the function in a program that asks the user to input a string and then passes it to the function. The number of words in the string should be displayed on the screen.

4. Average Number of Letters

Modify the program you wrote for problem 3 (Word Counter), so it also displays the average number of letters in each word.



5. Sentence Capitalizer

Write a function that accepts a C-string as an argument and capitalizes the first character of each sentence in the string. For instance, if the string argument is "`hello. my name is Joe. what is your name?`" the function should manipulate the string so it contains "`Hello. My name is Joe. What is your name?`" Demonstrate the function in a program that asks the user to input a string and then passes it to the function. The modified string should be displayed on the screen.

6. Vowels and Consonants

Write a function that accepts a C-string as its argument. The function should count the number of vowels appearing in the string and return that number.

Write another function that accepts a C-string as its argument. This function should count the number of consonants appearing in the string and return that number.

Demonstrate the two functions in a program that performs the following steps:

1. The user is asked to enter a string.
2. The program displays the following menu:
   A) Count the number of vowels in the string
   B) Count the number of consonants in the string
   C) Count both the vowels and consonants in the string
   D) Enter another string
   E) Exit the program
3. The program performs the operation selected by the user and repeats until the user selects E, to exit the program.

7. Name Arranger

Write a program that asks for the user's first, middle, and last names. The names should be stored in three different character arrays. The program should then store in a fourth array the name arranged in the following manner: the last name followed by a comma and a space, followed by the first name and a space, followed by the middle name. For example, if the user entered "`Carol Lynn Smith`", it should store "`Smith, Carol Lynn`" in the fourth array. Display the contents of the fourth array on the screen.



8. Sum of Digits in a String

Write a program that asks the user to enter a series of single-digit numbers with nothing separating them. Read the input as a C-string or a `string` object. The program should display the sum of all the single-digit numbers in the string. For example, if the user enters 2514, the program should display 12, which is the sum of 2, 5, 1, and 4. The program should also display the highest and lowest digits in the string.

9. Most Frequent Character

Write a function that accepts either a pointer to a C-string, or a `string` object, as its argument. The function should return the character that appears most frequently in the string. Demonstrate the function in a complete program.

10. `replaceSubstring` Function

Write a function named `replaceSubstring`. The function should accept three C-string or `string` object arguments. Let's call them *`string1`*, *`string2`*, and *`string3`*. It should search *`string1`* for all occurrences of *`string2`*. When it finds an occurrence of *`string2`*, it should replace it with *`string3`*. For example, suppose the three arguments have the following values:

```
string1:    "the dog jumped over the fence"
string2:    "the"
string3:    "that"
```

With these three arguments, the function would return a `string` object with the value "that dog jumped over that fence". Demonstrate the function in a complete program.

11. Case Manipulator

Write a program with three functions: `upper`, `lower`, and `reverse`. The `upper` function should accept a C-string as an argument. It should step through all the characters in the string, converting each to uppercase. The `lower` function, too, should accept a pointer to a C-string as an argument. It should step through all the characters in the string, converting each to lowercase. Like `upper` and `lower`, `reverse` should also accept a C-string. As it steps through the string, it should test each character to determine whether it is upper- or lowercase. If a character is uppercase, it should be converted to lowercase. If a character is lowercase, it should be converted to uppercase.

Test the functions by asking for a string in function `main`, then passing it to them in the following order: `reverse`, `lower`, and `upper`.



12. Password Verifier

Imagine you are developing a software package that requires users to enter their own passwords. Your software requires that user's passwords meet the following criteria:

- The password should be at least six characters long.
- The password should contain at least one uppercase and at least one lowercase letter.
- The password should have at least one digit.

Write a program that asks for a password and then verifies that it meets the stated criteria. If it doesn't, the program should display a message telling the user why.

13. Phone Number List

Write a program that has an array of at least 10 string objects that hold people's names and phone numbers. You may make up your own strings or use the following:

```
"Becky Warren, 678-1223"
"Joe Looney, 586-0097"
"Geri Palmer, 223-8787"
"Lynn Presnell, 887-1212"
"Holly Gaddis, 223-8878"
"Sam Wiggins, 486-0998"
"Bob Kain, 586-8712"
"Tim Haynes, 586-7676"
"Warren Gaddis, 223-9037"
"Jean James, 678-4939"
"Ron Palmer, 486-2783"
```

The program should ask the user to enter a name or partial name to search for in the array. Any entries in the array that match the string entered should be displayed. For example, if the user enters "Palmer" the program should display the following names from the list:

```
Geri Palmer, 223-8787
Ron Palmer, 486-2783
```

14. Check Writer

Write a program that displays a simulated paycheck. The program should ask the user to enter the date, the payee's name, and the amount of the check. It should then display a simulated check with the dollar amount spelled out, as shown here:

```
                                  Date: 12/24/2007


Pay to the Order of: John Phillips           $1920.85

One thousand nine hundred twenty and 85 cents
```

Be sure to format the numeric value of the check in fixed-point notation with two decimal places of precision. Be sure the decimal place always displays, even when the number is zero or has no fractional part. Use either C-strings or string class objects in this program.

Input Validation: Do not accept negative dollar amounts, or amounts over $10,000.

15. Digit Sums of Squares and Cubes

If you add up all the digits in 468, you get 4 + 6 + 8 = 18. The square and cube of 468 are 219024 and 102503232, respectively. Interestingly, if you add up the digits of the square or cube, you get 18 again. Are there other integers that share this property? Write a program that lists all positive integers k less than 1000 such that the three numbers k, k^2, and k^3 have digits that add up to the same number.

16. Dollar Amount Formatter

Modify Program 12-23 by adding a function

```
string dollarFormat(double amount)
```

that takes a dollar amount in numeric form and returns a string formatted in currency notation, with a $ sign and commas inserted at the appropriate locations. Test your function using suitable inputs.

17. User Name

Write a program that queries its environment, determines the user's login name, and then greets the user by name. For example, if the login name of the user is `gcm`, then the program prints

```
Hello, gcm
```

when it is executed.

This page intentionally left blank

CHAPTER 13

Advanced File and I/O Operations

TOPICS

13.1 Files
13.2 Output Formatting
13.3 Passing File Stream Objects to Functions
13.4 More Detailed Error Testing
13.5 Member Functions for Reading and Writing Files
13.6 Binary Files
13.7 Creating Records with Structures
13.8 Random-Access Files
13.9 Opening a File for Both Input and Output
13.10 Online Friendship Connections Case Study: *Object Serialization*
13.11 Tying It All Together: *File Merging and Color-Coded HTML*

13.1 Files

CONCEPT: A file is a collection of information, usually stored on a computer's disk. Information can be saved to files and later retrieved.

Files are crucial to the operation of many real-world programs. Examples of familiar types of software packages that use files extensively are:

- **Word Processors**: Word processing programs are used to write letters, memos, reports, and other documents. The documents are then saved in files so they can be edited and printed.
- **Database Management Systems**: DBMSs are used to create and maintain databases. Databases are files that contain large collections of information, such as payroll records, inventories, sales statistics, and customer records.
- **Spreadsheets**: Spreadsheet programs are used to work with numerical data. Numbers and mathematical formulas can be inserted into the rows and columns of the spreadsheet. The spreadsheet can then be saved to a file for use later.

- **Compilers**: Compilers translate the source code of a program, which is saved in a file, into an executable file. Throughout the previous chapters of this book you have created many C++ source files and compiled them to executable files.

In previous chapters, you learned how to open files, how to read files, and how to store data in files. In this chapter, we briefly review some of what you already know, then cover additional information on the use of files.

File-Naming Conventions

Each operating system has its own rules for naming files. Some systems allow long filenames such as

```
infoFrom1997
corpSalesReport
vehicleRegistrations
```

Other systems only allow shorter file names. MS-DOS, for example, allows filenames of no more than eight characters with an optional three-character extension.

Extensions are commonly used with filenames. The name and extension are separated by a period, called a "dot." While the filename identifies the file's purpose, the extension identifies the type of information contained in the file. For example, the .cpp extension identifies a C++ program. The filename `payroll.cpp` would identify a payroll program written in C++. Table 13-1 lists other example filenames and the type of information they contain.

Table 13-1 Filename Extensions

Filename and Extension	File Contents
`myprog.bas`	BASIC program
`menu.bat`	Microsoft Windows batch file
`install.doc`	Microsoft Word file
`crunch.exe`	Executable file
`bob.html`	HTML file
`model.java`	Java program or applet
`invent.obj`	Object file
`ansi.sys`	System device driver
`readme.txt`	Text file

Using Files

As you learned in Chapter 3, there are three steps to using a file: The file must be opened, the file is then read or written, and finally, the file should be closed when the program has finished using it. In C++, files are used through *file stream objects*. These are objects belonging to the `ifstream`, `ofstream`, and `fstream` classes. Access to these classes is gained by placing the include directive

```
#include <fstream>
```

in your program. Objects of type `ifstream` connect to input files, while objects of type `ofstream` connect to output files. Objects of the `fstream` class can be used for input, output, or both input and output at the same time. For each file that you want to use in your program, you should first determine whether you want to use it for input, output, or both, then define an appropriate file stream object to be used to connect to the file. For example, to read an input file `inFile`, write an output file `outFile`, and use a third file `ioFile` for either input or output, you would define three file stream objects as follows:

```
ifstream inFile;
ofstream outFile;
fstream ioFile;
```

Using Variable File Names

You have already learned how to specify names of files to be opened or created by coding the literal name of the file into your program. For example, to read a file called `demoFile.txt`, you would write

```
inFile.open("demoFile.txt");
```

While this is convenient for student programs, real-world applications cannot require a specific file name: They must read the name of the file the user wants to use. To do this, define an array of characters to hold the name of the file, read the name into the array, and then pass the array to the `open` function:

```
ifstream inFile;
char fileName[81];
cout << "Please enter the name of the file: ";
cin.getline(filename, 81);
inFile.open(fileName);
```

You should be aware that the `open` function of file stream objects expects the name of the file to be a C-string and will not accept C++ `string` objects. If you already have the name of the file in a string object, you need to use the `c_str` or `data` member functions to extract the C-string from the string object:

```
ifstream inFile;
string fileName;
cout << "Please enter the name of the file: ";
getline(cin, fileName);
inFile.open(fileName.data());
```

Using a string object for the filename has the drawback of requiring an extra step to extract the C-string, but it has the advantage of avoiding possible buffer overflow problems if the name of the file entered by the user is longer than the array can hold.

How the Program Locates Files

The name of a file being given to a program can be simple, consisting of just the name of the file, as in

```
myfile.dat
```

or it can be a full pathname like

```
C:\csc\programs\myfile.dat
```

which specifies the path through the directories on the system that the computer should follow to locate the file. Generally, the full pathname has to be specified unless the file is located in the current directory. The current directory is the directory the user is in when he or she executes the program from the command line. This will normally be the directory containing the file with the program's executable code.



NOTE: The notion of a current directory does not apply if you are executing your program from within your compiler's development environment. In this case, files that are not specified using full pathnames will be searched for in a *default* directory. The default directory is usually the one that contains the program project's source files. For further information, see the appendices for Borland C++ Builder, and Microsoft Visual C++.

Using the `fstream` Object

File stream objects require the programmer to specify whether the object will be used for input, output, or both. This is done through the use of *file open modes*. A file open mode is an option, or *flag*, that the programmer can set to determine how the file will be used. The `open` member function of the file stream objects is overloaded to take a file mode as an optional second parameter.

File open modes are predefined values that are members of the `ios` class. The `ios::in` mode is used to set the `fstream` object for input, and `ios::out` is used to set it for output. For example, we can open a file `input.dat` for input, and `output.dat` for output, using the `fstream` object and the appropriate file modes as follows:

```
fstream inFile, outFile;
inFile.open("input.dat", ios::in);
outFile.open("output.dat", ios::out);
```

It is possible to combine several file open mode flags when a file is being opened. The combination of flags is achieved through the *bitwise or* operator `|`. The bitwise or operator, which occupies the same key as the backslash on most keyboards, is discussed further in Appendix H.

Here is an example of combining file mode flags to open a file for both reading and writing:

```
fstream dataFile;
myFile.open("myfile.dat", ios::in | ios::out);
```

Reading and Writing Files

File stream objects have a lot of the same member functions as, and behave similarly to, the `iostream` objects `cin` and `cout`. In particular, the insertion operator `<<`, the extraction operator `>>`, many member functions such as `getline`, and even the library function `getline` for working with string objects work the same way with file stream objects as they do with the `iostream` objects. In fact, even the I/O manipulators work with the `fstream` objects. Program 13-1 shows the use of the insertion operator and of the `getline` function with an `fstream` object. It opens a file for writing, writes two

sentences to it, closes the file, and then opens the file for reading. The contents of the file are then printed on the screen.

Program 13-1

```
1  //This program demonstrates the use of an fstream object
2  //and file mode flags.
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  int main()
8  {
9     fstream dataFile;  // file object
10    string buffer;     // Used to read line from file
11
12    // Create a new file named myfile.dat to write to.
13    dataFile.open("myfile.dat", ios::out);
14
15    // Write two lines to the file.
16    dataFile << "Now is the time for all good men" << endl
17             << "to come to the aid of their country.";
18    // Close the file.
19    dataFile.close();
20
21    // Open the file for input.
22    dataFile.open("myfile.dat", ios::in);
23
24    // Read a line into a buffer and print the line.
25    getline(dataFile, buffer);
26    cout << buffer << endl;
27
28    // Read a second line and print it.
29    getline(dataFile, buffer);
30    cout << buffer << endl;
31
32    // Close the file.
33    dataFile.close();
34 }
```

Program Output

```
Now is the time for all good men
to come to the aid of their country.
```

There are other file modes besides `ios::in` and `ios::out`. Although we have illustrated file modes for `fstream` objects, file modes can be used with `ifstream` and `ofstream` objects as well. Of course, some file modes do not make sense with certain file stream objects: For example, you should not use the `ios::in` mode with an `ofstream` object. Certain file streams are opened with some of the modes already set by default. Table 13-2 gives details on default modes for `ifstream` and `ofstream` objects.

Table 13-2 Files Open Modes

File Type	Default Open Mode
`ofstream`	The file is opened for output only. (Information may be written to the file, but not read from the file.) If the file does not exist, it is created. If the file already exists, its contents are deleted (the file is erased).
`ifstream`	The file is opened for input only. (Information may be read from the file, but not written to it.) The file's contents will be read from its beginning. If the file does not exist, the open function fails.

Many times when a program opens a file for output, the assumption is that no file of that name already exists: The intention is to create a new file to hold data generated by the program. This is why the default action in opening a file for output is to delete any existing file of the same name and replace it with a new one. The `ios::app` mode can be used if the data already in the file is to be preserved and new data being generated is to be added, or *appended*, to the end of the file. The `ios::noreplace` flag specifies that an attempt to open a file for output should fail if the file already exists. This protects important files from deletion or modification.



NOTE: When used by itself, the `ios::out` causes the file's contents to be deleted if the file already exists. When it is used with the `ios::app` flag, however, the file's existing contents are preserved. If the file does not exist, it is created.

Program 13-2 illustrates the use of the `ios::app` file mode. It opens a file, writes two names to the file, and then closes the file. The program then reopens the file in append mode, adds two more names, closes the file, and terminates.

Program 13-2

```
1  // This program writes information to a file, closes the file,
2  // then reopens it and appends more information.
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8     fstream dataFile;  // file object
9
10    // Open a file to write to, and write to it.
11    dataFile.open("demofile.txt", ios::out);
12    dataFile << "Jones\n";
13    dataFile << "Smith\n";
14
15    // Close the file.
16    dataFile.close();
17
```

(program continues)

Program 13-2 *(continued)*

```
18    // Open the same file in append mode, and write to it.
19    dataFile.open("demofile.txt", ios::out|ios::app);
20    dataFile << "Willis\n";
21    dataFile << "Davis\n";
22
23    // Close the file.
24    dataFile.close();
25    return 0;
26 }
```

Output to File `demofile.txt`

```
Jones
Smith
Willis
Davis
```

Table 13-3 gives a list of file mode flags and their meanings.

Table 13-3 File Mode Flags

File Mode Flag	Meaning
`ios::app`	Append mode. If the file already exists, its contents are preserved and all output is written to the end of the file. By default, this flag causes the file to be created if it does not exist.
`ios::ate`	Initial output to the file will take place at the end of the file.
`ios::binary`	Binary mode. When a file is opened in binary mode, information is written to or read from it in pure binary format. (The default mode is text.)
`ios::in`	Input mode. Information will be read from the file. If the file does not exist, it will not be created and the open function will fail.
`ios::out`	Output mode. Information will be written to the file. By default, the file's contents will be deleted if it already exists.
`ios::trunc`	If the file already exists, its contents will be deleted (truncated). This is the default mode used by `ios::out`.

Using File Stream Constructors to Open Files

The `ifstream`, `ofstream`, and `fstream` classes all have constructors that can take a name of a file as well as an optional set of file open mode flags and automatically open the file. This eliminates the need for a separate call to the `open` member function. Here are some examples:

```
ifstream infile("names.dat");
fstream iofile("names.dat", ios::in | ios::out);
```

Testing for File Operation Errors

A call to open a file, such as

```
infile.open("names.dat", ios::in);
```

will fail if the `names.dat` file does not exist. It is good programming practice to check whether an attempt to open a file was successful before attempting to use the file. Testing for a successful file open operation is usually accomplished by testing the value of the stream object immediately after the call to open the file (or after the call to the constructor if the file is being opened in the constructor). The value of the stream object will be false (or zero) if the open failed. Thus the expression `!infile` will be true if the open call failed.

```
fstream infile("customer.dat", ios::in)
if (!infile)
 {
     cout << "The customer.dat file could not be opened";
     exit(1);
 }
```

Other file operations, such as attempts to read or write a file can fail for a variety of hardware-related reasons. For example, an attempt to write data will fail if the disk is full, or if the disk drive is malfunctioning. Likewise, an attempt to read a file will fail if there is no more data left to read in the file. File stream objects have a member function `fail`, which returns `true` if an I/O operation was not successfully completed. For example, a failed attempt to open a file can be detected by using code such as this:

```
infile.open("customer.dat");
if (infile.fail())
 {
     cout << "The customer.dat file could not be opened";
     exit(1);
 }
```

This code will detect a failed attempt to open a file for input and terminate the program.

Testing for the End of a File

Testing for the end of a file is accomplished by calling the `eof` member function, as you learned in Chapter 5. The `eof` function, however, is not the best way to determine whether there is still data left in the file when reading from the file using the extraction operator. Depending on whether the file contains whitespace after the last item, code using `eof` to check for the end of a file will either miss the last data item in the file or read the last data item twice. A better way to determine if there is data left in the file is to attempt a read, and then call the `fail` function to find out if the read was successful. For example, the following code fragment will read and print to the screen all numbers stored in a file:

```
string filename;
cout << "Enter the name of the file:";
getline(cin, filename);
ifstream file(filename.c_str());
if (!file)
 {
     cout << "Cannot open the file " << filename;
     exit(1);
 }
cout << "Contents of the file are:";
cout << endl;
```

```
int number;
file >> number;
while (!file.fail())
{
   cout << number << " ";
   file >> number;
}
```

There is another way of testing whether a read operation was successful. This method is based on the fact that the extraction operator >> returns a value that can be tested to determine the success of the operation. Thus the attempt to read and the subsequent test for success can be done by putting the read operation as the test for loop continuation in a `while` loop as follows:

```
cout << "Contents of the file are:";
cout << endl;
int number;
while (file >> number)
{
   cout << number << " ";
}
```

Notice that the initial read outside the loop and the second read at the end of the loop are both eliminated because they are folded into the loop test.

The `eof` function can reliably be used to test for end of data in binary files because there are no whitespace separators in that kind of file. Binary files are covered in a later section.



Checkpoint

13.1 How are files identified outside of a C++ program?

13.2 What are the three steps that must be taken when a file is used by a program?

13.3 What header file is required for a C++ program to perform file operations?

13.4 Describe each of the following file stream types:

```
ofstream
ifstream
fstream
```

13.5 What file operation must be performed before information can be written to or read from a file?

13.6 Assuming `diskInfo` is an `fstream` object, write a statement that opens the file `names.dat` for output.

13.7 Assuming `diskInfo` is an `fstream` object, write a statement that opens the file `customers.dat` for output, where all output will be written to the end of the file.

13.8 Assuming `diskInfo` is an `fstream` object, write a statement that opens the file `payable.dat` for both input and output.

13.9 Write an `if` statement that tests the file stream object `dataBase` to determine whether the file was opened successfully.

13.10 Write a statement that closes the file associated with the file stream object `outFile`.

13.2 Output Formatting

CONCEPT: Output formatting works the same way on all output stream objects.

Output formatting works the same way with file stream objects as with the `cout` object. In particular, the I/O manipulators you learned about in Chapter 3,

```
setw(n)                 fixed
showpoint               setprecision(n)
left                    right
```

all work the same way with output file streams as they do with `cout`. Program 13-3 demonstrates the use of `setw(n)` to arrange output being sent to a file into columns.

Program 13-3

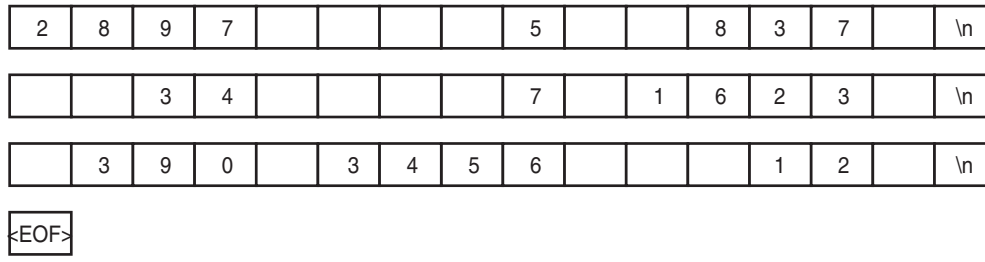
```
1  //This program formats a table of numbers into a file
2  //using setw(n).
3  #include <iostream>
4  #include <fstream>
5  #include <iomanip>
6  using namespace std;
7
8  int main()
9  {
10    const int ROWS = 3, COLS = 3;
11    fstream outFile("table.txt", ios::out);
12    int nums[ROWS][COLS] = { 2897,    5,  837,
13                               34,    7, 1623,
14                              390, 3456,   12 };
15
16    // Write the three rows of numbers.
17    for (int row = 0; row < ROWS; row++)
18    {
19       for (int col = 0; col < COLS; col++)
20       {
21          outFile << setw(4) << nums[row][col] << "  ";
22       }
23       outFile << endl;
24    }
25
26    // Close the file.
27    outFile.close();
28    return 0;
29 }
```

Output to File `table.txt`

```
2897     5   837
  34     7  1623
 390  3456    12
```

Figure 13-1 shows the way the characters appear in the file.

Figure 13-1



Formatting Using `sstream` Objects

In Chapter 12, you learned about the library class `ostringstream`. Objects of that class are capable of writing to strings stored in memory and can be used to format certain types of output. You need to include the appropriate header file in your program before you can use the `ostringstream` class:

```
#include <sstream>
```

Stream objects of `ostringstream` type are defined in the usual way:

```
ostringstream outStr;
```

Once a string stream object has been defined, we can write to it using the insertion operator << and the usual formatting manipulators. The C-string in the string stream can then be extracted using the member function `str`. For example, to print on the screen what has already been written into the string stream object, we can write

```
cout << outStr.str();
```

Program 13-4 demonstrates the use of a string output stream. It simply writes the squares of the first five positive integers to a string object, then extracts the string and prints it on the screen using `cout`.

Program 13-4

```
1  //This program demonstrates a string stream output object.
2  #include <iostream>
3  #include <sstream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     const int SIZE = 5;
10    double squares[SIZE] = {1, 4, 9, 16, 25};
11    ostringstream outStr;     //String stream object.
12
```

(program continues)

Program 13-4 *(continued)*

```
13    //Format everything in the in-memory string.
14    outStr << showpoint << fixed << setprecision(1);
15    for (int index = 0; index < SIZE; index++)
16       {
17          outStr << setw(10) << squares[index];
18       }
19    //Print the string on the screen.
20    cout << outStr.str();
21    return 0;
22 }
```

Program Output

```
      1.0       4.0       9.0      16.0      25.0
```



NOTE: On some compilers, you must write a null terminator to the end of a `ostringstream` object.

Consider the problem of formatting a set of prices for some products in columns, with the dollar amounts being right justified in fields of width 10. For example, if we had such a price table of three rows and two columns, it might look like this:

```
  $124.45     $5.00
   $67.67    $34.32
   $56.23    $89.00
```

We can format a price into a string object by first writing the dollar sign, followed immediately by the price:

```
ostringstream outStr;
outStr << showpoint << fixed << setprecision(2);
outStr << '$' << amount;
```

We can then extract the string `outStr.str()` and print it right justified in a field of 10 spaces. Program 13-5 implements this strategy to format a table of prices. The formatting procedure is embodied in a function that takes as parameters the dollar amount to be formatted and returns the formatted amount as a string object. The main function then writes this string right justified in a field of 10.

Program 13-5

```
1  // This program demonstrates the use of an ostringstream
2  // object to do sophisticated formatting.
3  #include <iostream>
4  #include <iomanip>
5  #include <sstream>
6  using namespace std;
7
8  string dollarFormat(double); // Function Prototype
9
10 int main()
11 {
12    const int ROWS = 3, COLS = 2;
13
```

(program continues)

Program 13-5 *(continued)*

```
14    double amount[ROWS][COLS] = {184.45,    7, 59.13,
15                                 64.32,   7.29,  1289  };
16
17    // Format table of dollar amounts in columns of width 10
18    for (int row = 0; row< ROWS; row++)
19       {
20          for (int column = 0; column < COLS; column++)
21           {
22              cout << setw(10)
23                   << dollarFormat(amount[row][column]);
24           }
25          cout << endl;
26       }
27    return 0;
28 }
29
30 //************************************************
31 // formats a dollar amount                       *
32 //************************************************
33 string dollarFormat(double amount)
34 {
35    // Create ostringstream object
36    ostringstream outStr;
37
38    // Set up format information and write to outStr.
39    outStr << showpoint << fixed << setprecision(2);
40    outStr << '$' << amount;
41
42    // Extract and return the string inside outStr.
43    return outStr.str();
44 }
```

Program Output

```
   $184.45     $7.00
    $59.13    $64.32
     $7.29  $1289.00
```

Table 13-4 shows a list of other I/O manipulators that can be used with C++ stream objects and gives a brief description of their meanings.

Table 13-4 I/O Manipulators

Manipulator	Description
`dec`	Displays subsequent numbers in decimal format.
`endl`	Writes new line and flushes output stream.
`fixed`	Uses fixed notation for floating-point numbers.
`flush`	Flushes output stream.
`hex`	Inputs or outputs in hexadecimal.
`left`	Left justifies output.

(table continues)

Table 13-4 I/O Manipulators *(continued)*

Manipulator	Description
`oct`	Inputs or outputs in octal.
`right`	Right justifies output.
`scientific`	Uses scientific notation for floating-point numbers.
`setfill(ch)`	Makes `ch` the fill character.
`setprecision(n)`	Sets floating-point precision to `n`.
`setw(n)`	Set width of output field to `n`.
`showpoint`	Forces decimal point and trailing zeros to be displayed.
`noshowpoint`	Prints no trailing zeros and drops decimal point if possible.
`showpos`	Prints a + with nonnegative numbers.
`noshowpos`	Prints no + with nonnegative numbers.

You have already encountered some of these manipulators in Chapter 3. The `oct`, `dec`, and `hex` manipulators can be used with both input and output streams; they allow numbers to be read or written using the octal, decimal, or hexadecimal number systems. For example, Program 13-6 will print the numbers 63 in octal, decimal, and hexadecimal, then decimal again.

Program 13-6

```
1  //This program demonstrates output of numbers using
2  //the octal, decimal, and hexadecimal number systems.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     int x = 63;
10
11    cout << setw(20) << left << "Octal:"  << right
12         << oct << x << endl;
13    cout << setw(20) << left << "Decimal:" << right
14         << dec << x << endl;
15    cout << setw(20) << left << "Hexadecimal:" << right
16         << hex << x << endl;
17    cout << setw(20) << left << "Decimal:" << right
18         << dec << x << endl;
19    return 0;
20 }
```

Program Output

```
Octal:              77
Decimal:            63
Hexadecimal:        3f
Decimal:            63
```

Recall from Chapter 3 that when a program writes data to an open file, the data does not go directly to the file. Instead, the data is stored in an *output buffer* associated with the file and is later transferred to the file in a process known as *flushing* the buffer. Usually the buffer is only flushed if it is full or when the file is closed. The `endl` and `flush` manipulators allow the programmer to flush the buffer at any time, hence forcing transfer of buffered data to the file. For example, the following statement flushes the buffer of an output stream:

```
dataOut << flush;
```

The `scientific` manipulator causes floating-point numbers to be written out in scientific notation, that is, in the form d.dddEdd. The *fill* character is the character that is written when a printed number does not fill the entire field it is printed in. By default, the fill character is a blank. The programmer can specify a different fill character by using the `setfill` manipulator. For example,

```
dataOut << setfill('%');
```

will make the percent character (`%`) the fill character.

Output Formatting with Member Functions

Stream manipulators are not the only way to format data. Field width, precision, and the various format *flags* may also be modified by using member functions of the stream objects. For example, the display field width of an output stream `dataOut` can be set with the member function

```
dataOut.width(n);
```

This is equivalent to using the stream manipulator

```
dataOut << setw(n);
```

Similarly, there is a member function `precision(n)` for setting the precision for floating-point numbers; the call

```
dataOut.precision(n);
```

is equivalent to the statement

```
dataOut << setprecision(n);
```

Note that the names of the member functions are different from the names of the corresponding manipulators. There is also a `setf` member function that can be used to do the job done by the manipulator flags. It is passed a flag that is a value defined in the `ios` class. For example, the call

```
dataOut.setf(ios::fixed);
```

is equivalent to the using the manipulator

```
dataOut << fixed.
```

Multiple format flags can be set with a single call to `setf` by combining them using the bitwise or operator. Here is a statement that sets the flags for fixed-point notation, decimal point displaying, and left-justification:

```
dataOut.setf(ios::fixed | ios::showpoint | ios::left);
```

Regardless of the way format flags are set, they can be turned off with the `unsetf` function. It works like `setf`, except the format flags you specify are disabled. The following statement turns off the fixed and left-justification flags:

```
dataOut.unsetf(ios::fixed | ios::left);
```

Table 13-5 shows the member functions that can be used for formatting.

Table 13-5 Formatting Member Functions

Member Functions	Description
`width()`	Sets the display field width.
`precision()`	Sets the precision of floating-point numbers.
`setf()`	Sets the specified format flags.
`unsetf()`	Disables, or turns off, the specified format flags.



Checkpoint

13.11 Write `cout` statements with member function calls that perform the following:

A) Set the field width to six spaces.
B) Set the precision to four decimal places.
C) Set the display mode to fixed-point notation.
D) Set the display mode to left justification and scientific notation.
E) Turn off the scientific notation display mode.

13.12 The following program will not compile because the lines have been mixed up.

```
#include <iomanip>
}
cout << person << endl;
string person = "Wolfgang Smith";
int main()
cout << person << endl;
{
#include <iostream>
cout.setf(ios::left);
cout.width(20);
cout.setf(ios::right);
using namespace std;
return 0;
```

When the lines are properly arranged the program should display the following:

```
       Wolfgang Smith
Wolfgang Smith
```

Rearrange the lines in the correct order. Test the program by entering it on the computer, compiling it, and running it.

13.3 Passing File Stream Objects to Functions

CONCEPT: File stream objects should be passed by reference to functions.



VideoNote
Passing File Stream Objects to Functions

A file object maintains a state within it that changes with every file operation. Part of this state tracks the current file position, that is, the place in the file where the next read or write will take place.

Suppose that you have a file and two functions that need to take turns reading from the file. For example, the file might consist of runs of text followed by runs of numbers. Your program might have a main function that opens the file and processes it by calling a function *F* whenever it comes to a run of text, and by calling another function *G* whenever it comes to a run of numbers.

How should `main` pass the file object to the two functions? If the file object is passed by value, the called function will get a copy of the file object. When function *F* is called, it will read the file until it comes to the end of the run of text, but this new file position will be reflected only in the copy of the file object that was passed to *F*. When *F* returns, its copy of the file object, being a local variable, is destroyed. The main function will still have its copy of the file object, but this copy will not reflect the read operations performed by *F*. In particular, the file position in `main`'s file object will still be at the beginning of the run of text. This will lead to an error when the file object is passed to *G*. For this reason, file objects should be passed by reference.

Here is an example of a function with a file object parameter passed by reference. The function takes a file object as parameter and echoes the contents of the file to the screen.

```
void showContents(fstream &file)
{
   string str;

   file >> str;
   while (!file.fail())
   {
      cout << str << endl;
      file >> str;
   }
}
```

A complete example is given in Program 13-7.

Program 13-7

```
1 // This program uses the file stream object's fail()
2 // member function to detect the end of the data in
3 // a file. The program also demonstrates how to pass
4 // file stream objects as parameters.
5 #include <iostream>
6 #include <fstream>
7 #include <string>
8 using namespace std;
9
```

(program continues)

Program 13-7 *(continued)*

```
10 // Function prototype
11 void showContents(fstream &);
12
13 int main()
14 {
15    fstream dataFile;
16
17    dataFile.open("demofile.txt", ios::in);
18    if (dataFile.fail())
19    {
20       cout << "File open error!" << endl;
21       exit(1);
22    }
23    cout << "File opened successfully.\n";
24    cout << "Now reading information from the file.\n\n";
25    showContents(dataFile);
26    dataFile.close();
27    cout << "\nDone.\n";
28    return 0;
29 }
30
31 //*********************************************************
32 // Definition of function showContents. Accepts an fstream  *
33 // reference as its argument. Reads strings from the file   *
34 // displays them on the screen.                             *
35 //*********************************************************
36
37 void showContents(fstream &file)
38 {
39    string str;
40
41    file >> str;
42    while (!file.fail())
43    {
44       cout << str << endl;
45       file >> str;
46    }
47 }
```

Program Screen Output

```
File opened successfully.
Now reading information from the file.

Jones
Smith
Willis
Davis

Done.
```

13.4 More Detailed Error Testing

CONCEPT: All stream objects have error state bits that indicate the condition of the stream.

All stream objects contain a set of bits that act as flags. These flags indicate the current state of the stream. Table 13-6 lists these bits.

Table 13-6 Files Condition Bit Flags

Bit	Description
`ios::eofbit`	Set when the end of an input stream is encountered.
`ios::failbit`	Set when an attempted operation has failed.
`ios::hardfail`	Set when an unrecoverable error has occurred.
`ios::badbit`	Set when an invalid operation has been attempted.
`ios::goodbit`	Set when all the flags above are not set. Indicates the stream is in good condition.

These bits can be tested by the member functions listed in Table 13-7. (You've already learned about the `eof()` and `fail()` functions.) One of the functions listed in the table, `clear()`, can be used to set a status bit.

Table 13-7 Member Functions That Report on the Bit Flags

Function	Description
`eof()`	Returns true (nonzero) if the `eofbit` flag is set; otherwise returns false.
`fail()`	Returns true (nonzero) if the `failbit` or `hardfail` flags are set; otherwise returns false.
`bad()`	Returns true (nonzero) if the `badbit` flag is set; otherwise returns false.
`good()`	Returns true (nonzero) if the `goodbit` flag is set; otherwise returns false.
`clear()`	When called with no arguments, clears all the flags listed above. Can also be called with a specific flag as an argument.

The function `showState`, shown here, accepts a file stream reference as its argument. It shows the state of the file by displaying the return values of the `eof()`, `fail()`, `bad()`, and `good()` member functions:

```
void showState(fstream &file)
{
   cout << "File Status:\n";
   cout << "  eof bit: " << file.eof() << endl;
   cout << "  fail bit: " << file.fail() << endl;
   cout << "  bad bit: " << file.bad() << endl;
   cout << "  good bit: " << file.good() << endl;
   file.clear();         // Clear any bad bits
}
```

Program 13-8 uses the `showState` function to display `testFile`'s status after various operations. First, the file is created and the integer value 10 is stored in it. The file is then closed and reopened for input. The integer is read from the file, and then a second read operation is performed. Since there is only one item in the file, the second read operation will result in an error.

Program 13-8

```
1  // This program demonstrates the return value of
2  // the stream object error testing member functions.
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  // Function prototype
8  void showState(fstream &);
9
10 int main()
11 {
12    // Open a file, write a number, and show file status.
13    fstream testFile("stuff.dat", ios::out);
14    if (testFile.fail())
15    {
16       cout << "cannot open the file.\n";
17       return 0;
18    }
19    int num = 10;
20    cout << "Writing to the file.\n";
21    testFile << num;
22    showState(testFile);
23    testFile.close();
24
25    // Open the same file, read the number, show status.
26    testFile.open("stuff.dat", ios::in);
27    if (testFile.fail())
28    {
29       cout << "cannot open the file.\n";
30       return 0 ;
31    }
32    cout << "Reading from the file.\n";
33    testFile >> num;
34    showState(testFile);
35
36    // Attempt an invalid read, and show status.
37    cout << "Forcing a bad read operation.\n";
38    testFile >> num;
39    showState(testFile);
40
41    // Close file and quit.
42    testFile.close();
43    return 0;
44 }
45
```

(program continues)

Program 13-8 *(continued)*

```
46 //*********************************************************
47 // Definition of function showState. This function uses   *
48 // an fstream reference as its parameter. The return      *
49 // values of the eof(), fail(), bad(), and good() member  *
50 // functions is displayed. The clear() function is called *
51 // before the function returns.                           *
52 //*********************************************************
53 void showState(fstream &file)
54 {
55    cout << "File Status:\n";
56    cout << "  eof bit: " << file.eof() << endl;
57    cout << "  fail bit: " << file.fail() << endl;
58    cout << "  bad bit: " << file.bad() << endl;
59    cout << "  good bit: " << file.good() << endl;
60    file.clear();  // Clear any bad bits.
61 }
```

Program Screen Output

```
Writing to the file.
File Status:
  eof bit: 0
  fail bit: 0
  bad bit: 0
  good bit: 1
Reading from the file.
File Status:
  eof bit: 1
  fail bit: 0
  bad bit: 0
  good bit: 0
Forcing a bad read operation.
File Status:
  eof bit: 1
  fail bit: 1
  bad bit: 0
  good bit: 0
```

13.5 Member Functions for Reading and Writing Files

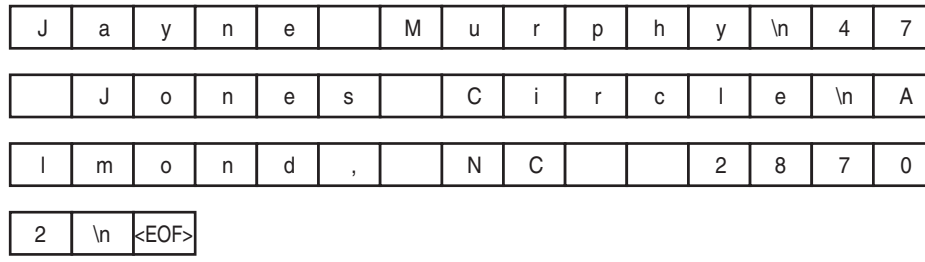
CONCEPT: **File stream objects have member functions for more specialized file reading and writing.**

If whitespace characters are part of the information in a file, a problem arises when the file is read by the >> operator. Since the operator considers whitespace characters as delimiters, it does not read them. For example, consider the file `murphy.txt` that contains the following information:

Jayne Murphy
47 Jones Circle
Almond, NC 28702

Figure 13-2 shows the way the information is recorded in the file.

Figure 13-2



The problem that arises from the use of the >> operator is evident in the output of Program 13-9.

Program 13-9

```
1  // This program shows the behavior of the >> operator
2  // on files that contain spaces as part of the information.
3  // The program reads the contents of the file and transfers
4  // those contents to standard output.
5  #include <iostream>
6  #include <fstream>
7  using namespace std;
8
9  int main()
10 {
11    // Variables needed to read file.
12    const int LENGTH = 81;
13    fstream file;
14    char input[LENGTH];
15
16    // Open the file.
17    file.open("murphy.txt", ios::in);
18    if (!file)
19    {
20       cout << "File open error!" << endl;
21       return 0;
22    }
23
24    // Read the file and echo to screen.
25    file >> input;
26    while (!file.fail())
27    {
28       cout << input;
29       file >> input;
30    }
31
```

(program continues)

Program 13-9 *(continued)*

```
32    // Close the file.
33    file.close();
34    return 0;
35 }
```

Program Screen Output

```
JayneMurphy47JonesCircleAlmond,NC28702
```

The `getline` Member Function

The problem with Program 13-9 can be solved by using the file stream object's `getline` member function. The function reads a "line" of information, including whitespace characters. Here is an example of the function call:

```
dataFile.getline(str, 81, '\n');
```

The three arguments in this statement are as follows.

`str` This is the name of a character array, or a pointer to a section of memory. The information read from the file will be stored here.

`81` This number is one greater than the maximum number of characters to be read. In this example, a maximum of 80 characters will be read.

`'\n'` This is a delimiter character of your choice. If this delimiter is encountered, it will cause the function to stop reading before it has read the maximum number of characters. (This argument is optional. If it's left out, `'\n'` is the default.)

The statement is an instruction to read a line of characters from the file. The function will read until it has read 80 characters or encounters a `\n`, whichever happens first. The line of characters will be stored in the `str` array.

Program 13-10 is a modification of Program 13-9. It uses the `getline` member function to read whole lines of information from the file.

Program 13-10

```
1  // This program uses the file stream object's getline() member
2  // function to read a line of information from the file.
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  int main()
8  {
9     // Variables needed for file input.
10    const int LENGTH = 81;
11    fstream nameFile;
12    char input[LENGTH];
13
```

(program continues)

Program 13-10 *(continued)*

```
14    // Open the file.
15    nameFile.open("murphy.txt", ios::in);
16    if (!nameFile)
17    {
18       cout << "File open error!" << endl;
19       return 0;
20    }
21
22    // Read the file one line at at time.
23    nameFile.getline(input,LENGTH);
24    while (!nameFile.fail())
25    {
26       cout << input << endl;
27       nameFile.getline(input,LENGTH);
28    }
29
30    // Close the file.
31    nameFile.close();
32    return 0;
33 }
```

Program Screen Output

```
Jayne Murphy
47 Jones Circle
Almond, NC 28702
```

Since the third argument of the `getline` function was left out in Program 13-10, its default value is `\n`. Sometimes you might want to specify another delimiter. For example, consider a file that contains multiple names and addresses, and that is internally formatted in the following manner:

Contents of `addresses.txt`

```
Jayne Murphy$47 Jones Circle$Almond, NC 28702\n$Bobbie Smith$
217 Halifax Drive$Canton, NC 28716\n$Bill Hammet$PO Box 121$
Springfield, NC 28357\n$
```

Think of this file as consisting of three records. A record is a complete set of information about a single item. Also, the records in the file are made of three fields. The first field is the person's name. The second field is the person's street address or PO box number. The third field contains the person's city, state, and ZIP code. Notice that each field ends with a `$` character, and each record ends with a `\n` character. Program 13-11 demonstrates how a `getline` function can be used to detect the `$` characters.

Program 13-11

```
1 // This file demonstrates the getline function with a
2 // user-specified delimiter.
3 #include <iostream>
4 #include <fstream>
5 using namespace std;
6
```

(program continues)

Program 13-11 *(continued)*

```
 7 int main()
 8 {
 9    // Variables needed to read file.
10    const int LENGTH = 81;
11    char input[LENGTH];
12
13    // Open the file.
14    fstream dataFile("addresses.txt", ios::in);
15    if (!dataFile)
16    {
17       cout << "Error opening file.";
18       return 0;
19    }
20
21    // Read lines terminated by '$' sign and output.
22    dataFile.getline(input, LENGTH, '$');
23    while (!dataFile.fail())
24    {
25       cout << input << endl;
26       dataFile.getline(input, LENGTH, '$');
27    }
28
29    // Close the file.
30    dataFile.close();
31    return 0;
32 }
```

Program Output

```
Jayne Murphy
47 Jones Circle
Almond, NC 28702

Bobbie Smith
217 Halifax Drive
Canton, NC 28716

Bill Hammet
PO Box 121
Springfield, NC 28357
```

Notice that the `\n` characters, which mark the end of each record, are also part of the output. They cause an extra blank line to be printed on the screen, separating the records.



NOTE: When using a printable character, such as `$`, to delimit information in a file, be sure to select a character that will not actually appear in the information itself. Since it's doubtful that anyone's name or address contains a `$` character, it's an acceptable delimiter. If the file contained dollar amounts, however, another delimiter would have been chosen.

As demonstrated in Program 13-1, another way to read lines of input is to use a `string` class object instead of an array of `char`, and then call the `getline` library function in place

of the `getline` member function discussed in this section. An advantage of the member function discussed in this section, however, is that it allows the use of a delimiter other than the end of line.

The `get` Member Function

Another useful member function is `get`. It reads a single character from the file. Here is an example of its usage:

```
inFile.get(ch);
```

In this example, `ch` is a `char` variable. A character will be read from the file and stored in `ch`. Program 13-12 shows the function used in a complete program. The user is asked for the name of a file. The file is opened and the `get` function is used in a loop to read the file's contents, one character at a time.

Program 13-12

```
1  // This program asks the user for a file name. The file is
2  // opened and its contents are displayed on the screen.
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  int main()
8  {
9     // Variables needed to read file one character at a time.
10    const int LENGTH = 81;
11    fstream file;
12    char ch, fileName[LENGTH];
13
14    // Get file name and open file.
15    cout << "Enter a file name: ";
16    cin >> fileName;
17    file.open(fileName, ios::in);
18    if (!file)
19    {
20       cout << fileName << " could not be opened.\n";
21       return 0;
22    }
23
24    // Read file one character at a time and echo to screen.
25    file.get(ch);
26    while (!file.fail())
27    {
28       cout << ch;
29       file.get(ch);
30    }
31
32    // Close file
33    file.close();
34    return 0;
35 }
```

Program 13-12 will display the contents of any file. The `get` function even reads whitespaces, so all the characters will be shown exactly as they appear in the file.

There is another version of the `get` function that takes no parameters and returns the character read, which is used as follows:

```
char ch;
ch = inFile.get();
```

This version of `get` returns a special `EOF` value when it gets to the end of the input stream. The following code will open a file and copy its contents to the screen by reading the file one character at a time:

```
char ch;
infile.open(fileName, ios::in);
ch = inFile.get();
while (ch != EOF)
{
   cout << ch;
   ch = inFile.get();
}
```

The peek Member Function

The `peek` member function is similar to `get`, but there is an important difference. When the `get` function is called, it returns the next character available from the input stream and simultaneously removes that character from the stream. In contrast, the `peek` function returns a copy of the next character available but leaves the character itself in the input stream. Thus `get()`reads a character from the file, but `peek()` just "looks" at the next character without actually reading it. To see the difference, suppose that a newly opened file contains the string `"abc"`. Then the sequence of statements

```
char ch = inFile.get();  // Read a character
cout << ch;              // Output the character
ch = inFile.get();       // Read another character
cout << ch;              // Output the character
```

will print the two characters `"ab"` on the screen. However, the statements

```
char ch = inFile.peek(); // Return the next character without reading it
cout << ch;              // Output the character
ch = inFile.get();       // Now read the next character
cout << ch;              // Output the character
```

will print the two characters `"aa"` on the screen.

The `peek` function is useful when you need to know what kind of data you are about to read before you actually read it, so you can decide on the best input method to use. If the data is numeric, it is best read with the stream extraction operator >>, but if the data is a non-numeric sequence of characters, then it should be read with `get` or `getline`. The following program uses the `peek` function in making a modified copy of a file by increasing the value of each integer number appearing in the file by 1.

Program 13-13

```
// This program demonstrates the peek member function.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    // Variables needed to read characters and numbers from
    // an input file
    const int LENGTH = 81;
    char fileName[LENGTH];
    fstream inFile;
    char ch;
    int number;

    fstream outFile;          // File object for output file

    // Open the file to be modified
    cout << "Enter a file name: ";
    cin >> fileName;
    inFile.open(fileName, ios::in);
    if (!inFile)
    {
       cout << "Cannot open file " <<  fileName;
       exit(1);
    }

    // Open the file to receive the modified copy
    outFile.open("modified.txt", ios::out);
    if (!outFile)
    {
        cout << "Cannot open the output file.";
        exit(2);
    }

    // Copy the input file one character at a time
    // except numbers in the input file must have 1
    // added to them

    // Peek at the first character
    ch = inFile.peek();
    while (ch != EOF)
    {
        // Examine current character
        if (isdigit(ch))
        {
            // numbers should be read with >>
            inFile >> number;
            outFile << number + 1;
        }
```

(program continues)

Program 13-13 *(continued)*

```
51          else
52          {
53              // just a simple character, read it and copy it.
54              ch = inFile.get();
55              outFile << ch;
56          }
57          // Peek at the next character from input file
58
59          ch = inFile.peek();
60      }
61
62      // Close the files.
63      inFile.close();
64      outFile.close();
65      return 0;
66  }
```

Sample Input File

```
Amy is 23 years old. Robert is 50 years old. The
difference between their ages is 27 years. Amy was born
in 1986.
```

Program Ouput for the Given Sample Input File

```
Amy is 24 years old. Robert is 51 years old. The
difference between their ages is 28 years. Amy was born
in 1987.
```

The program cannot tell beforehand whether the next character to be read is a digit that starts a number (in which case, the entire number should be read using the stream extraction operator >>) or just an ordinary nondigit character (in which case, the character should be read using a call to the `get()` member function). The program therefore uses `peek()`to examine characters without actually reading them (lines 40 and 57). If a character is a digit, the extraction operator is called to read the number that starts with that character (lines 44–49). Otherwise, the character is read using a call to `get()`(lines 50–55) and copied to the target file.

The `put` Member Function

The `put` member function writes a single character to the file. Here is an example of its usage:

```
outFile.put(ch);
```

In this statement, the variable `ch` is assumed to be a `char` variable. Its contents will be written to the file associated with the file stream object `outFile`. Program 13-14 demonstrates the `put` function.

Program 13-14

```
1  // This program demonstrates the put() member function.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8     // Variable needed to open and write file.
9     char ch;
10
11    // Open the file.
12    fstream dataFile("sentence.txt", ios::out);
13    if (!dataFile)
14    {
15       cout << "Error opening the file.";
16       return 0;
17    }
18
19    // Instructions to user.
20    cout << "Type a sentence and be sure to end it with a ";
21    cout << "period.\n";
22
23    // Read characters from cin and put to file.
24    cin.get(ch);
25    while (ch != '.')
26    {
27       dataFile.put(ch);
28       cin.get(ch);
29    }
30    dataFile.put(ch);
31
32    // Close the file.
33    dataFile.close();
34    return 0;
35 }
```

Program Screen Output with Example Input Shown in Bold

```
Type a sentence and be sure to end it with a
period.
```
I am on my way to becoming a great programmer.[Enter]

Resulting Contents of the File `sentence.txt`:

```
I am on my way to becoming a great programmer.
```



VideoNote
Rewinding a File

Rewinding a File

Many times it is useful to open a file, process all the data in it, rewind the file back to the beginning, and process it again, perhaps in a slightly different fashion. For example, a user may ask the program to search a database for all records of a certain kind, and when those are found, the user may want to search the database for all records of some other kind.

File stream classes offer a number of different member functions that can be used to move around in a file. One such method is the

```
seekg(offset, place);
```

member function of the input stream classes (the file "seeks" to a certain place in the file; the 'g' is for "get" and denotes that the function works on an input stream, because we "get" data from an input stream). The new location in the file to seek to is given by the two parameters: the new location is at an offset of `offset` bytes from the starting point given by `place`. The offset parameter is a `long` integer, while `place` can be one of three values defined in the `ios` class. The starting place may be the beginning of the file, the current place in the file, or the end of the file. These places are indicated by the constants `ios:beg`, `ios::cur`, and `ios::end`, respectively.

More information on moving around in files will be given in a later section. Here we are interested in moving to the beginning of the file. To move to the beginning of a file, use the call

```
seekg(0L, ios::beg);
```

to move 0 bytes relative to the beginning of the file.



NOTE: If you are already at the end of the file, you must clear the end of file flag *before* calling this function. Thus, to move to the beginning of a file stream `dataIn` that you have just read to the end, you need the two statements

```
dataIn.clear();
dataIn.seekg(0L, ios::beg);
```

Program 13-15 illustrates how to rewind a file. It creates a file, writes some text to it, and closes the file. The file is then opened for input, read once to the end, rewound, and then read again.

Program 13-15

```
1  // Program shows how to rewind a file. It writes a
2  // text file and opens it for reading, then rewinds
3  // it to the beginning and reads it again.
4  #include <iostream>
5  #include <fstream>
6  using namespace std;
7
8  int main()
9  {
10    // Variables needed to read or write file one
11    // character at a time.
12    char ch;
13    fstream ioFile("rewind.txt", ios::out);
14
15    // Open file.
16    if (!ioFile)
```

(program continues)

Program 13-15 *(continued)*

```
17    {
18       cout << "Error in trying to create file";
19       return 0;
20    }
21
22    // Write to file and close.
23    ioFile << "All good dogs " << endl
24           << "growl, bark, and eat." << endl;
25    ioFile.close();
26
27    //Open the file.
28    ioFile.open("rewind.txt", ios::in);
29    if (!ioFile)
30    {
31       cout << "Error in trying to open file";
32       return 0;
33    }
34
35    // Read the file and echo to screen.
36    ioFile.get(ch);
37    while (!ioFile.fail())
38    {
39       cout.put(ch);
40       ioFile.get(ch);
41    }
42
43    //Rewind the file.
44    ioFile.clear();
45    ioFile.seekg(0, ios::beg);
46
47    //Read file again and echo to screen.
48    ioFile.get(ch);
49    while (!ioFile.fail())
50    {
51       cout.put(ch);
52       ioFile.get(ch);
53    }
54    return 0;
55 }
```

Program Output

```
All good dogs
growl, bark, and eat.
All good dogs
growl, bark, and eat.
```



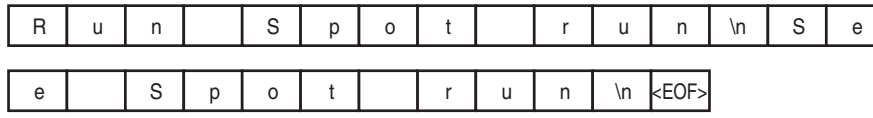
Checkpoint

13.13 Make the required changes to the following program so it writes its output to the file `output.txt` instead of to the screen.

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Today is the first day\n";
   cout << "of the rest of your life.\n";
   return 0;
}
```

13.14 Describe the purpose of the `eof` member function.

13.15 Assume the file `input.txt` contains the following characters:



What will the following program display on the screen?

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
   const int SIZE = 81;

   fstream inFile("input.txt", ios::in);
   char item[SIZE];
   inFile >> item;
   while (!inFile.fail())
   {
      cout << item << endl;
      inFile >> item;
   }
   return 0;
}
```

13.16 Describe the difference between reading a file with the >> operator and with the `getline` member function.

13.17 Describe the difference between the `getline` and `get` member functions.

13.18 Describe the purpose of the `put` member function.

13.19 What will be stored in the file `out.dat` after the following program runs?

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;

int main()
{
   const int SIZE = 5;
```

```
    ofstream outFile("out.dat");
    double nums[ ] = {100.279, 1.719, 8.602, 7.777, 5.099};
    outFile << setprecision(2);
    for (int count = 0; count < SIZE; count++)
    {
       outFile << setw(8) << nums[count];
    }
    outFile.close();
    return 0;
 }
```

13.20 The following program skeleton, when complete, will allow the user to store names and telephone numbers in a file. Complete the program.

```
#include <iostream>
#include <fstream>
#include <cctype>  // Needed for toupper
using namespace std;

int main()
{
   // Define a file stream object here and use
   // the file stream to open the file phones.dat.
   const int NAME_LENGTH = 81;
   const int PHONE_LENGTH = 26;
   char name[NAME_LENGTH], phone[PHONE_LENGTH];

   cout << "This program allows you to add names and phone\n";
   cout << "numbers to phones.dat.\n";
   do
   {
      char add;
      cout << "Do you wish to add an entry? ";
      cin >> add;
      if (toupper(add) == 'Y')
      {
         // Write code here that asks the user for a name
         // and phone number, then stores it in the file.
      }
   } while (toupper(add) == 'Y');
   // Don't forget to close the file.
   return 0;
}
```

13.6 Binary Files

CONCEPT: **Values of numeric data types such as `int` and `double` must be formatted for output before being written to text files. No such formatting takes place when numbers are written to binary files.**

A short integer number such as 1297 has both a string representation "1297" (shown in Figure 13-3) and a binary numeric representation (shown in Figure 13-4). Both representations can

be viewed as sequences of bytes. The string representation depends on the type of encoding used to represent individual characters and is 4 bytes long when the ASCII encoding is used. The number of bytes in the binary numeric representation depends on the type of the number and is 2 bytes long when the number is a `short int`. The conversion of string representation to numeric is called *parsing*, while the reverse conversion from numeric to string is called *formatting*.

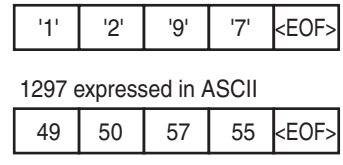
Although people find it natural to work with numbers in their string representation, computer hardware is better adapted to processing numbers in their binary form. This is why numbers must be parsed when input from the keyboard or from a file that has been edited by a person. It is also the reason why numbers must be formatted when being output in a form that will be viewed by humans. There are times, however, when a program is outputting data to a file that will only be read by other programs and will never be viewed by humans. In those cases, formatting of numeric data during output and the parsing of numbers during input can be omitted. When data is written in unformatted form, it is said to be written in *binary,* and files written in this way are called *binary files*. In contrast, files that hold formatted data are called *text files*.

As a convenience to programmers, the stream insertion operator << provides automatic formatting of numbers during output. Likewise, the stream extraction operator >> provides parsing of numeric input. For example, consider the following program fragment:

```
ofstream file("num.dat");
short x = 1297;
file << x;
```

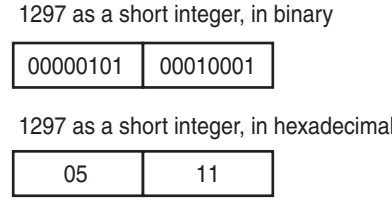
The last statement writes the contents of `x` to the file. When the number is written, however, it is stored as the characters `'1'`, `'2'`, `'9'`, and `'7'`. This is illustrated in Figure 13-3.

Figure 13-3



The number 1297 isn't stored in memory (in the variable `x`) in the fashion depicted in Figure 13-3, however. It is formatted as a binary number, occupying 2 bytes on a typical PC. Figure 13-4 shows how the number is represented in memory, using binary or hexadecimal.

Figure 13-4



The unformatted representation of the number shown in Figure 13-4 is the way the "raw" data is stored in memory. Information can be stored in a file in its pure, binary format. The first step is to open the file in binary mode. This is accomplished by using the `ios::binary` flag. Here is an example:

```
file.open("stuff.dat", ios::out | ios::binary);
```

Notice the `ios::out` and `ios::binary` flags are joined in the statement with the | operator. This causes the file to be opened in both output and binary modes.



NOTE: By default, files are opened in text mode.

The `write` member function of the `ostream` and `ofstream` classes can be used to write binary data to a file or other output stream. To call this function, you specify the address of a buffer containing an array of bytes to be written, and an integer indicating how many bytes are to be written:

```
write(addressOfBuffer, numberOfBytes);
```

The `write` member function does not distinguish between integers, floats, or some other type in the buffer; it just treats the buffer as an array of bytes. Because C++ does not support a pointer to a byte, the prototype of `write` specifies that the address of a buffer be a pointer to a `char`:

```
write(char *addressOfBuffer, int numberOfBytes);
```

This means that when we call `write`, we need to tell the compiler to interpret the address of the buffer as a pointer to `char`. We do this by using a special form of type casting called a `reinterpret_cast`. Briefly, `reinterpret_cast` is used to force the compiler to interpret the bits of one type as if they defined a value of a different type. Here is an example of using `reinterpet_cast` to convert a pointer to a `double` into a pointer to a `char`.

```
double d = 45.9;
double *pd = &d;
char *pChar;
// convert pointer to double to pointer to char
pChar = reinterpret_cast<char *>(pd);
```

In general, to convert a value to some target type, use the expression

```
reinterpret_cast<TargetType>(value);
```

Here are examples of using `write` to write a `double` and an array of `double` to a file.

```
double dl = 45.9;
double dArray[3] = { 12.3, 45.8, 19.0 };
ofstream outFile("stuff.dat", ios::binary);
outFile.write(reinterpret_cast<char *>(&dl), sizeof(dl));
outFile.write(reinterpret_cast<char *>(dArray),
                                       sizeOf(dArray));
```

Notice that in writing a single variable such as `dl`, we treat the variable itself as the buffer and pass its address (in this case the address is `&dl`). However, in using an array as the buffer, we just pass the array because the array is already an address.

If the data we are writing happens to be character data, there is no need to use the cast. Here are some examples of writing character data.

```
char ch = 'X';
char charArray[5] = "Hello";
outFile.write(&ch, sizeof(ch));
outFile.write(charArray, sizeof(charArray));
```

There is a `read` member function in the `istream` and `ifstream` classes that can be used to read binary data written by `write`. It takes as parameters the address of a buffer in which the bytes read are to be stored, and the number of bytes to read:

```
read(addressOfBuffer, numberOfBytes)
```

The address of the buffer must be interpreted as a pointer to `char` using `reinterpret_cast`. You can find out if the specified number of bytes was successfully read by calling the `fail()` member function on the input stream.

Program 13-16 demonstrates the use of `write` and `read`. The program initializes an array of integers and then stores the number of array entries in the array using the statements

```
int buffer[ ] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int size = sizeof(buffer)/sizeof(buffer[0]);
```

Recall that the `sizeof` operator can be used on variables to determine the number of bytes occupied by the variable. Here `sizeof(buffer)` returns the number of bytes allocated to the array by the initialization statement, and `sizeof(buffer[0])` returns the number of bytes occupied by a single array entry. By dividing the former by the latter, we obtain the number of array entries, which we then store in `size`.

Program 13-16 demonstrates the use of `write` and `read`.

Program 13-16

```
1  //This program uses the write and read functions.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8     // File object used to access file.
9     fstream file("nums.dat", ios::out | ios::binary);
10    if (!file)
11    {
12       cout << "Error opening file.";
13       return 0;
14    }
15
16    // Integer data to write to binary file.
17    int buffer[ ] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
18    int size = sizeof(buffer)/sizeof(buffer[0]);
19
```

(program continues)

Program 13-16 *(continued)*

```
20    // Write the data and close the file.
21    cout << "Now writing the data to the file.\n";
22    file.write(reinterpret_cast<char *>(buffer),
23               sizeof(buffer));
24    file.close();
25
26    // Open the file and use a binary read to read
27    // contents of the file into an array.
28    file.open("nums.dat", ios::in);
29    if (!file)
30    {
31       cout << "Error opening file.";
32       return 0;
33    }
34
35    cout << "Now reading the data back into memory.\n";
36    file.read(reinterpret_cast<char *>(buffer),
37              sizeof(buffer));
38
39    // Write out the array entries.
40    for (int count = 0; count < size ; count++)
41       cout << buffer[count] << " ";
42
43    // Close the file.
44    file.close();
45    return 0;
46 }
```

Program Screen Output

```
Now writing the data to the file.
Now reading the data back into memory.
1 2 3 4 5 6 7 8 9 10
```

13.7 Creating Records with Structures

CONCEPT: **Structures may be used to store fixed-length records to a file.**

Earlier in this chapter the concept of fields and records was introduced. A field is an individual piece of information pertaining to a single item. A record is made up of fields and is a complete set of information about a single item. For example, a set of fields might be a person's name, age, address, and phone number. Together, all those fields that pertain to one person make up a record.

In C++, structures provide a convenient way to organize information into fields and records. For example, the following structure declaration could be used to create a record containing information about a person.

```
const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
struct Info
{
   char name[NAME_SIZE];
   int age;
   char address1[ADDR_SIZE];
   char address2[ADDR_SIZE];
   char phone[PHONE_SIZE];
};
```

Besides providing an organizational structure for information, structures also package information into a single unit. For example, assume the structure variable `person` is declared as

```
Info person;
```

Once the members (or fields) of `person` are filled with information, the entire variable may be written to a file using the `write` function:

```
file.write(reinterpret_cast<char*>(&person), sizeof(person));
```

The first argument is the address of the `person` variable. The `reinterpret_cast<char*>` cast operator is necessary because `write` expects the first argument to be a pointer to a `char`. When you pass the address of anything other than a `char` to the `write` function, you must make it look like a pointer to a `char` with the cast operator. The second argument is the `sizeof` operator. It tells `write` how many bytes to write to the file. Program 13-17 demonstrates this technique.



NOTE: Since structures can contain a mixture of data types, you should always use the `ios::binary` mode when opening a file to store them.

Program 13-17 allows you to build a file by filling the members of the `person` variable, then writing the variable to the file. Program 13-18 opens the file and reads each record into the `person` variable, then displays the information on the screen.

Program 13-17

```
1 //This program demonstrates the use of a structure variable to
2 //store a record of information to a file.
3 #include <iostream>
4 #include <fstream>
5 #include <cctype>  // for toupper
6 using namespace std;
7
8 const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
9
```

(program continues)

Program 13-17 *(continued)*

```
10 // Declare a structure for the record
11 struct Info
12 {
13    char name[NAME_SIZE];
14    int  age;
15    char address1[ADDR_SIZE];
16    char address2[ADDR_SIZE];
17    char phone[PHONE_SIZE];
18 };
19
20 int main()
21 {
22    Info person;    // Store information about a person
23    char response;  // User response
24
25    // Create file object and open file.
26    fstream people("people.dat", ios::out | ios::binary);
27    if (!people)
28    {
29       cout << "Error opening file. Program aborting.\n";
30       return 0;
31    }
32
33    // Keep getting information from user and writing it
34    // to the file in binary mode.
35    do
36    {
37       cout << "Enter the following information about a "
38            << "person:\n";
39       cout << "Name: ";
40       cin.getline(person.name, NAME_SIZE);
41       cout << "Age: ";
42       cin >> person.age;
43       cin.ignore(); // Skip over remaining newline.
44       cout << "Address line 1: ";
45       cin.getline(person.address1, ADDR_SIZE);
46       cout << "Address line 2: ";
47       cin.getline(person.address2, ADDR_SIZE);
48       cout << "Phone: ";
49       cin.getline(person.phone, PHONE_SIZE);
50       people.write(reinterpret_cast<char *>(&person),
51                    sizeof(person));
52       cout << "Do you want to enter another record? ";
53       cin >> response;
54       cin.ignore();
55    } while (toupper(response) == 'Y');
56
57    // Close file.
58    people.close();
59    return 0;
60 }
```

(program continues)

Program 13-17 *(continued)*

Program Screen Output with Example Input Shown in Bold

```
Enter the following information about a person:
Name: Charlie Baxter[Enter]
Age: 42[Enter]
Address line 1: 67 Kennedy Bvd.[Enter]
Address line 2: Perth, SC  38754[Enter]
Phone: (803)555-1234[Enter]
Do you want to enter another record? Y[Enter]
Enter the following information about a person:
Name: Merideth Murney[Enter]
Age: 22[Enter]
Address line 1: 487 Lindsay Lane[Enter]
Address line 2: Hazelwood, NC  28737[Enter]
Phone: (704)453-9999[Enter]
Do you want to enter another record? N[Enter]
```

Program 13-18

```
1  // This program demonstrates the use of a structure
2  // variable to read a record of information from a file.
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
8
9  // Declare a structure for the record
10 struct Info
11 {
12    char name[NAME_SIZE];
13    int age;
14    char address1[ADDR_SIZE];
15    char address2[ADDR_SIZE];
16    char phone[PHONE_SIZE];
17 };
18
19 int main()
20 {
21    Info person;   // Store person information
22    char response; // User response
23
24    // Create file object and open file for binary reading.
25    fstream people("people.dat", ios::in | ios::binary);
26    if (!people)
27    {
28       cout << "Error opening file. Program aborting.\n";
29       return 0;
30    }
31
32    // Label the output.
33    cout << "Here are the people in the file:\n\n";
```

(program continues)

Program 13-18 *(continued)*

```
34
35    // Read one structure at a time and echo to screen.
36    people.read(reinterpret_cast<char *>(&person),
37                sizeof(person));
38    while (!people.eof())
39    {
40       cout << "Name: ";
41       cout << person.name << endl;
42       cout << "Age: ";
43       cout << person.age << endl;
44       cout << "Address line 1: ";
45       cout << person.address1 << endl;
46       cout << "Address line 2: ";
47       cout << person.address2 << endl;
48       cout << "Phone: ";
49       cout << person.phone << endl;
50       cout << "\nStrike any key to see the next record.\n";
51       cin.get(response);
52       people.read(reinterpret_cast<char *>(&person),
53                   sizeof(person));
54    }
55    cout << "That's all the information in the file!\n";
56    people.close();
57    return 0;
58 }
```

Program Screen Output (Using the same file created by Program 13-17 as input)

```
Here are the people in the file:

Name: Charlie Baxter
Age: 42
Address line 1: 67 Kennedy Bvd.
Address line 2: Perth, SC  38754
Phone: (803)555-1234

Strike any key to see the next record.
Name: Merideth Murney
Age: 22
Address line 1: 487 Lindsay Lane
Address line 2: Hazelwood, NC  28737
Phone: (704)453-9999

Strike any key to see the next record.
That's all the information in the file!
```



NOTE: Structures containing pointers cannot be correctly stored to disk using the techniques of this section. This is because if the structure is read into memory on a subsequent run of the program, it cannot be guaranteed that all program variables will be at the same memory locations. Because `string` class objects contain implicit pointers, they cannot be a part of a structure that has to be stored.



Checkpoint

13.21 Describe how you set up a C++ program to have multiple files open at the same time.

13.22 How would the number 479 be stored in a text file? (Show the character and ASCII code representation.)

13.23 Describe the differences between the `write` member function and the << operator.

13.24 What are the purposes of the two arguments needed for the `write` member function?

13.25 What are the purposes of the two arguments needed for the `read` member function?

13.26 Describe the relationship between fields and records.

13.27 Assume the following structure declaration, variable, and file stream object definition exist in a program:

```
const int NAME_SIZE = 51;
struct Data
{
   char customer[NAME_SIZE];
   int num;
   double balance;
};
Data cust;
fstream file("stuff", ios::out | ios::binary);
```

Write a statement that uses the `write` member function to store the contents of `cust` in the file.

13.8 Random-Access Files

CONCEPT: Random access means nonsequentially accessing information in a file.

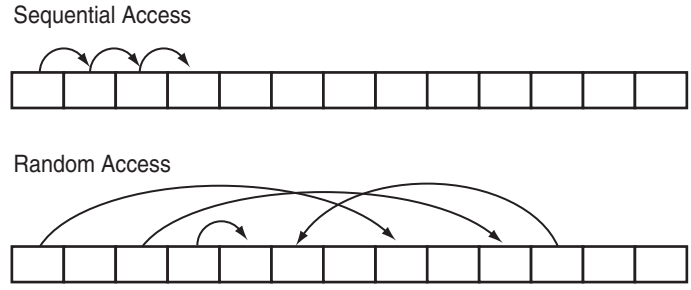
All of the programs created so far in this chapter have performed *sequential file access*. When a file is opened, the position where reading and/or writing will occur is at the file's beginning (unless the `ios::app` mode is used, which causes data to be written to the end of the file). If the file is opened for output, bytes are written to it one after the other. If the file is opened for input, data is read beginning at the first byte. As the reading or writing continues, the file stream object's read/write position advances sequentially through the file's contents.

The problem with sequential file access is that in order to read a specific byte from the file, all the bytes that precede it must be read first. For instance, if a program needs information stored at the 100th byte of a file, it will have to read the first 99 bytes to reach it. If you've ever searched for a song on a cassette tape, you understand sequential access. To find a song, you have to listen to all the songs that come before it, or fast-forward over them. There is no way to immediately jump to that particular song.

Although sequential file access is useful in many circumstances, it can slow a program down tremendously. If the file is very large, locating information buried deep inside it can

take a long time. Alternatively, C++ allows a program to perform *random file access*. In random file access, a program may immediately jump to any byte in the file without first reading the preceding bytes. The difference between sequential and random file access is like the difference between a cassette tape and a compact disc. When listening to a CD, there is no need to listen to or fast-forward over unwanted songs. You simply jump to the track that you want to listen to. This is illustrated in Figure 13-5.

Figure 13-5



The `seekp` and `seekg` Member Functions

File stream objects have two member functions that are used to move the read/write position to any byte in the file. They are `seekp` and `seekg`. The `seekp` function is used with files opened for output and `seekg` is used with files opened for input. (It makes sense if you remember that "p" stands for "put" and "g" stands for "get." `seekp` is used with files that you put information into, and `seekg` is used with files you get information out of.)

Here is an example of `seekp`'s usage:

```
file.seekp(20L, ios::beg);
```

The first argument is a `long` integer representing an offset into the file. This is the number of the byte you wish to move to. In this example, `20L` is used. (Remember, the L suffix forces the compiler to treat the number as a long integer.) This statement moves the file's write position to byte number 20. (All numbering starts at 0, so byte number 20 is actually the 21st byte.)

The second argument is called the mode flag, and it designates where to calculate the offset from. The flag `ios::beg` means the offset is calculated from the beginning of the file. Alternatively, the offset can be calculated from the end of the file or the current position in the file. Table 13-8 lists the flags for all three of the random-access modes.

Table 13-8 File Positioning Flags

Mode Flag	Description
`ios::beg`	The offset is calculated from the beginning of the file.
`ios::end`	The offset is calculated from the end of the file.
`ios::cur`	The offset is calculated from the current position.

Table 13-9 shows examples of `seekp` and `seekg` using the various mode flags.

Table 13-9 File Seek Operations

Statement	How It Affects the Read/Write Position
`file.seekp(32L, ios::beg);`	Sets the write position to the 33rd byte (byte 32) from the beginning of the file.
`file.seekp(-10L, ios::end);`	Sets the write position to the 11th byte (byte 10) from the end of the file.
`file.seekp(120L, ios::cur);`	Sets the write position to the 121st byte (byte 120) from the current position.
`file.seekg(2L, ios::beg);`	Sets the read position to the 3rd byte (byte 2) from the beginning of the file.
`file.seekg(-100L, ios::end);`	Sets the read position to the 101st byte (byte 100) from the end of the file.
`file.seekg(40L, ios::cur);`	Sets the read position to the 41st byte (byte 40) from the current position.
`file.seekg(0L, ios::end);`	Sets the read position to the end of the file.

Notice that some of the examples in Table 13-9 use a negative offset. Negative offsets result in the read or write position being moved backward in the file, while positive offsets result in a forward movement.

Assume the file `letters.txt` contains the following data:

```
abcdefghijklmnopqrstuvwxyz
```

Program 13-19 uses the `seekg` function to jump around to different locations in the file, retrieving a character after each stop.

Program 13-19

```
1  // This program demonstrates the seekg function.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main()
7  {
8     // Variable to access file.
9     char ch;
10
11    // Open the file for reading
12    fstream file("letters.txt", ios::in);
13    if (!file)
14    {
15      cout << "Error opening file.";
16      return 0;
17    }
18
```

(program continues)

Program 13-19 *(continued)*

```
19    // Get fifth byte from beginning of alphabet file.
20    file.seekg(5L, ios::beg);
21    file.get(ch);
22    cout << "Byte 5 from beginning: " << ch << endl;
23
24    // Get tenth byte from end of alphabet file.
25    file.seekg(-10L, ios::end);
26    file.get(ch);
27    cout << "Byte 10 from end: " << ch << endl;
28
29    // Go forward three bytes from current position
30    file.seekg(3L, ios::cur);
31    file.get(ch);
32    cout << "Byte 3 from current: " << ch << endl;
33
34    // Close file.
35    file.close();
36    return 0;
37 }
```

Program Screen Output

```
Byte 5 from beginning: f
Byte 10 from end: q
Byte 3 from current: u
```

Program 13-20 shows another example of the `seekg` function. It opens the `people.dat` file created by Program 13-17. The file contains two records. Program 13-20 displays record 1 (the second record) first, then displays record 0.

Program 13-20

```
1  // This program demonstrates the use of a structure
2  // variable to read a record of information from a file.
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
8
9  // Declare a structure for the record
10 struct Info
11 {
12    char name[NAME_SIZE];
13    int age;
14    char address1[ADDR_SIZE];
15    char address2[ADDR_SIZE];
16    char phone[PHONE_SIZE];
17 };
18
19 // Function Prototypes
20 long byteNum(int);
21 void showRec(Info);
22
```

(program continues)

Program 13-20 *(continued)*

```
23 int main()
24 {
25    // Person information.
26    Info person;
27
28    // Create file object and open the file.
29    fstream people("people.dat", ios::in | ios::binary);
30    if (!people)
31    {
32       cout << "Error opening file. Program aborting.\n";
33       return 0;
34    }
35
36    // Skip forward and read record 1 in the file.
37    cout << "Here is record 1:\n";
38    people.seekg(byteNum(1), ios::beg);
39    people.read(reinterpret_cast<char *>(&person),
40                sizeof(person));
41    showRec(person);
42
43    // Skip backwards and read record 0 in the file.
44    cout << "\nHere is record 0:\n";
45    people.seekg(byteNum(0), ios::beg);
46    people.read(reinterpret_cast<char *>(&person),
47                sizeof(person));
48    showRec(person);
49
50    // Close the file.
51    people.close();
52    return 0;
53 }
54
55
56 //*********************************************************
57 // Definition of function byteNum. Accepts an integer as     *
58 // its argument. Returns the byte number in the file of the  *
59 // record whose number is passed as the argument.            *
60 //*********************************************************
61 long byteNum(int recNum)
62 {
63    return sizeof(Info) * recNum;
64 }
65
66 //*********************************************************
67 // Definition of function showRec. Accepts an Info structure *
68 // as its argument, and displays the structure's contents.   *
69 //*********************************************************
```

(program continues)

Program 13-20 *(continued)*

```
70 void showRec(Info record)
71 {
72    cout << "Name: ";
73    cout << record.name << endl;
74    cout << "Age: ";
75    cout << record.age << endl;
76    cout << "Address line 1: ";
77    cout << record.address1 << endl;
78    cout << "Address line 2: ";
79    cout << record.address2 << endl;
80    cout << "Phone: ";
81    cout << record.phone << endl;
82 }
```

Program Screen Output (Using the same file created by Program 13-17 as input)

```
Here is record 1:
Name: Merideth Murney
Age: 22
Address line 1: 487 Lindsay Lane
Address line 2: Hazelwood, NC  28737
Phone: (704)453-9999

Here is record 0:
Name: Charlie Baxter
Age: 42
Address line 1: 67 Kennedy Bvd.
Address line 2: Perth, SC  38754
Phone: (803)555-1234
```

The program has two important functions other than `main`. The first, `byteNum`, takes a record number as its argument and returns that record's starting byte. It calculates the record's starting byte by multiplying the record number by the size of the `Info` structure. This returns the offset of that record from the beginning of the file. The second function, `showRec`, accepts an `Info` structure as its argument and displays its contents on the screen.

The `tellp` and `tellg` Member Functions

File stream objects have two more member functions that may be used for random file access: `tellp` and `tellg`. Their purpose is to return, as a `long` integer, the current byte number of a file's read and write position. As you can guess, `tellp` is used to return the write position and `tellg` is used to return the read position. Assuming `pos` is a `long` integer, here is an example of the functions' usage:

```
pos = outFile.tellp();
pos = inFile.tellg();
```

Program 13-21 demonstrates the `tellg` function. It opens the `letters.txt` file, which was also used in Program 13-19. The file contains the following characters:

```
abcdefghijklmnopqrstuvwxyz
```

Program 13-21

```
1  // This program demonstrates the tellg function.
2  #include <iostream>
3  #include <fstream>
4  #include <cctype> // For toupper
5  using namespace std;
6
7  int main()
8  {
9     // Variables used to read the file.
10    long offset;
11    char ch;
12
13    char response;   //User response
14
15    // Create the file object and open the file.
16    fstream file("letters.txt", ios::in);
17    if (!file)
18    {
19       cout << "Error opening file.";
20       return 0;
21    }
22
23    // Work with the file.
24    do
25    {
26       // Where in the file am I?
27       cout << "Currently at position "
28            << file.tellg() << endl;
29
30       // Get a file offset from the user.
31       cout << "Enter an offset from the "
32          << "beginning of the file: ";
33       cin >> offset;
34
35       // Read the character at the given offset.
36       file.seekg(offset, ios::beg);
37       file.get(ch);
38       cout << "Character read: "
39            << ch << endl;
40       cout << "Do it again? ";
41       cin >> response;
42    } while (toupper(response) == 'Y');
43    file.close();
44    return 0;
45 }
```

(program continues)

Program 13-21 *(continued)*

Program Output with Example Input Shown in Bold

```
Currently at position 0
Enter an offset from the beginning of the file: 5[Enter]
Character read: f
Do it again? y[Enter]
Currently at position 6
Enter an offset from the beginning of the file: 0[Enter]
Character read: a
Do it again? y[Enter]
Currently at position 1
Enter an offset from the beginning of the file: 20[Enter]
Character read: u
Do it again? n[Enter]
```

13.9 Opening a File for Both Input and Output

CONCEPT: **You may perform input and output on an `fstream` file without closing it and reopening it.**

Sometimes you'll need to perform both input and output on a file without closing and reopening it. For example, consider a program that allows you to search for a record in a file and then make changes to it. A read operation is necessary to copy the information from the file to memory. After the desired changes have been made to the information in memory, a write operation is necessary to replace the old data in the file with the new data in memory.

Such operations are possible with `fstream` objects. The `ios::in` and `ios::out` file access flags may be joined with the | operator, as shown in this example declaration:

```
fstream file("data.dat", ios::in | ios::out)
```

The same operation may be accomplished with the `open` member function:

```
file.open("data.dat", ios::in | ios::out);
```

You may also specify the `ios::binary` flag if binary data is to be written to the file. Here is an example:

```
file.open("data.dat", ios::in | ios::out | ios::binary);
```

When an `fstream` file is opened with both the `ios::in` and `ios::out` flags, the file's current contents are preserved and the read/write position is initially placed at the beginning of the file. If the file does not exist, it is created (unless the `ios::nocreate` is also used).

Programs 13-22, 13-23, and 13-24 demonstrate many of the techniques we have discussed. Program 13-22 sets up a file with five blank inventory records. Each record is a structure with members for holding a part description, quantity on hand, and price. Program 13-23 displays the contents of the file on the screen. Program 13-24 opens the file in both input and output modes and allows the user to change the contents of a specific record.

Program 13-22

```
1  // This program sets up a file of blank inventory records.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  const int DESC_SIZE = 31, NUM_RECORDS = 5;
7  // Declaration of Invtry structure.
8  struct Invtry
9  {
10    char desc[DESC_SIZE];
11    int qty;
12    double price;
13 };
14
15 int main()
16 {
17    // Variables needed to write the file.
18    Invtry record = { "", 0, 0.0 };
19
20    // Create file object and open file.
21    fstream inventory("invtry.dat", ios::out | ios::binary);
22    if (!inventory)
23    {
24       cout << "Error opening file.";
25       return 0;
26    }
27
28    // Now write the blank records.
29    for (int count = 0; count < NUM_RECORDS; count++)
30    {
31       cout << "Now writing record " << count << endl;
32       inventory.write(reinterpret_cast<char *>(&record),
33                       sizeof(record));
34    }
35
36    //Close the file.
37    inventory.close();
38    return 0;
39 }
```

Program Screen Output

```
Now writing record 0
Now writing record 1
Now writing record 2
Now writing record 3
Now writing record 4
```

Program 13-23 simply displays the contents of the inventory file on the screen. It can be used to verify that Program 13-22 successfully created the blank records and that Program 13-24 correctly modified the designated record.

Program 13-23

```
// This program displays the contents of the inventory file.
#include <iostream>
#include <fstream>
using namespace std;

const int DESC_SIZE = 31;

// Declaration of Invtry structure.
struct Invtry
{
   char desc[DESC_SIZE];
   int qty;
   double price;
};

int main()
{
   // Buffer used for reading.
   Invtry record;

   // Create and open the file for reading.
   fstream inventory("invtry.dat", ios::in | ios::binary);
   if (!inventory)
   {
      cout << "Error in opening the file.";
      return 0;
   }

   // Now read and display the records
   inventory.read(reinterpret_cast<char *>(&record),
                  sizeof(record));
   while (!inventory.eof())
   {
      cout << "Description: ";
      cout << record.desc << endl;
      cout << "Quantity: ";
      cout << record.qty << endl;
      cout << "Price: ";
      cout << record.price << endl << endl;
      inventory.read(reinterpret_cast<char *>(&record),
                     sizeof(record));
   }
   inventory.close();
   return 0;
}
```

Here is the screen output of Program 13-23 if it is run immediately after Program 13-22 sets up the file of blank records.

Program Screen Output

```
Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0
```

Program 13-24 allows the user to change the contents of an individual record in the inventory file.

Program 13-24

```
1  // This program allows the user to edit a specific
2  // record in the inventory file.
3  #include <iostream>
4  #include <fstream>
5  using namespace std;
6
7  const int DESC_SIZE = 31;
8  // Declaration of Invtry structure.
9  struct Invtry
10 {
11    char desc[DESC_SIZE];
12    int qty;
13    double price;
14 };
15
16 int main()
17 {
18    // Variables needed to read the file.
19    Invtry record;
20    long recNum;
21
```

(program continues)

Program 13-24 *(continued)*

```
22    // Open the file.
23    fstream inventory("invtry.dat", ios::in | ios::out |
24                          ios::binary);
25    if (!inventory)
26    {
27       cout << "Error opening file.";
28       return 0;
29    }
30
31    // Move to the desired record and read it into record.
32    cout << "Which record do you want to edit?";
33    cin >> recNum;
34    inventory.seekg(recNum * sizeof(record), ios::beg);
35    inventory.read(reinterpret_cast<char *>(&record),
36                   sizeof(record));
37
38    // Get new data from user and edit in-memory record.
39    cout << "Description: ";
40    cout << record.desc << endl;
41    cout << "Quantity: ";
42    cout << record.qty << endl;
43    cout << "Price: ";
44    cout << record.price << endl;
45    cout << "Enter the new data:\n";
46    cout << "Description: ";
47    cin.ignore();
48    cin.getline(record.desc, DESC_SIZE);
49    cout << "Quantity: ";
50    cin >> record.qty;
51    cout << "Price: ";
52    cin >> record.price;
53
54    // Move to the right place in file and write the record.
55    inventory.seekp(recNum * sizeof(record), ios::beg);
56    inventory.write(reinterpret_cast<char *>(&record),
57                    sizeof(record));
58
59    // Close the file.
60    inventory.close();
61    return 0;
62 }
```

Program Screen Output with Example Input Shown in Bold

```
Which record do you want to edit? 2[Enter]
Description:
Quantity: 0
Price: 0.0
Enter the new data:
Description: Wrench[Enter]
Quantity: 10[Enter]
Price: 4.67[Enter]
```



Checkpoint

13.28 Describe the difference between the `seekg` and the `seekp` functions.

13.29 Describe the difference between the `tellg` and the `tellp` functions.

13.30 Describe the meaning of the following file access flags.

```
ios::beg
ios::end
ios::cur
```

13.31 What is the number of the first byte in a file?

13.32 Briefly describe what each of the following statements do.

```
file.seekp(100L, ios::beg);
file.seekp(-10L, ios::end);
file.seekg(-25L, ios::cur);
file.seekg(30L, ios::cur);
```

13.33 Describe the mode that each of the following statements cause a file to be opened in.

```
file.open("info.dat", ios::in | ios::out);
file.open("info.dat", ios::in | ios::app);
file.open("info.dat", ios::in | ios::out | ios::ate);
file.open("info.dat", ios::in | ios::out | ios::binary);
```

13.10 Online Friendship Connections Case Study: *Object Serialization*

Online Friendship Connections is an online service that helps people meet and make new friends. People who want to join the club and use its services fill out a registration form, stating their names, age, contact information, gender, hobbies, personal interests, and other pertinent information about themselves. They also specify the qualities they are looking for in a new friend. The service will then try to get two people together if the personal information submitted indicates that there is a high probability of a good match.

Object Serialization

Online Friendship Connections will store information about its members in files. Member information will be manipulated by a C++ program, and will be stored in objects of appropriately designed classes. These objects may involve pointers to other objects, forming a network of objects whose structure must somehow be preserved when the data are stored to a file. This structure is then reconstructed when the data is read back from the file at a later time. The process of transforming complex networks of objects interconnected through pointers into a form that can be stored in a disk file (or on some other medium outside of central memory) is called *object serialization*.

In this section, we will illustrate some of the techniques used in serializing objects by looking at a simple case in which an object containing a C++ `string` object is serialized. Recall that C++ strings are normally implemented using pointers to dynamically allocated array of `char`.

Designing the Classes Needed by the Program

A simple class that stores a portion of the information submitted by members of Online Friendship Connections might include a first name, middle initial, last name, and the age of a member. In addition to the usual getter and setter functions, we need member functions for *serializing* the object: that is, a member function that converts the object into data stored in a file:

```
void store(ofstream &outFile);
```

We also need a member function for *deserializing* an object: that is, one that reads from a file data previously placed there by `store`, recovers its structure, and sets the data members of the object correctly:

```
void load(ifstream &inFile);
```

After adding a constructor and a `display` member function, we come up with the following class:

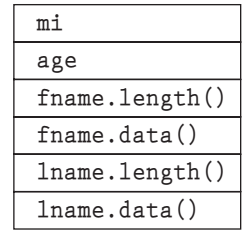
Contents of `serialization.h`

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class Person
{
  string fname, lname;
  char mi;
  int age;
public:
  string getFname(){return fname;}
  string getLname(){return lname;}
  char getMi(){return mi;}
  int getAge(){return age;}

  void setFname(string name){fname = name;}
  void setLname(string name){lname = name;}
  void setMi(char ch){mi = ch;}

  // read data from file
  void load(ifstream &inFile);
  // store data to file
  void store(ofstream  &outFile);

  // constructor
  Person(string fname = "", char mi = 0,
         string lname = "", int age = 0);

  void display()
  {
    cout << fname << " " << mi << " " << lname << endl
         << "Age : " << age << endl;
  }
};
```

Determining a Serialization Scheme

We cannot just write the contents of a `Person` object to a disk file because the string members contain pointers to arrays of characters which need to be stored so that the string objects can be reconstructed at a later time when the object is deserialized. Because strings have varying lengths, `Person` objects will occupy varying amounts of space on the disk when they are deserialized. A simple but effective serialization scheme is to first write all the members of the object that take up constant space, and then write each member whose space requirement may vary on the disk, preceded by the number of bytes that the member occupies. For the `Person` class, we can use the scheme shown in Figure 13-6.

Figure 13-6



The code for the `store` function is then very straightforward and can be seen in the listing of the file `serialization.cpp`. To design the `load` function, we note that we need to reconstruct the `fname` and `lname` strings by first reading their data portions into an in-memory `buffer` that is an array of character. To do this for `fname`, we first read the number of bytes occupied by its data portion from the file:

```
int firstNameLength;
inFile.read(addr(&firstNameLength), sizeof(int));
```

We must then read that many bytes into a buffer and null terminate the buffer to turn it into a C-string:

```
inFile.read(buffer, firstNameLength);
buffer[firstNameLength] = '\0';
```

Finally, we convert the C-string to a string object by assigning it to the `fname` member. The C++ string has a convert constructor that automatically converts C-strings to string objects to make such assignments possible.

```
frame = buffer;
```

The `buffer` array is used as a temporary holding place. Making it an instance member of the class would allocate space for it in every object and would waste a lot of memory. A better idea to make it a static member of the class, so that the scratch space can be shared by all members of the object. However, we note that it is only used by the `load` member function. For this reason, we make it local static. That way, space for the buffer is allocated once instead of being allocated anew for each call to `load`. Static local variables were described in Chapter 6.

The rest of the member functions needed to implement the `Person` class are shown in the listing of the `serialization.cpp` file.

Contents of `serialization.cpp`

```
#include "serialization.h"

Person::Person(string fname,  char mi,
               string lname,  int age)
{
   this->fname = fname;
   this->lname = lname;
   this->mi = mi;
   this->age = age;
}

//*****************************************
// Stores mi, age, then length of fname,      *
// then data for fname, then length of lname, *
// then data for lname                        *
//*****************************************
void Person::store(ofstream &outFile)
{
   outFile.write(&mi, sizeof(mi));
   outFile.write(reinterpret_cast<char *>(&age),
                 sizeof(age));

   // Write length and data for fname and lname.
   int firstNameLength = fname.length();
   outFile.write(reinterpret_cast<char *>(&firstNameLength),
                 sizeof(int));
   outFile.write(fname.data(), firstNameLength);
   int lastNameLength = lname.length();
   outFile.write(reinterpret_cast<char *>(&lastNameLength),
                 sizeof(int));
   outFile.write(lname.data(), lastNameLength);
}

//*****************************************
// Reads the data in the format written by     *
// Person::store                               *
//*****************************************
void Person::load(ifstream &inFile)
{
   const int BUFFER_SIZE = 256;
   static char buffer[256]; //used to read names

   inFile.read(&mi, sizeof(mi));
   inFile.read(reinterpret_cast<char *>(&age), sizeof(age));

   // First get length and data for fname.
   int firstNameLength;
   inFile.read(reinterpret_cast<char *>(&firstNameLength),
               sizeof(int));

```

```
51    // Read the data for fname into a local buffer.
52    inFile.read(buffer, firstNameLength);
53
54    // Null terminate the buffer.
55    buffer[firstNameLength] = '\0';
56    fname = buffer;  //take advantage of convert constructor
57
58    // Do the same thing for length and data for lname.
59    int lastNameLength;
60    inFile.read(reinterpret_cast<char *>(&lastNameLength),
61                sizeof(int));
62    inFile.read(buffer, lastNameLength);
63    buffer[lastNameLength] = '\0';
64    lname = buffer;
65 }
```

We need two separate programs to demonstrate the serialization capabilities of the `Person` class. Program 13-25, which generates no screen output, creates an array of two objects, serializes them, and writes them to a file.

Program 13-25

```
1  //This program demonstrates object serialization.
2  #include "serialization.h"
3  int main()
4  {
5     // Array of objects to store in file.
6     Person people[ ] =
7         { Person("Joseph", 'X', "Puff", 32),
8           Person("Louise", 'Y', "Me", 28)
9         };
10    // Open a file and store the array of people.
11    ofstream outFile("MorePeople.dat", ios::binary);
12    if(!outFile)
13    {
14      cout << "The output file cannot be opened";
15      exit(1);
16    }
17
18    // Store the people data in the file.
19    people[0].store(outFile);
20    people[1].store(outFile);
21    cout << "Data has been written to the file "
22         << " 'Morepeople.dat'";
23
24   // Close file.
25   outFile.close();
26   return 0;
27 }
```

Program 13-26 opens the file created by Program 13-25, deserializes the two objects in the file, and displays them on the screen.

Program 13-26

```
1  //This program demonstrates object deserialization.
2  #include "serialization.h"
3  int main()
4  {
5    const int NUM_PEOPLE = 2;
6    Person people[NUM_PEOPLE];
7    // Open a file and load the array of people.
8    ifstream inFile("MorePeople.dat", ios::binary);
9    if(!inFile)
10   {
11     cout << "The input file cannot be opened";
12     exit(1);
13   }
14
15   // Read the data from the file.
16   for (int k = 0; k < NUM_PEOPLE; k++)
17      people[k].load(inFile);
18
19   // Display the data.
20   for (int k = 0; k < NUM_PEOPLE; k++)
21      people[k].display();
22
23   // Close the file.
24   inFile.close();
25   return 0;
26 }
```

Program Output

```
Joseph X Puff
Age : 32
Louise Y Me
Age : 28
```

13.11 Tying It All Together: *File Merging and Color-Coded HTML*

Suppose that you have two files, with each file containing a sorted list of names and each name occurring on a line by itself, as illustrated in Table 13-10.

Table 13-10

Black File	Blue File
Abrams, Elaine	Avon, Martha
Bostrom, Andy	Gomez, Diane
Potus, Nicholas	Pistachio, Mary
Radon, Joseph	Rhodes, Peter
Williams, Nancy	Wilson, Zelda
	Zazinski, Pete

You want to merge the contents of the two files into one file in such a way that the merged file is sorted in alphabetic order. You also want people to be able to tell at a glance which of the two original files a given line in the merged file came from. One way to do this is to color-code the original files and then display each line of the merged file in the color of the originating file:

Abrams, Elaine
Avon, Martha
Bostrom, Andy
Gomez, Diane
Pistachio, Mary
Potus, Nicholas
Radon, Joseph
Rhodes, Peter
Williams, Nancy
Wilson, Zelda
Zazinski, Pete

To accomplish this, we code the output file in HTML and arrange for the browser to display each line with the appropriate color. This can be done via what are called CSS styles. We will use the HTML span elements to enclose a line and then color the content of the span element using the CSS style attribute. For the two files shown above, our program will produce the following output:

```
<span style = "color:black"> Abrams, Elaine <br/> </span>
<span style = "color:blue"> Avon, Martha <br/> </span>
<span style = "color:black"> Bostrom, Andy <br/> </span>
<span style = "color:blue"> Gomez, Diane <br/> </span>
<span style = "color:blue"> Pistachio, Mary <br/> </span>
<span style = "color:black"> Potus, Nicholas <br/> </span>
<span style = "color:black"> Radon, Joseph <br/> </span>
<span style = "color:blue"> Rhodes, Peter <br/> </span>
<span style = "color:black"> Williams, Nancy <br/> </span>
<span style = "color:blue"> Wilson, Zelda <br/> </span>
<span style = "color:blue"> Zazinski, Pete <br/> </span>
```

The `<br/>` HTML element signifies a line break.

Our solution to this problem will use a subclass of `fstream` that has a member function for writing a string inside of an HTML span element. The span element will specify the color the browser should use to display the string:

```
class ColorCodedStream : public fstream
{
public:
    void writeInColor(string str, string  aColor)
    {
        *this << "<span style = \"color:" + aColor + "\"> ";
        *this << str << " <br/> ";
        *this << "</span>\n";
    }
};
```

Our program will need to open two files for reading and a third file for writing. To avoid repetition of code, we write a function

```
void openFile(fstream &file, string descr);
```

that takes a file object and a description (`"black"`, `"blue"`, or `"output"`), prompts the user for the name of a file, and then opens the file. A file described as `"black"` or `"blue"` is opened for input, while a file described as `"output"` is opened for output.

Our program also uses the library function

```
getline(istream &in, string &str);
```

to read strings from the input file one line at a time. The program has two variables

```
string blackInput, blueInput
```

that are used to hold the line that was last read from the corresponding file. Because a read may be unsuccessful, the program tests each file object for errors before using the input last read from it. For example, the code

```
if (blackFile && !blueFile)
 {
     // Only blackInput is good
     outputFile.writeInColor(blackInput, "black");
     getline(blackFile, blackInput);
 }
```

determines that the last read from the black file was good while the read from the blue file failed, so it processes the input from the black file. After the input from the black file has been written out, the black file is read again to prepare for the next iteration of the loop. When the program cannot read any more data from either file (this is checked at the top of the loop that begins at line 39) the program terminates.

Program 13-27

```
1  // This program demonstrates file merging and the use
2  // of CSS to determine text colors in HTML documents.
3  #include <stdlib.h>
4  #include <iostream>
5  #include <fstream>
6  #include <string>
7  using namespace std;
8
9  // This subclass of fstream adds the ability to
10 // write a string that is automatically embedded
11 // in an HTML span element with a color specification
12 // style.
13 class ColorCodedStream : public fstream
14 {
15 public:
16      void writeInColor(string str, string  aColor)
17      {
18          *this << "<span style = \"color:" + aColor + "\"> ";
```

(program continues)

Program 13-27 *(continued)*

```
19          *this << str << " <br/> ";
20          *this << "</span>\n";
21      }
22  };
23
24  void openFile(fstream &file, string descr); // Prototype
25
26  int main()
27  {
28      ColorCodedStream outputFile;
29      fstream blackFile, blueFile;
30      openFile(blackFile, "black");
31      openFile(blueFile, "blue");
32      openFile(outputFile, "output");
33
34      string blackInput, blueInput;
35      // read the first line from each file
36      getline(blackFile, blackInput);   // Read black file into buffer
37      getline(blueFile, blueInput); // Read blue file into buffer
38
39      while (blackFile || blueFile)
40      {
41          if (blackFile &&  blueFile)
42          {
43              // Both buffers have fresh data
44              if (blackInput <= blueInput)
45              {
46                  outputFile.writeInColor(blackInput, "black");
47                  getline(blackFile, blackInput);
48              }
49              else
50              {
51                  outputFile.writeInColor(blueInput, "blue");
52                  getline(blueFile, blueInput);
53              }
54          }
55          if (blackFile && !blueFile)
56          {
57              // Only blackInput is good
58              outputFile.writeInColor(blackInput, "black");
59              getline(blackFile, blackInput);
60          }
61          if (blueFile && !blackFile)
62          {
63              // Only blueInput is good
64              outputFile.writeInColor(blueInput, "blue");
65              getline(blueFile, blueInput);
66          }
67      }
68      return 0;
69  }
```

(program continues)

Program 13-27 *(continued)*

```
70 //*************************************************************
71 // Opens a specified file for reading or writing. The descr argument *
72 // is used in prompting for the name of the file.                    *
73 //*************************************************************
74 void openFile(fstream &file, string descr)
75 {
76      string fileName;
77      cout << "Enter the  name of the " << descr << " file: ";
78      cin >> fileName;
79
80      // Determine whether the file should be opened for reading
81      // or writing based on the description (descr).
82      if (descr == "output")
83          file.open(fileName.data(), ios::out);
84      else
85          file.open(fileName.data(), ios::in);
86
87      // Check if file open was successful
88      if (!file)
89      {
90          cout << "Cannot open the file " << fileName;
91          exit(1);
92      }
93 }
```

Sample Program Interaction with User Input Shown in Bold

```
Enter the name of the black file: blackfile.txt[Enter]
Enter the name of the blue file: bluefile.txt[Enter]
Enter the name of the output file: mergedfile.html[Enter]
```

The contents of the output file can be viewed in a browser.

Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. All files are assigned a(n)_____ that is used for identification purposes by the operating system and the user.
2. Before a file can be used, it must first be _____.
3. When a program is finished using a file, it should _____ it.
4. The _____ header file is required for file I/O operations.
5. The three file stream data types are _____, _____, and _____.
6. The _____ file stream data type is for output files.
7. The _____ file stream data type is for input files.

8. The _____ file stream data type is for output files, input files, or files that perform both input and output.
9. Write a statement that defines a file stream object named `people`. The object will be used for file output.
10. Write a statement that defines a file stream object named `pets`. The object will be used for file input.
11. Write a statement that defines a file stream object named `places`. The object will be used for both output and input.
12. Write two statements that use the `people` file stream object to open a file named `people.dat`. (Show how to open the file with a member function and at definition.) The file should be opened for output.
13. Write two statements that use the `pets` file stream object to open a file named `pets.dat`. (Show how to open the file with a member function and at definition.) The file should be opened for input.
14. Write two statements that use the `places` file stream object to open a file named `places.dat`. (Show how to open the file with a member function and at definition.) The file should be opened for both input and output.
15. If a file fails to open, the file stream object will be set to _____.
16. Write a program segment that defines a file stream object named `employees`. The file should be opened for both input and output (in binary mode). If the file fails to open, the program segment should display an error message.
17. The same formatting techniques used with _____ may also be used when writing information to a file.
18. The _____ member function reports when the end of the file has been encountered.
19. The _____ member function reads a line of text from a file.
20. The _____ member function reads a single character from a file.
21. The _____ member function writes a single character to a file.
22. _____ files contain data that is unformatted and not necessarily stored as ASCII text.
23. _____ files contain information formatted as ASCII text.
24. A record is a complete set of information about a single item and is made up of _____.
25. In C++, _____ provide a convenient way to organize information into fields and records.
26. The _____ member function writes “raw” binary data to a file.
27. The _____ member function reads “raw” binary data from a file.

28. The _____ operator is necessary if you pass anything other than a pointer to `char` as the first argument of the two functions mentioned in questions 26 and 27.
29. In _____ file access, the contents of the file are read in the order they appear in the file, from the file's start to its end.
30. In _____ file access, the contents of a file may be read in any order.
31. The _____ member function moves a file's read position to a specified byte in the file.
32. The _____ member function moves a file's write position to a specified byte in the file.
33. The _____ member function returns a file's current read position.
34. The _____ member function returns a file's current write position.
35. The _____ mode flag causes an offset to be calculated from the beginning of a file.
36. The _____ mode flag causes an offset to be calculated from the end of a file.
37. The _____ mode flag causes an offset to be calculated from the current position in the file.
38. A negative offset causes the file's read or write position to be moved _____ in the file from the position specified by the mode.

Algorithm Workbench

39. Give a pseudocode algorithm for determining the length of a file: that is, the number of bytes that are stored in the file.
40. Give a pseudocode algorithm for comparing two files to see if their contents are identical.
41. Design a pseudocode algorithm for reversing the contents of a text file into another file. Assume that the amount of memory is limited, so that you cannot read the entire source file into memory before you start writing it to a second file in reverse order.
42. Suppose that you have two text files that contain sequences of integers separated by white space (blank space, tabs, and line breaks). The integers in both files appear in sorted order, with smaller values near the beginning of the file and large values closer to the end. Write a pseudocode algorithm that merges the two sequences into a single sorted sequence that is written to a third file.

Find the Error

43. Each of the following programs or program segments has errors. Find as many as you can.

A)
```
fstream file(ios::in | ios::out);
file.open("info.dat");
if (!file)
{
   cout << "Could not open file.\n";
}
```

B)
```
ofstream file;
file.open("info.dat", ios::in);
if (file)
{
   cout << "Could not open file.\n";
}
```
C)
```
fstream file("info.dat");
if (!file)
{
   cout << "Could not open file.\n";
}
```
D)
```
fstream dataFile("info.dat", ios:in | ios:binary);
int x = 5;
dataFile << x;
```
E)
```
fstream dataFile("info.dat", ios:in);
int x;
while (dataFile.eof())
{
   dataFile >> x;
   cout << x << endl;
}
```
F)
```
fstream dataFile("info.dat", ios:in);
char line[81];
dataFile.getline(line);
```
G)
```
fstream dataFile("info.dat", ios:in);
char stuff[81];
dataFile.get(stuff);
```
H)
```
fstream dataFile("info.dat", ios:in);
char stuff[81] = "abcdefghijklmnopqrstuvwxyz";
dataFile.put(stuff);
```
I)
```
fstream dataFile("info.dat", ios:out);
struct Date
{
   int month;
   int day;
   int year;
};
Date dt = { 4, 2, 98 };
dataFile.write(&dt, sizeof(int));
```
J)
```
fstream inFile("info.dat", ios:in);
int x;
inFile.seekp(5);
inFile >> x;
```

Soft Skills

44. Learning to look beyond the symptoms of a problem to identify the root cause is an important skill. Bugs in a program are sometimes the result of careless mistakes, but at other times, they reflect a fundamental misunderstanding of some concept.

Suppose that a friend has been trying to determine why his file processing program is not working correctly. You notice that he is passing file objects to functions by value. In addition to simply telling your friend that file parameters need to be passed by reference, what can you tell him that will help him understand *why* files need to be passed by reference?

Suppose now that you need to demonstrate this bug to other people. Bugs usually occur in the context of a larger program, which can make it difficult for a person unfamiliar with the program to understand what is happening. Write a program that is as short as possible, but still has the file-passing bug.

Programming Challenges



1. File Previewer

Write a program that asks the user for the name of a text file. The program should display the first 10 lines of the file on the screen. If the file has fewer than 10 lines, the entire file should be displayed along with a message indicating the entire file has been displayed.

2. File Display Program

Write a program that asks the user for the name of a file. The program should display the contents of the file on the screen. If the file's contents won't fit on a single screen, the program should display 24 lines of output at a time, and then pause. Each time the program pauses, it should wait for the user to type a key before the next 24 lines are displayed.



3. Punch Line

Write a program that reads and prints a joke and its punch line from two different files. The first file contains a joke, but not its punch line. The second file has the punch line as its last line, preceded by "garbage." The main function of your program should open the two files and then call two functions, passing each one the file it needs. The first function should read and display each line in the file it is passed (the joke file). The second function should display only the last line of the file it is passed (the punch line file). It should find this line by seeking to the end of the file and then backing up to the beginning of the last line. Data to test your program can be found in the `joke.dat` and `punchline.dat` files.

4. Tail of a File

Write a program that asks the user for the name of a text file. The program should display the last 10 lines of the file on the screen (the "tail" of the file). If the file has less than 10 lines, the entire file is displayed, with a message that the entire file has been displayed. The program should do this by seeking to the end of the file and then backing up to the tenth line from the end.

5. String Search

Write a program that asks the user for the name of a file and a string to search for. The program will search the file for all occurrences of the specified string and display all lines that contain the string. After all occurrences have been located, the program should report the number of times the string appeared in the file.



6. Sentence Filter

A program that processes an input file and produces an output file is called a *filter*. Write a program that asks the user for two file names. The first file will be opened for input and the second file will be opened for output. (It will be assumed that the first file contains sentences that end with a period.) The program will read the contents of the first file and change all the letters other than the first letter of sentences to lowercase. First letter of sentences should be made uppercase. The revised contents should be stored in the second file.



VideoNote
Solving the File Encryption Filter Problem

7. File Encryption Filter

File encryption is the science of writing the contents of a file in a secret code. Your encryption program should work like a filter, reading the contents of one file, modifying the information into a code, and then writing the coded contents out to a second file. The second file will be a version of the first file, but written in a secret code.

Although there are complex encryption techniques, you should come up with a simple one of your own. For example, you could read the first file one character at a time, and add 10 to the ASCII code of each character before it is written to the second file.

8. File Decryption Filter

Write a program that decrypts the file produced by the program in Programming Challenge 7. The decryption program should read the contents of the coded file, restore the information to its original state, and write it to another file.

9. Letter Frequencies

The letter *e* is the most frequently used letter in English prose, and the letter *z* is the least frequently used. A friend of yours doing a sociology experiment believes that this may not necessarily be true of the writings of first-year college students. To test his theory, he asks you to write a program that will take a text file and print, for each letter of the English alphabet, the number of times the letter appears in the file.

Hint: Use an integer array of size 128, and use the ASCII values of letters to index into the array to store and retrieve counts for the letters.

10. Corporate Sales Data Output

Write a program that uses a structure to store the following information on a company division:

Division name (such as East, West, North, or South)
Quarter (1, 2, 3, or 4)
Quarterly sales

The user should be asked for the four quarters' sales figures for the East, West, North, and South divisions. The information for each quarter for each division should be written to a file.

Input Validation: Do not accept negative numbers for any sales figures.

11. Corporate Sales Data Input

Write a program that reads the information in the file created by the program in Programming Challenge 9. The program should calculate and display the following figures:

- Total corporate sales for each quarter
- Total yearly sales for each division
- Total yearly corporate sales
- Average quarterly sales for the divisions
- The highest and lowest quarters for the corporation

12. Inventory Program

Write a program that uses a structure to store the following inventory information in a file:

Item description
Quantity on hand
Wholesale cost
Retail cost
Date added to inventory

The program should have a menu that allows the user to perform the following tasks:

- Add new records to the file
- Display any record in the file
- Change any record in the file

Input Validation: The program should not accept quantities, or wholesale or retail costs less than 0. The program should not accept dates that the programmer determines are unreasonable.

13. Inventory Screen Report

Write a program that reads the information in the file created by the program in Programming Challenge 11. The program should calculate and display the following information:

- The total wholesale value of the inventory
- The total retail value of the inventory
- The total quantity of all items in the inventory

Group Project

14. Customer Accounts

This program should be designed and written by a team of students. Here are some suggestions:

- One student should design function `main`, which will call other program functions or class member functions. The remainder of the functions will be designed by other members of the team.
- The requirements of the program should be analyzed so each student is given about the same workload.

Write a program that uses a structure to store the following information about a customer account:

- Name
- Address
- City, state, and ZIP
- Telephone number
- Account balance
- Date of last payment

The structure should be used to store customer account records in a file. The program should have a menu that lets the user perform the following operations:

- Enter new records into the file
- Search for a particular customer's record and display it
- Search for a particular customer's record and delete it
- Search for a particular customer's record and change it
- Display the contents of the entire file

Input Validation: When the information for a new account is entered, be sure the user enters data for all the fields. No negative account balances should be entered.

This page intentionally left blank

CHAPTER 14

Recursion

TOPICS

14.1 Introduction to Recursion
14.2 The Recursive Factorial Function
14.3 The Recursive gcd Function
14.4 Solving Recursively Defined Problems
14.5 A Recursive Binary Search Function
14.6 Focus on Problem Solving and Program Design: *The QuickSort Algorithm*
14.7 The Towers of Hanoi
14.8 Focus on Problem Solving: *Exhaustive and Enumeration Algorithms*
14.9 Focus on Software Engineering: *Recursion Versus Iteration*
14.10 Tying It All Together: *Infix and Prefix Expressions*

14.1 Introduction to Recursion

CONCEPT: A recursive function is one that calls itself.

You have seen instances of functions calling other functions. Function *A* can call function *B*, which can then call Function *C*. It's also possible for a function to call itself. A function that calls itself is a *recursive function*. Look at this `message` function:

```
void message()
{
   cout << "This is a recursive function.\n";
   message();
}
```

This function displays the string `"This is a recursive function.\n"`, and then calls itself. Each time it calls itself, the cycle is repeated. Can you see a problem with the function? There's no way to stop the recursive calls. This function is like an infinite loop because there is no code to stop it from repeating.

To be useful, a recursive function must have a way of controlling the number of recursive calls. The following is a modification of the `message` function. It passes an integer argument that holds the number of times the function is to call itself.

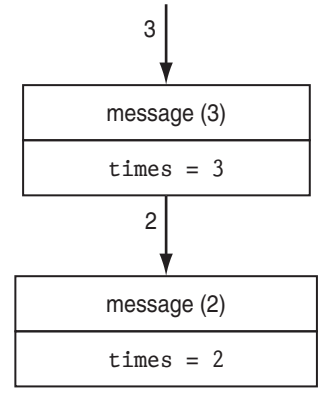
```
void message(int times)
{
   if (times > 0)
   {
      cout << "This is a recursive function.\n";
      message(times - 1);
   }
}
```

This function contains an `if` statement that controls the recursion. As long as the `times` argument is greater than zero, it will display the message and call itself again. Each time it calls itself, it passes `times - 1` as the argument. For example, let's say a program calls the function with the following statement:

```
message(3);
```

The argument, 3, will cause the function to be called four times. The first time the function is called, the `if` statement will display the message and call itself with 2 as the argument. Figure 14-1 illustrates this.

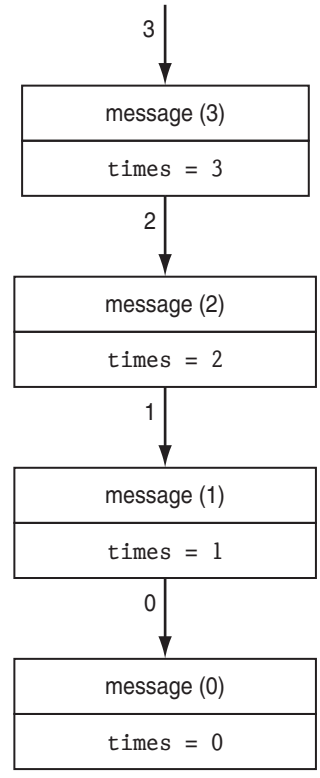
Figure 14-1



The diagram in Figure 14-1 illustrates two separate calls of the `message` function. Each time the function is called, a new instance of the `times` parameter is created in memory. The first time the function is called, the `times` parameter is set to 3. When the function calls itself, a new instance of `times` is created, and the value 2 is passed into it. This cycle repeats until zero is passed to the function. This is illustrated in Figure 14-2.

As you can see from Figure 14-2, the function will be called four times, so the *depth of recursion* is four. When the function reaches the fourth call, the `times` parameter will be set to 0. At that point, the `if` statement will stop the recursive chain of calls and the

Figure 14-2



fourth instance of the function will return. Control of the program will return from the fourth instance of the function to the point in the third instance directly after the recursive function call:

```
if (times > 0)
{
   cout << "This is a recursive function.\n";
   message(times - 1);
}
```

Control returns here.

Because there are no more statements to be executed after the function call, the third instance of the function returns control of the program back to the second instance. This repeats until all instances of the function return. Program 14-1 demonstrates the recursive `message` function, modified to show the value of the parameter to each call.

Program 14-1

```
1 // This program demonstrates a simple recursive function.
2 #include <iostream>
3 using namespace std;
4
```

(program continues)

Program 14-1 *(continued)*

```
// Function prototype
void message(int);

int main()
{
   message(3);
   return 0;
}

//*********************************************************
// Definition of function message. If the value in times    *
// is greater than 0, the message is displayed and the      *
// function is recursively called with the argument         *
// times - 1.                                               *
//*********************************************************
void message(int times)
{
   if (times > 0)
   {
      cout << "Message " << times << "\n";
      message(times - 1);
   }
}
```

Program Output

```
Message 3
Message 2
Message 1
```

To further illustrate the inner workings of this recursive function, let's look at another version of the program. In Program 14-2, a message is displayed each time the function is entered, and another message is displayed just before the function returns.

Recursive functions work by breaking a complex problem down into subproblems of the same type. This breaking down process stops when it reaches a *base case*, that is, a subproblem that is simple enough to be solved directly. For example, in the recursive `message` function of the preceding examples, the base case is when the parameter `times` is 0.

Program 14-2

```
// This program demonstrates a simple recursive function.
#include <iostream>
using namespace std;

// Function prototype
void message(int);

```

(program continues)

Program 14-2 *(continued)*

```
 8 int main()
 9 {
10    message(3);
11    return 0;
12 }
13
14 //*********************************************************
15 // Definition of function message. If the value in times    *
16 // is greater than 0, the message is displayed and the      *
17 // function is recursively called with the argument         *
18 // times - 1.                                               *
19 //*********************************************************
20 void message(int times)
21 {
22    cout << "Message " << times << ".\n";
23    if (times > 0)
24    {
25       message(times - 1);
26    }
27    cout << "Message " << times << " is returning.\n";
28 }
```

Program Output

```
Message 3.
Message 2.
Message 1.
Message 0.
Message 0 is returning.
Message 1 is returning.
Message 2 is returning.
Message 3 is returning.
```

You should consider the use of recursion when there is a way to express the solution of a problem in terms of solutions of simpler, or smaller, problems of the same type. As an example, one can envision sorting a long list of names by splitting the list into two sublists and assigning the two sublists to two different people to sort. Once the sublists are sorted, they can be merged into a sorted version of the original list by a suitable collating process. In this case, the problems of sorting the sublists are the simpler problems of the same type, and the base cases occur when the sublists consist of a single name.

Let's look at a simple example of recursion that performs a useful task. The function `numChars` counts the number of times a specific character appears in a string.

```
int numChars(char ch, char str[], int subscript)
{
   if (str[subscript] == '\0') // base case
      return 0;
   if (str[subscript] == ch)
```

```
         return 1 + numChars(ch, str, subscript+1);
      else
         return numChars(ch, str, subscript+1);
   }
```

The function's parameters are

- `ch`: the character to be searched for and counted
- `str`: an array containing the string to be searched
- `subscript`: the starting subscript for the search

The first `if` statement determines whether the base case, that is, the end of the string, has been reached:

```
if (str[subscript] == '\0')
```

If the end of the string has been reached, the function returns 0, indicating there are no more characters to count. Otherwise, the following `if` statement is executed:

```
if (str[subscript] == ch)
   return 1 + numChars(ch, str, subscript+1);
else
   return numChars(ch, str, subscript+1);
```

If `str[subscript]` contains the search character, the function performs a recursive call. The return statement returns `1` + the number of times the search character appears in the string, starting at `subscript + 1`. If `str[subscript]` does not contain the search character, a recursive call is made to search the remainder of the string. Program 14-3 demonstrates the program.

Program 14-3

```
1  // This program demonstrates a recursive function for
2  // counting the number of times a character appears
3  // in a string.
4  #include <iostream>
5  using namespace std;
6
7  // Function prototype
8  int numChars(char, char [], int);
9
10 int main()
11 {
12    char array[] = "abcddddef";
13
14    cout << "The letter d appears "
15         << numChars('d', array, 0) << " times.\n";
16    return 0;
17 }
18
```

(program continues)

Program 14-3 *(continued)*

```
19 //***********************************************
20 // Function numChars. This recursive function    *
21 // counts the number of times the character      *
22 // search appears in the string str. The search  *
23 // begins at the subscript stored in subscript.  *
24 //***********************************************
25 int numChars(char search, char str[], int subscript)
26 {
27    if (str[subscript] == '\0')   //base case
28       return 0;
29    if (str[subscript] == search)
30       return 1 + numChars(search, str, subscript+1);
31    else
32       return numChars(search, str, subscript+1);
33 }
```

Program Output

```
The letter d appears 4 times.
```

Direct and Indirect Recursion

The examples we have discussed so far show recursive functions that directly call themselves. This is known as *direct recursion*. There is also the possibility of creating *indirect recursion* in a program. This occurs when function A calls function B, which in turn calls function A. There can even be several functions involved in the recursion. For example, function A could call function B, which could call function C, which calls function A.



Checkpoint

14.1 What is a recursive function's base case?

14.2 What happens if a recursive function does not handle base cases correctly?

14.3 What will the following program display?

```
#include <iostream>
using namespace std;

// Function prototype
void showMe(int arg);

int main()
{
   int num = 0;

   showMe(num);
   return 0;
}
```

```
void showMe(int arg)
{
   if (arg < 10)
    showMe(++arg);
   else
    cout << arg << endl;
}
```

14.4 What is the difference between direct and indirect recursion?

14.2 The Recursive Factorial Function

CONCEPT: The recursive factorial function accepts an argument and calculates its factorial. Its base case is when the argument is 0.

Let's use an example from mathematics to examine an application of recursion. In mathematics, the notation *n!* represents the *factorial* of the number *n*. The factorial of an integer *n* is defined as

$$n! = 1 \times 2 \times 3 \times \ldots \times n; \text{ if } n > 0$$
$$= 1; \text{ if } n = 0$$

The rule states that when *n* is greater than 0, its factorial is the product of all the positive integers from 1 up to *n*. For instance, 6! can be calculated as $1 \times 2 \times 3 \times 4 \times 5 \times 6$. The rule also specifies the base case: the factorial of 0 is 1.

We can define the factorial of a number using recursion as follows:

$$\text{factorial}(n) = n \times \text{factorial}(n - 1) \text{ if } n > 0$$
$$= 1; \text{ if } n = 0$$

The C++ implementation of this recursive definition is

```
int factorial(int num)
{
   if (num == 0) // base case
      return 1;
   else
      return num * factorial(num – 1);
}
```

Consider a program that displays the value of 3! with the following statement:

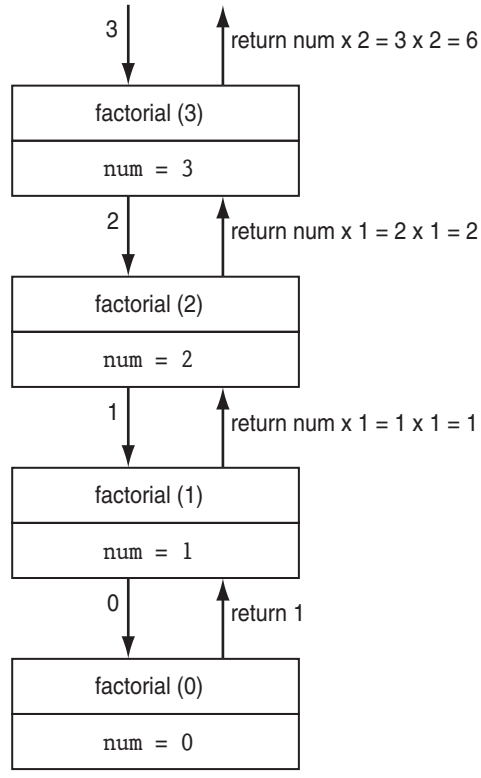
```
cout << factorial(3) << endl;
```

The first time the function is called, `num` is set to 3. The `if` statement will execute the following line:

```
return num * factorial(num - 1);
```

Although this is a `return` statement, it does not immediately return. Before the return value can be determined, the value of `factorial(num - 1)` must be determined. The function is called recursively until the fourth call, in which the `num` parameter will be set to zero. The diagram in Figure 14-3 illustrates the value of `num` and the return value during each call of the function.

Figure 14-3



Program 14-4 demonstrates the `factorial` function.

Program 14-4

```
1  // This program demonstrates a recursive function
2  // to calculate the factorial of a number.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype
7  int factorial(int);
8
9  int main()
10 {
11    int number;
12
13    cout << "Enter an integer value and I will display\n";
14    cout << "its factorial: ";
15    cin  >> number;
16    cout << "The factorial of " << number << " is ";
17    cout << factorial(number)   << endl;
18    return 0;
19    }
20
```

(program continues)

Program 14-4 *(continued)*

```
21 //*************************************************
22 // Definition of factorial. A recursive function to  *
23 // calculate the factorial of the parameter, num.    *
24 //*************************************************
25 int factorial(int num)
26 {
27    if (num == 0)  //base case
28       return 1;
29    else
30       return num * factorial(num - 1);
31 }
```

Program Output with Example Input

```
Enter an integer value and I will display
its factorial: 4
The factorial of 4 is 24
```

14.3 The Recursive gcd Function

CONCEPT: **There is a recursive method for finding the greatest common divisor (gcd) of two numbers.**

Our next example of recursion is the calculation of the greatest common divisor, or gcd, of two numbers. Using Euclid's algorithm, the gcd of two positive integers, x and y, is

$$\begin{aligned} \gcd(x, y) &= y && \text{if } y \text{ divides } x \text{ with no remainder} \\ &= \gcd(y, \text{remainder of } x/y); && \text{otherwise} \end{aligned}$$

This definition states that the gcd of x and y is y if x/y has no remainder. Otherwise, the answer is the gcd of y and the remainder of x/y. Program 14-5 shows the recursive C++ implementation:

Program 14-5

```
1  // This program demonstrates a recursive function to
2  // calculate the greatest common divisor (gcd) of two
3  // numbers.
4  #include <iostream>
5  using namespace std;
6
7  // Function prototype
8  int gcd(int, int);
9
10 int main()
11 {
12    int num1, num2;
13
```

(program continues)

Program 14-5 *(continued)*

```
14    cout << "Enter two integers: ";
15    cin >> num1 >> num2;
16    cout << "The greatest common divisor of " << num1;
17    cout << " and " << num2 << " is ";
18    cout << gcd(num1, num2) << endl;
19    return 0;
20 }
21
22 //*********************************************************
23 // Definition of gcd. This function uses recursion to     *
24 // calculate the greatest common divisor of two integers, *
25 // passed into the parameters x and y.                     *
26 //*********************************************************
27 int gcd(int x, int y)
28 {
29    if (x % y == 0)    //base case
30       return y;
31    else
32       return gcd(y, x % y);
33 }
```

Program Output with Example Input Shown in Bold

```
Enter two integers: 49 28
The greatest common divisor of 49 and 28 is 7
```

14.4 Solving Recursively Defined Problems

CONCEPT: **Some problems naturally lend themselves to recursive solutions.**

Some problems naturally lend themselves to recursive solutions. One well-known example is the calculation of *Fibonacci numbers*. The Fibonacci numbers, named after the Italian mathematician Leonardo Fibonacci (born circa 1170), form the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

Notice that after the second number, each number in the sequence is the sum of the two previous numbers. The Fibonacci sequence can be defined as:

$F_0 = 0$,
$F_1 = 1$,
$F_N = F_{N-1} + F_{N-2}$ for all $N \geq 2$.

It is clear that the problem of computing a Fibonacci number other than the first two can be reduced to the problems of computing the two preceding Fibonacci numbers. Thus this problem makes a good candidate for a recursive solution. The problems of computing the first two Fibonacci numbers are the base cases. Here is the recursive C++ function for computing the nth number in the Fibonacci sequence:

```
int fib(int n)
{
   if (n <= 0)       // base case
      return 0;
   else if (n == 1)  // base case
      return 1;
   else
      return fib(n - 1) + fib(n - 2);
}
```

The function is demonstrated in Program 14-6, which displays the first 10 numbers in the Fibonacci sequence.

Program 14-6

```
1  // This program demonstrates a recursive function
2  // that calculates Fibonacci numbers.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype
7  int fib(int);
8
9  int main()
10 {
11    cout << "The first 10 Fibonacci numbers are:\n";
12    for (int x = 0; x < 10; x++)
13        cout << fib(x) << " ";
14    cout << endl;
15    return 0;
16 }
17
18 //****************************************
19 // Function fib. Accepts an int argument  *
20 // in n. This function returns the nth    *
21 // Fibonacci number.                      *
22 //****************************************
23
24 int fib(int n)
25 {
26    if (n <= 0)        //base case
27       return 0;
28    else if (n == 1)   //base case
29       return 1;
30    else
31       return fib(n - 1) + fib(n - 2);
32 }
```

Program Output

```
The first 10 Fibonacci numbers are:
0 1 1 2 3 5 8 13 21 34
```

Another such example is Ackermann's function. A Programming Challenge at the end of this chapter asks you to write a recursive function that calculates Ackermann's function.

14.5 A Recursive Binary Search Function

CONCEPT: The binary search algorithm can be defined as a recursive function.



VideoNote
Recursive Binary Search

In Chapter 9 you learned about the binary search algorithm and how it can be used to search a sorted array for a given value. Let us look to see how the binary search algorithm can be formulated using recursion. Suppose that we want to write the function so that it has prototype

```
int binarySearch(int array[], int first, int last, int value)
```

where the parameter `array` is the array to be searched; the parameter `first` holds the subscript of the first element in the search range (the portion of the array to be searched); the parameter `last` holds the subscript of the last element in the search range; and the parameter `value` holds the value to be searched for. The function will return the subscript of `value` if it is found within the array, and will return –1 otherwise.

In order to use recursion, we need to find a way of breaking down the problem of searching a range of a sorted array for a given value into smaller problems of the same type. We start by comparing value to the middle element of the search range. If `value` is equal to the middle element, we are done and we return the subscript of the middle element. Otherwise, if `value` is smaller than the middle element, then we must search for it in the lower half of the original range (a recursive call on a smaller problem of the same type); but if `value` is larger than the middle element, we must search for it in the upper half of the original range. Notice that every time we make a recursive call, the search range will be smaller. The base case is when the search range is empty. Here is the function:

```
int binarySearch(int array[], int first, int last, int value)
{
   int middle;         // mid point of search

   if (first > last)   // base case
      return -1;
   middle = (first + last) / 2;
   if (array[middle] == value)
      return middle;
   if (array[middle] < value)
      return binarySearch(array, middle+1,last,value);
   else
      return binarySearch(array, first,middle-1,value);
}
```

This function is demonstrated in Program 14-7.

Program 14-7

```
1 // This program demonstrates a recursive function that
2 // performs a binary search on an integer array.
3 #include <iostream>
4 using namespace std;
5
```

(program continues)

Program 14-7 *(continued)*

```
// Function prototype
int binarySearch(int [], int, int, int);

const int SIZE = 20;

int main()
{
   int tests[SIZE] = { 101, 142, 147, 189, 199, 207, 222,
                       234, 289, 296, 310, 319, 388, 394,
                       417, 429, 447, 521, 536, 600};
   int result; // Result of the search
   int  empID; // What to search for

   cout << "Enter the Employee ID you wish to search for: ";
   cin >> empID;
   result = binarySearch(tests, 0, SIZE - 1, empID);
   if (result == -1)
      cout << "That number does not exist in the array.\n";
   else
   {
      cout << "That ID is found at element " << result;
      cout << " in the array\n";
   }
return 0;
}

//*********************************************************
// The binarySearch function performs a recursive binary   *
// search on a range of elements of an integer array. The  *
// parameter first holds the subscript of the range's      *
// starting element, and last holds the subscript of the   *
// ranges's last element. The parameter value holds the    *
// the search value. If the search value is found, its     *
// array subscript is returned. Otherwise, -1 is returned  *
// indicating the value was not in the array.              *
//*********************************************************
int binarySearch(int array[], int first, int last, int value)
{
   int middle;          // Mid point of search

   if (first > last)    // Base case
      return -1;
   middle = (first + last)/2;
   if (array[middle]==value)
      return middle;
   if (array[middle]<value)
      return binarySearch(array, middle+1,last,value);
   else
      return binarySearch(array, first,middle-1,value);
}
```

Program Output with Example Input Shown in Bold

```
Enter the Employee ID you wish to search for: 521 [Enter]
That ID is found at element 17 in the array
```

14.6 Focus on Problem Solving and Program Design: *The QuickSort Algorithm*

CONCEPT: **The QuickSort algorithm uses recursion to sort lists efficiently.**

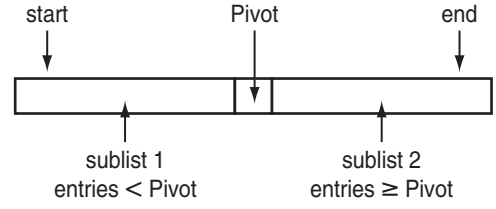


VideoNote
QuickSort

QuickSort is a recursive sorting algorithm that was invented in 1960 by C. A. R. Hoare. It is very efficient and is often used to sort lists of items stored in arrays. QuickSort is usually written as a recursive function with three parameters that define a portion of an array to be sorted. The three parameters are an array `arr` containing a list of items, and two subscripts `start` and `end` denoting the beginning and end of the segment of `arr` that is to be sorted. Let us write `arr[start . . end]` for these three parameters. To sort the entire array, you call QuickSort with `start` set to 0 and `end` set to the size of the array minus 1.

QuickSort works as follows. If `start` is greater than or equal to `end`, then the segment of `arr` to be sorted has at most one element and is therefore already sorted. In this case, QuickSort returns immediately. Otherwise, QuickSort *partitions* `arr[start . . end]` by selecting one of the elements in `arr[start . . end]` to be a *pivot* element, and then rearranging `arr[start . . end]` so that all entries that are less than the pivot are to the left of the pivot, and all entries greater than or equal to the pivot are to the right of the pivot. In effect, the partition step rearranges `arr[start . . end]` so that it consists of a sublist 1, the pivot element, and a sublist 2 as shown in Figure 14-4.

Figure 14-4



Depending on the value selected to be the pivot element, one or the other of the two sublists may be empty. For example, if the pivot element happens to be the minimum array element, there will be no array entries less than the pivot and sublist 1 will be empty.

Notice that once the partition stage is completed and we have the situation shown in Figure 14-4, the pivot element will be in the right place. By recursively applying the QuickSort procedure to the two sublists, each of the sublists will be partitioned, putting whatever element was selected to be the pivot for that sublist in its right place. The process continues until the length of the sublists is at most one. At that point, the original array will be sorted.

Let us assume that we have a function

```
int partition(int arr[], int start, int end)
```

which when called will

1. select a pivot element from `arr[start . . end]`

2. rearrange `arr[start . . end]` into sublist 1, the pivot element, and sublist 2 (see Figure 14-4) so that the pivot element is at position `p` and sublist 1 and sublist 2 are respectively `arr[start . . p-1]` and `arr[p+1 . . end]`,
3. return the position `p` of the pivot.

We can then implement QuickSort in C++ as follows:

```
void quickSort(int arr[], int start, int end)
{
   if (start < end)
   {
      // Partition the array and get the pivot point.
      int p = partition(arr, start, end);

      // Sort the portion before the pivot point.
      quickSort(arr, start, p - 1);

      // Sort the portion after the pivot point.
      quickSort(arr, p + 1, end);
   }
}
```

Now let us consider the process of partitioning the array segment `arr[start . . end]`. The partitioning algorithm selects `arr[start]` to be the pivot element and then builds the two sublists on the left and right of the pivot element in stages. Initially, the portion of the array that has been partitioned consists of just the pivot element by itself. In effect, the initial situation will be as shown in Figure 14-4, with sublist 1 and sublist 2 being empty and all the array entries that have not yet been added to the partitioned part lying to the right of sublist 2.

The main idea is to extend the partitioned portion of the array one element at a time by considering the element *X* that is just to the right of sublist 2. If such an *X* is greater than or equal to the pivot, it is added to the end of sublist 2 by leaving it where it is and moving on to consider the next element. If *X* is less than the pivot element, it is added to the end of sublist 1 by placing it just to the left of the pivot element. One way to do this is to store *X* in a temporary location, move every element in sublist 2 up one position, move the pivot element up one position, and then drop *X* into the array position just vacated by the pivot element. This simplistic strategy moves too many array elements and does not result in an efficient algorithm. Instead, we can put *X* to the left of the pivot more efficiently by first exchanging *X* with the array item *Y* that is just to the right of the pivot element, and then exchanging *X* with the pivot element. The first exchange puts *Y*, which is greater or equal to the pivot element, at the end of sublist 2 while putting *X* in a position that is adjacent to the pivot. The second exchange then puts *X* to the left of the pivot. This is repeated until the entire list has been partitioned. The code for the partition function is

```
int partition(int arr[], int start, int end)
{
   // The pivot element is taken to be the element at
   // the start of the subrange to be partitioned.
   int pivotValue = arr[start];
   int pivotPosition = start;

   // Rearrange the rest of the array elements to
   // partition the subrange from start to end.
   for (int pos = start + 1; pos <= end; pos++)
```

```
        {
           if (arr[pos] < pivotValue)
           {
              // arr[scan] is the "current" item.
              // Swap the current item with the item to the
              // right of the pivot element.
              swap(arr[pivotPosition + 1], arr[pos]);
              // Swap the current item with the pivot element.
              swap(arr[pivotPosition], arr[pivotPosition + 1]);
              // Adjust the pivot position so it stays with the
              // pivot element.
              pivotPosition ++;
           }
        }
        return pivotPosition;
     }
```

The `swap` function used in `partition` is part of the standard template library. You need to include the `algorithm` header file to use it.

Program 14-8 demonstrates the QuickSort algorithm in action.

Program 14-8

```
1  // This program demonstrates the QuickSort algorithm.
2  #include <iostream>
3  #include <algorithm>   //needed for swap function
4  using namespace std;
5
6  // Function prototypes
7  void quickSort(int [], int, int);
8  int partition(int [], int, int);
9
10 int main()
11 {
12    // Array to be sorted.
13    const int SIZE = 10;
14    int array[SIZE] = {17, 53, 9, 2, 30, 1, 82, 64, 26, 5};
15
16    // Echo the array to be sorted.
17    for (int k = 0; k < SIZE; k++)
18       cout << array[k] << " ";
19    cout << endl;
20
21    // Sort the array using QuickSort.
22    quickSort(array, 0, SIZE-1);
23
24    // Print the sorted array.
25    for (int k = 0; k < SIZE; k++)
26       cout << array[k] << " ";
27    cout << endl;
28
29    return 0;
30 }
31
```

(program continues)

Program 14-8 *(continued)*

```
32 //************************************************
33 // quickSort uses the QuickSort algorithm to     *
34 // sort arr from arr[start] through arr[end].    *
35 //************************************************
36 void quickSort(int arr[], int start, int end)
37 {
38    if (start < end)
39    {
40       // Partition the array and get the pivot point.
41       int p = partition(arr, start, end);
42
43       // Sort the portion before the pivot point.
44       quickSort(arr, start, p - 1);
45
46       // Sort the portion after the pivot point.
47       quickSort(arr, p + 1, end);
48    }
49 }
50
51 //************************************************************
52 // partition rearranges the entries in the array arr from   *
53 // start to end so all values greater than or equal to the  *
54 // pivot are on the right of the pivot and all values less  *
55 // than are on the left of the pivot.                       *
56 //************************************************************
57 int partition(int arr[], int start, int end)
58 {
59    // The pivot element is taken to be the element at
60    // the start of the subrange to be partitioned.
61    int pivotValue = arr[start];
62    int pivotPosition = start;
63
64    // Rearrange the rest of the array elements to
65    // partition the subrange from start to end.
66    for (int pos = start + 1; pos <= end; pos++)
67    {
68       if (arr[pos] < pivotValue)
69       {
70          // arr[scan] is the "current" item.
71          // Swap the current item with the item to the
72          // right of the pivot element.
73          swap(arr[pivotPosition + 1], arr[pos]);
74          // Swap the current item with the pivot element.
75          swap(arr[pivotPosition], arr[pivotPosition + 1]);
76          // Adjust the pivot position so it stays with the
77          // pivot element.
78          pivotPosition ++;
79       }
80    }
81    return pivotPosition;
82 }
```

Program Output

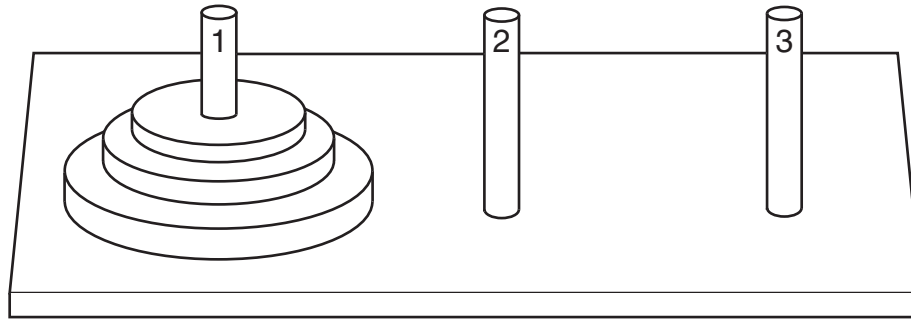
```
17 53 9 2 30 1 82 64 26 5
1 2 5 9 17 26 30 53 64 82
```

14.7 The Towers of Hanoi

CONCEPT: **There are problems that have simple recursive solutions, but which are otherwise very difficult to solve.**

The Towers of Hanoi is a game that is often used in computer science textbooks to illustrate the power of recursion. The game uses three pegs and a set of disks of different sizes with holes through their centers. The game begins with all of the disks stacked on the first of the three pegs as shown in Figure 14-5.

Figure 14-5 The pegs and disks in the Towers of Hanoi game



The object of the game is to move all the disks from the first peg to the third, while abiding by the following rules:

- All disks must rest on a peg except while being moved.
- Only one disk may be moved at a time.
- No disk may be placed on top of a smaller disk.

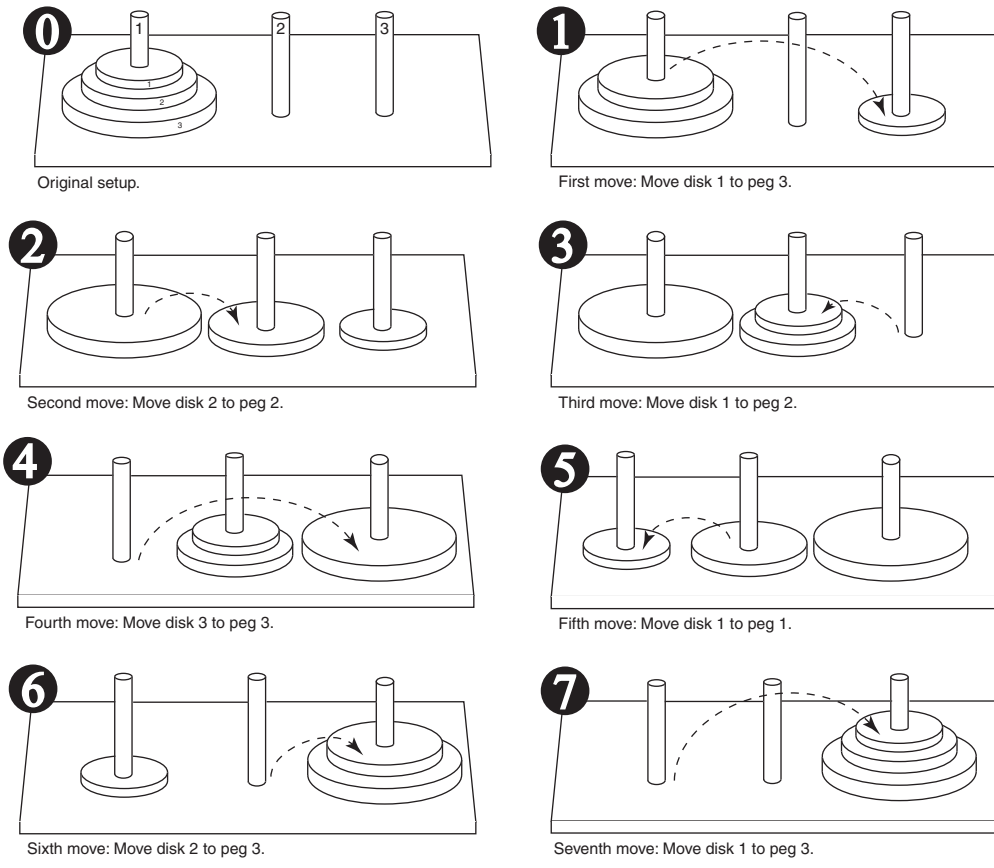
Let us look at some examples of how the game is played. The simplest case is when there is only one disk: in this case, you solve the game in one move, by moving the disk from peg 1 to peg 3.

If you have two disks, you can solve the game with three moves:

1. Move a disk from peg 1 to peg 2 (it must be the top one.)
2. Move a disk from peg 1 to peg 3.
3. Move a disk from peg 2 to peg 3.

Notice that although the object of the game is to move the disks from peg 1 to peg 3, it is necessary to use peg 2 as a temporary resting place for some of the disks. The complexity of the solution increases rapidly as the number of disks to be moved increases. Moving three disks requires seven moves as shown in Figure 14-6.

Figure 14-6



There is a charming legend associated with this game. According to this legend, there is a group of monks in a temple in Hanoi who have a set of pegs with 64 disks. The monks are busy moving the 64 disks, initially stacked on the first peg, to the third peg. When the monks complete their task the world will come to an end.

Let us now return to the problem and consider its solution in the general case when we can have any number of disks. The problem can be stated as:

Move n disks from peg 1 to peg 3 using peg 2 as a temporary peg.

It is very difficult to see how this problem can be solved using loops. Happily, it is not difficult to envision a recursive solution: If we can (recursively) move $n - 1$ disks from peg 1 to peg 2 while using peg 3 as the temporary peg, then the largest disk will be left sitting alone on peg 1. We can then move the large disk from peg 1 to peg 3 in one move. We can then (recursively) move the $n - 1$ disks from peg 2 to peg 3, this time using peg 1 as the temporary peg. This plan can be formulated in pseudocode as follows:

To move n disks from peg 1 to peg 3, using peg 2 as a temporary peg:
    If n > 0 Then
        Move n − 1 disks from peg 1 to peg 2, using peg 3 as a temporary peg.

Move a disk from peg 1 to peg 3.
Move n − 1 disks from peg 2 to peg 3, using peg 1 as a temporary peg.
End If

We will now write a function that implements this solution by printing a sequence of moves that solves the game. We will also use names rather than numbers to describe the pegs. The object of the function is then to move a stack of disks from a source peg (peg 1) to a destination peg (peg 2) using a temporary peg (peg 3). Here is the code for the function:

```
void moveDisks(int n, string source, string dest, string temp)
{
   if (n > 0)
   {
      // Move n - 1 disks from source to temp
      // using dest as the temporary peg.
      moveDisks(n - 1, source, temp, dest);

      // Move a disk from source to dest.
      cout << "Move a disk from " << source
           << " to " << dest << endl;

      // Move n - 1 disks from temp to dest
      // using source as the temporary peg.
      moveDisks(n - 1, temp, dest, source);
   }
}
```

The base case occurs when $n = 0$ and there are no disks to be moved. In this case the function call returns without doing anything. The function is demonstrated in program 14-9.

Program 14-9

```
1  // This program displays a solution to the towers of
2  // Hanoi game.
3
4  #include <iostream>
5  using namespace std;
6
7  // Function prototype
8  void moveDisks(int, string, string, string);
9
10 int main()
11 {
12    // Play the game with 3 disks.
13    moveDisks(3, "peg 1", "peg 3", "peg 2");
14    cout << "All the disks have been moved!"
15
16    return 0;
17 }
18
```

(program continues)

Program 14-9 *(continued)*

```
19 //************************************************
20 // The moveDisks function displays disk moves used  *
21 // to solve the Towers of Hanoi game.               *
22 // The parameters are:                              *
23 //    n      : The number of disks to move.         *
24 //    source : The peg to move from.                *
25 //    dest   : The peg to move to.                  *
26 //    temp   : The temporary peg.                   *
27 //************************************************
28 void
29 moveDisks(int n, string source, string dest, string temp)
30  {
31      if (n > 0)
32      {
33          // Move n - 1 disks from source to temp
34          // using dest as the temporary peg.
35          moveDisks(n - 1, source, temp, dest);
36
37          // Move a disk from source to dest.
38          cout << "Move a disk from " << source
39               << " to " << dest << endl;
40
41         // Move n - 1 disks from temp to dest
42         // using source as the temporary peg.
43          moveDisks(n - 1, temp, dest, source);
44      }
45 }
```

Program Output

```
Move a disk from peg 1 to peg 3
Move a disk from peg 1 to peg 2
Move a disk from peg 3 to peg 2
Move a disk from peg 1 to peg 3
Move a disk from peg 2 to peg 1
Move a disk from peg 2 to peg 3
Move a disk from peg 1 to peg 3
All the disks have been moved!
```



NOTE: You can find many animations on the World Wide Web and on YouTube. Type "Towers of Hanoi Animation" into your favorite search engine.

14.8 Focus on Problem Solving: *Exhaustive and Enumeration Algorithms*

CONCEPT: An enumeration algorithm is one that generates all possible combinations of items of a certain type; an exhaustive algorithm is one that searches through such a set of combinations to find the best one.

Many problems can only be solved by examining all possible combinations of items of a certain type and then choosing the best one. For example, consider the problem of making change for $1.00 using the U.S. system of coins. A few of the solutions to this problem are:

one dollar coin
two fifty-cent coins
four quarters
one fifty-cent coin and two quarters
three quarters, two dimes, and one nickel.

In fact, there are 293 ways to make change for $1.00, so we need to have a systematic method for generating them. Suppose we want to make change for a given amount using the fewest coins. A strategy for this problem that almost immediately suggests itself is to give as many of the largest coin as possible, then as many of the second largest coin as possible, and so on, until you have made change for the complete amount. It turns out that for the U.S. system of coins, this procedure, which is called the *greedy strategy,* always finds the best solution. However, the procedure does not work for other systems of coins. For example, if there are only three coin sizes,

1, 20, 25

and one has to make change for 44 cents, the greedy strategy will give one quarter and 19 pennies, for a total of 20 coins. The best solution uses six coins: two twenty-cent pieces and four pennies. In general, one would have to try all possible ways of making change to determine the best one. An algorithm that searches through all possible combinations to solve a problem is called an *exhaustive* algorithm; an algorithm that generates all possible combinations is an *enumeration* algorithm.

Recursive techniques are often useful in exhaustive and enumeration algorithms. In this section, we look at a recursive algorithm that counts the number of different ways to make change for a given amount. With some modification, the algorithm can be adapted to keep track of the different combinations, and either enumerate the list of all such combinations, or report which combination is best. Although the algorithm works for any system that includes a one-cent piece among its coins, we will assume the American system with the six coin values: 1, 5, 10, 25, 50, and 100.

The main idea is this. Suppose we want to calculate the number of ways to make change for 24 cents using coins in the set 1, 5, 10, 25, 50, 100. Since there is no way to make change for 24 cents that uses coins in the set 25, 50, 100, the largest usable coin is a dime and we can just calculate the number of ways to make change for 24 cents using coins in the set 1, 5, 10. Moreover, we cannot use more than two 10-cent pieces in making change for 24 cents, so we only need to count the number of ways to make change that use zero, one, or two 10-cent pieces and add them all together to get our answer. Table 14-1 lists these possibilities, shows how each possibility can be decomposed into a smaller problem of the same type, and shows the call to the recursive `mkChange` function that would be invoked to solve the subproblem. The parameters for the `mkChange` function will be explained shortly.

Table 14-1

number of ways to make change for 24 cents using no dimes	=	number of ways to make change for 24 cents using coins in the set 1, 5	=	`mkChange(24,1);`
number of ways to make change for 24 cents using one dime	=	number of ways to make change for 14 cents using coins in the set 1, 5	=	`mkChange(14,1);`
number of ways to make change for 24 cents using two dimes	=	number of ways to make change for 4 cents using coins in the set 1, 5	=	`mkChange(4,1);`

We are now ready to present the implementation of the algorithm. The set of possible coin values is given by an array

```
const int coinValues[] = {1, 5, 10, 25, 50, 100};
```

and the algorithm itself is embodied in the recursive function

```
int mkChange(amount, largestIndex)
```

where the first parameter is the amount to make change for, the second is the index of the largest coin in the `coinValues` array to be used in making that amount, and the integer returned is the number of combinations possible to make the specified amount of change using the specified maximum coin value. Thus the call to make change for 24 cents using coin values 1, 5 is

```
mkChange(24,1);
```

In this case, the second parameter `1` is the index of the largest coin to be used, that is the index of the nickel in the `coinValues` array. Likewise, the call to make change for 14 cents using the same coin values is

```
mkChange(14,1);
```

Program 14-10 implements this algorithm for the U.S. system of coins. It would work for any other coin system by simply changing the coin set size and the values in the `coinValues` array. The algorithm assumes that the `coinValues` array lists its values in increasing order.

Notice how the function handles the base case in lines 23–24. It returns 1 when the amount equals 0, so that when the calling function deducts coins that equal the desired amount exactly in line 34, `nWays` will be incremented by 1 in line 38. The function also returns 1 when `largestIndex` equals 0 to indicate that any amount can be composed in just 1 way using pennies (this wouldn't necessarily be true if the smallest coin were not 1).

Program 14-10

```
1 // This program demonstrates a recursive function that finds
2 // and counts all possible combinations of coin values to
3 // make a specified amount of change.
4
5 #include <iostream>
6 using namespace std;
7
```

(program continues)

Program 14-10 *(continued)*

```
 8 const int COIN_SET_SIZE = 6;
 9 const int coinValues[ ] = {1, 5, 10, 25, 50, 100};
10
11 //*********************************************************
12 // This function returns the number of ways to make change  *
13 // for an amount if we can only use coinValues in the array *
14 // positions 0 through largestIndex                         *
15 //*********************************************************
16
17 int mkChange(int amount, int largestIndex)
18 {
19    // Don't use coin values bigger than amount.
20    while(coinValues[largestIndex] > amount)
21       largestIndex--;
22
23    if (amount == 0 || largestIndex == 0)
24       return 1;
25
26    // Number of ways to make change for amount.
27    int nWays = 0;
28    // Number of coins of largest index to use.
29    int nCoins = 0 ;
30
31    while (nCoins <= amount/coinValues[largestIndex])
32    {
33       int amountLeft;
34       amountLeft = amount - nCoins * coinValues[largestIndex];
35
36       // Add the number of ways to make change with nCoins
37       // of the largest index.
38       nWays = nWays + mkChange(amountLeft, largestIndex-1);
39
40       nCoins++;
41    }
42    return nWays;
43 }
44
45 int main( )
46 {
47    // Display possible coin values.
48    cout << "Here are the valid coin values, in cents: ";
49    for (int index = 0; index < COIN_SET_SIZE; index ++)
50       cout << coinValues[index] << " ";
51    cout << endl;
52
53    // Get input from user.
54    int amount;
55    cout << "Enter the amount of cents to make change for: ";
56    cin  >> amount;
57
```

(program continues)

Program 14-10 *(continued)*

```
58    // Compute and display number of ways to make change.
59    cout << "Number of possible combinations is "
60         << mkChange(amount, COIN_SET_SIZE-1)
61         << endl;
62    return 0;
63 }
```

Program Output with Example Input Shown in Bold

```
Here are the valid coin values, in cents: 1 5 10 25 50 100
Enter (as an integer) the amount of cents to make change for: 11[Enter]
Number of possible combinations: 4
```

14.9 Focus on Software Engineering: *Recursion Versus Iteration*

CONCEPT: **Recursion and iteration are equivalent in expressive power.**

Recursion and iteration are equivalent in expressive power in the sense that whatever can be done with one can also be done with the other. In any program, any recursive function can be replaced with an equivalent function that uses loops and no recursion, and conversely, any function that uses loops can be replaced with an equivalent recursive function that uses no loops.

In general, programs that use recursion incur more overhead than equivalent programs that use iteration. This is because recursion typically involves the making of several function calls. For each such call, the machine must pass parameters to the call, keep track of the return address, create the function's local variables, and finally, destroy the local variables when the fuction returns. Current computers are fast enough that for many problems people would not notice this difference in efficiency between an algorithm that uses recursion and one that does not. In such cases, it does not make much difference whether one uses recursion or iteration.

There are, however, some recursive algorithms (like the one used to compute the Fibonacci sequence) that in the course of solving a problem recompute solutions to the same subproblems over and over again. Such algorithms tend to be extremely inefficient and should always be avoided in favor of iteration.

In general, recursion should be used whenever the problem has a natural recursive solution that does not unncessarily recompute solutions to subproblems and the equivalent solution based on iteration either is not obvious or is difficult.

14.10 Tying It All Together: *Infix and Prefix Expressions*

A binary operator is said to be *infix* if it is written between its operands, as in the expression x + y. It is said to *prefix* if it is written before its operands, as in the expression + x y. Finally, it is said to be *postfix* if it is written after its operands as in x y +. An arithmetic expression consisting of numbers, variables, and operators is called *infix* if it uses only infix operators, *prefix* if it uses only prefix operators, and *postfix* if all of its operators are postfix. Table 14-2 shows the infix, prefix, and postfix forms of five different expressions.

Table 14-2

Infix Expression	Prefix Expression	Postfix Expression
2	2	2
x	x	x
x + 2	+ x 2	x 2 +
x + 23 * y	+ x * 23 y	x 23 y * +
(x + 23) * y	* + x 23 y	x 23 + y *

An infix expression with more than one operator can be evaluated in different ways yielding different results. Consider the expression 2 + 5 *3. If we add before multiplying, the result is 21, but if we multiply and then add, we get 17. Infix expressions depend on elaborate rules of operator precedence to determine how the expression is evaluated. In addition, parentheses must sometimes be used with infix expressions to override the precedence rules.

Prefix and postfix expressions do not suffer from these drawbacks, and need neither parentheses nor rules of precedence. Instead, their operators are simply applied in the order in which they are encountered. The omission of parentheses allows prefix and postfix expressions to be stored in very compact forms, leading to savings in the amount of memory used. Because algorithms that work with prefix and postfix expressions do not need to process the parentheses or deal with precedence, they are often simpler.

Most programming languages, however, use infix expressions because that is what people are accustomed to. Many compilers and interpreters internally translate infix expressions to prefix or postfix so they can take advantage of the resulting efficiencies in storage and processing.

It is useful, when working with prefix expressions, to know they can be defined recursively:

1. A simple variable such as x, or a number such as 23, is a prefix expression.
2. Any operator followed by two prefix expressions is a prefix expression.

Based on this recursive definition, we will develop a strategy for converting a fully parenthesized infix expression to its prefix equivalent. First, note that an infix expression that involves no operators (it is an identifier or a number) is already in prefix form, in which case there is nothing to do. Otherwise, place the outermost operator of the fully parenthesized infix expression before its operands and then recursively apply this strategy to the

subexpressions (the operands of the outermost operator). Continue this until all subexpressions have been converted to prefix. Here is an example of this process:

1. Original infix expression is (x + 23)* y.
2. Place the outermost operator before its operands to give the result * (x + 23) y.
3. Recursively apply the same strategy to the inner subexpression x + 23 by placing + before x and 23 to give the result * + x 23 y.
4. Recursively apply the strategy to x, 23, and y. However, these are all base cases so they remain unchanged. The procedure terminates with the result * + x 23 y.

Having gained some practice working with prefix expressions, let's write a program that reads in prefix expressions, evaluates them, and prints the results. We assume that the prefix expressions contain no variables.

We use a recursive strategy. The base case is when the prefix expression is a single number. In that case, we just read the number and return its value. A prefix expression that is not a single number must consist of an operator followed by two prefix expressions. To evaluate such an expression, we read and store the operator, recursively evaluate the two prefix expressions to get two results, and then apply the operator to the two results. The recursive function `prefixExpr()` shown in Program 14-11 implements this strategy.

The `prefixExpr()` function uses the `peek()` member function of the `istream` class to skip whitespace and locate the beginning of the prefix expression. The `peek()` function returns the next available character from the stream without actually reading it and removing it from the stream. We use `peek()` to ensure that we do not skip a character that is part of the expression while we are skipping leading whitespace. We also use the `peek()` function to check if the first non-space character is a digit: if it is, we know the prefix expression is a number and we read it using the extraction operator in line 43:

```
exprStream >> number;
```

A non-space character that begins a prefix expression but is not a digit must be an operator. In that case, we read the character using the `get()` member function in line 52:

```
ch = exprStream.get();
```

The main function of this program just reads one line at a time, transforms the string retrieved into an `istringstream` object, and calls the `prefixExpr()` function. The user can enter multiple infix expressions with each expression being entered on its own line. The program terminates when the user enters a blank line.

Program 14-11

```
1  // This program evaluates prefix expressions.
2  #include <stdlib.h>
3  #include <string>
4  #include <sstream>
5  #include <iostream>
6  using namespace std;
7
8  int prefixExpr(istream &exprStream); //Prototype
9
10 int main()
```

(program continues)

Program 14-11 *(continued)*

```
{
    string input;
    cout << "Enter prefix expressions to evaluate.\n"
         << "Press enter after each expression,\n"
         << "and press enter on a blank line to quit.\n\n" ;
    cout << "Enter a prefix expression to evaluate: ";
    getline(cin, input);
    while (input.size() != 0)
    {
        // Convert string to istringstream
        istringstream exprStream(input);
        // Evaluate the prefix expression
        cout << prefixExpr(exprStream) << endl;
        // Get next line of input
        cout << "Enter a prefix expression to evaluate: ";
        getline(cin, input);
    }
    return 0;
}

//*************************************************************
// Takes an istream that contains a single prefix expression p  *
// and returns the integer value of p                           *
//*************************************************************
int prefixExpr(istream &exprStream)
{

   // Peek at first non-space character in prefix expression
   char ch = exprStream.peek();
   while (isspace(ch))
   {
       ch = exprStream.get();   // Read the space character
       ch = exprStream.peek();  // Peek again
   }

   if (isdigit(ch))
   {
       // The prefix expression is a  single number
       int number;
       exprStream >> number;
       return number;
   }
   else
   {
       // The prefix expression is an operator followed
       // by two prefix expressions: Compute values of
       //the prefix expressions

       // Read the operator
       ch = exprStream.get();

```

(program continues)

Program 14-11 *(continued)*

```
62          // Recursively evaluate the two subexpressions
63          int value1 = prefixExpr(exprStream);
64          int value2 = prefixExpr(exprStream);
65
66          // Apply the operator
67          switch(ch)
68          {
69              case '+': return value1 + value2;
70              case '-': return value1 - value2;
71              case '*': return value1 * value2;
72              case '/': return value1 / value2;
73              default:  cout << "Bad input expression";
74                        exit(1);
75          }
76      }
77  }
```

Program Output with Example Input Shown in Bold

```
Enter prefix expressions to evaluate.
Press enter after each expression,
and press enter on a blank line to quit.

Enter a prefix expression to evaluate: 34[Enter]
34
Enter a prefix expression to evaluate: + 23 5[Enter]
28
Enter a prefix expression to evaluate: * +23 5 2[Enter]
56
Enter a prefix expression to evaluate:[Enter]
```

Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. What type of recursive function do you think would be more difficult to debug; one that uses direct recursion, or one that uses indirect recursion? Why?
2. Which repetition approach is less efficient; a loop or a recursive function? Why?
3. When should you choose a recursive algorithm over an iterative algorithm?
4. The _____ of recursion is the number of times a function calls itself.
5. _____ recursion is when a function explicitly calls itself.
6. _____ recursion is when function A calls function B, which in turn calls function A.

Predict the Output

7. What is the output of the following programs?

A)
```
#include <iostream>
using namespace std;

int function(int);
int main()
{
   int x = 10;

   cout << function(x) << endl;
   return 0;
}

int function(int num)
{
   if (num <= 0)
      return 0;
   else
      return function(num - 1) + num;
}
```

B)
```
#include <iostream>
using namespace std;

void function(int);

int main()
{
   int x = 10;

   function(x);
   return 0;
}

void function(int num)
{
   if (num > 0)
   {
      for (int x = 0; x < num; x++)
         cout << '*';
      cout << endl;
      function(num - 1);
   }
}
```

C)
```
#include <iostream>
using namespace std;

void function(char [], int, int);

int main()
{
    char names[] = "Adam and Eve";
    function(names, 0, 13);
    return 0;
}
void function(char array[], int pos, int size)
{
   if (pos < size - 1)
   {
      function(array, pos + 1, size);
      cout << array[pos];
   }
}
```

Soft Skills

8. Programming is communication; the programmer "explains" to a computer how to carry out a task, with the explanation being the program. Can you think of any cases where communication directed to people uses direct or indirect recursion? Are there cases where such a use of recursion is indispensable?

Programming Challenges

myCodeMate

1. Iterative Factorial

Write an iterative version (using a loop instead of recursion) of the factorial function shown in this chapter. Demonstrate the use of the function in a program that prints the factorial of a number entered by the user.

2. Recursive Conversion

Convert the following function to one that uses recursion.

```
void sign(int n)
{
   while (n > 0)
   {
      cout << "No Parking\n";
      n--;
   }
}
```

Demonstrate the function with a driver program.

3. QuickSort Template

Create a template version of the `quickSort` algorithm that will work with any data type that overloads the comparison operators. Demonstrate the template with a driver function.

4. Recursive Array Sum

Write a function that accepts two arguments, an array of integers and a number indicating the number of elements in the array. The function should recursively calculate the sum of all the numbers in the array. Demonstrate the use of the function in a program that asks the user to enter an array of numbers and prints its sum.



VideoNote
Solving the Recursive Multiplication Problem

5. Recursive Multiplication

Write a recursive function that accepts two arguments into the parameters `x` and `y`. The function should return the value of `x` times `y`. Remember, multiplication can be performed as repeated addition:

7 * 4 = 4 + 4 + 4 + 4 + 4 + 4 + 4

6. Recursive Member Test

Write a recursive boolean function named `isMember`. The function should accept three parameters: an array of integers, an integer indicating the number of elements in the array, and an integer value to be searched for. The function should return `true` if the value is found in the array, or `false` if the value is not found. Demonstrate the use of the function in a program that asks the user to enter an array of numbers and a value to be searched for.

7. String Reverser

Write a recursive function that accepts a string as its argument and prints the string in reverse order. Demonstrate the function in a driver program.

8. Ackermann's Function

Ackermann's function is a recursive mathematical algorithm that can be used to test how well a computer performs recursion. Write a function `A(m, n)` that solves Ackermann's function. Use the following logic in your function:

```
If m = 0 then  return n + 1
If n = 0 then  return A(m-1, 1)
Otherwise,     return A(m-1, A(m, n-1))
```

Test your function in a driver program that displays the following values:

A(0, 0) A(0, 1) A(1, 1) A(1, 2) A(1, 3) A(2, 2) A(3, 2)

9. Prefix to Postfix

Write a program that reads prefix expressions and converts them to postfix. Each prefix expression should be entered on a separate line. The program should keep reading prefix expressions and converting them to postfix until a blank line is entered.

10. Prefix to Infix

Write a program that reads prefix expressions and converts them to infix. The infix expressions should be fully parenthesized to show the order of application of the operators. Each prefix expression should be entered on a separate line. The program should keep reading prefix expressions and converting them to infix until a blank line is entered.

This page intentionally left blank

CHAPTER 15

Polymorphism and Virtual Functions

TOPICS

15.1 Type Compatibility in Inheritance Hierarchies
15.2 Polymorphism and Virtual Member Functions
15.3 Abstract Base Classes and Pure Virtual Functions
15.4 Focus on Object-Oriented Programming: *Composition Versus Inheritance*
15.5 Secure Encryption Systems, Inc., Case Study
15.6 Tying It All Together: *Let's Move It*

15.1 Type Compatibility in Inheritance Hierarchies

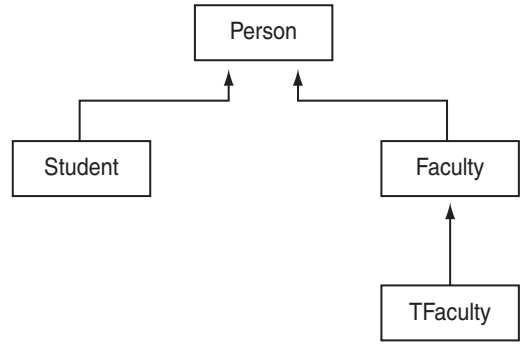
CONCEPT: **Objects of a derived class can be used wherever objects of a base class object are expected.**

Hierarchies of Inheritance

As you learned in Chapter 11, it often makes sense to create a new class based on an existing class if the new class is a special version of the existing one. The derived class can then itself serve as the base class for other classes, resulting in an *inheritance hierarchy*. For example, in Chapter 11, we used the process of inheritance to create a hierarchy of several classes: `Person`, `Student`, `Faculty`, and `TFaculty`. The relationship of inheritance is normally depicted using rectangles to represent the classes and arrows pointing from the derived class to the base class, as shown in Figure 15-1.

This hierarchy may of course be extended. For example, the `Student` class might itself be used as a base class for two other derived classes, `CStudent` and `RStudent`. These last two classes might be used to represent a type of student that commutes, and another type of student that is resident on campus.

Figure 15-1



Type Compatibility in Inheritance

Certain type compatibility relationships hold among different classes in an inheritance hierarchy. Because objects in an inheritance hierarchy are commonly accessed through pointers, we state these rules in terms of pointers:

- *A derived class pointer can always be assigned to a base class pointer*. This means that base class pointers can point to derived class objects.
- *A type cast is required to perform the opposite assignment of a base class pointer to a derived class pointer*. An error may result at run time if the base class pointer does not actually point to a derived class object.

Let us illustrate the use of these rules with a simple example.

```
class Base
{
public:
   int i;
   Base(int k){i = k;}
};
class Derived : public Base
{
public:
   double d;
   Derived(int k, double g) : Base(k){ d = g ;}
};

Base *pb = new Base(5);
Derived *pd = new Derived(6, 10.5);
```

The first rule says we can assign derived class pointers to base class pointers. Thus we can write:

```
Base *pb1 = pd;
Base *pb2 = new Derived(7, 11.5);
```

The second rule says we can assign a base class pointer to a derived class pointer if we use a type cast. Thus we can write:

```
Derived *pd1 = static_cast<Derived *>(pb1);
```

This assignment leaves `pd1`, (which is a pointer to `Derived`) pointing to a `Derived` class object, and allows subsequent accesses to `Derived` class members through `pd1` to execute correctly:

```
cout << pd1->d;
```

Consider now the scenario in which a `Base` class pointer `pb` pointing to a `Base` class object is assigned to a `Derived` class pointer using a type cast.

```
pd = static_cast<Derived *>pb;
```

The statement compiles correctly, but when it is executed, it leaves a `Derived` class pointer pointing to a `Base` class object. A subsequent access to `Derived` class members through `pd` will cause a runtime error:

```
cout << pd->d;   // Error
```

The error occurs because the `Base` class object pointed to by `pd` does not have a member `d`.

These types of compatibility rules hold even for deep inheritance hierarchies. For example, if as in Figure 15-1 `Person` is a base class for `Faculty` which is in turn a base class for `TFaculty`, then you can assign a `TFaculty` pointer to a `Person` pointer:

```
TFaculty  *tF = new TFaculty(name, disc);
Person *p ;
p = tF;
```

The assignment in the reverse direction, however, requires a type cast:

```
tF = static_cast<TFaculty *>(p);
```

To give further illustration of these concepts, consider the following program which uses modified versions of classes first encountered in Section 12 of Chapter 11.

Contents of `Inheritance4.h`

```
#include <string>
using namespace std;

enum Discipline { ARCHEOLOGY, BIOLOGY, COMPUTER_SCIENCE };
enum Classification { FRESHMAN, SOPHOMORE, JUNIOR, SENIOR };

class Person
{
protected:
  string name;
public:
  Person() { setName(""); }
  Person(string pName) { setName(pName); }
  void setName(string pName) { name = pName; }
  string getName() { return name; }
};
```

```
17
18 class Student:public Person
19 {
20 private:
21   Discipline major;
22   Person *advisor;
23 public:
24
25   Student(string sname, Discipline d, Person *adv)
26   : Person(sname)
27   {
28     major = d;
29     advisor = adv;
30   }
31   void setMajor(Discipline d) { major = d; }
32   Discipline getMajor() { return major; }
33   void setAdvisor(Person *p) { advisor = p; }
34   Person *getAdvisor() { return advisor; }
35 };
36
37 class Faculty:public Person
38 {
39 private:
40   Discipline department;
41 public:
42   Faculty(string fname, Discipline d) : Person(fname)
43   {
44      department = d;
45   }
46   void setDepartment(Discipline d) { department = d; }
47   Discipline getDepartment( ) { return department; }
48 };
49
50 class TFaculty : public Faculty
51 {
52 private:
53   string title;
54 public:
55   TFaculty(string fname, Discipline d, string title)
56   : Faculty(fname, d)
57   {
58     setTitle(title);
59   }
60   void setTitle(string title) { this->title = title; }
61
62   // Override getName()
63   string getName( )
64    {
65      return title + " " + Person::getName();
66    }
67 };
```

Here are other examples of assigning derived class pointers to base classes:

```
Person *ptp;
TFaculty *ptf;
// Pointer to Derived class is assigned to Base class
// pointer.
ptp = new TFaculty("Indiana Jones", ARCHEOLOGY, "Dr.");
// Assigning a base class pointer to a derived class
// pointer requires a typecast
ptf = static_cast<TFaculty *>(ptp);
```

In this section of code, the `new` operator returns a pointer to the derived class `TFaculty`, which is assigned to the base class pointer `ptp`. The base class pointer `ptp` is then assigned to `ptf` using a type cast.

These type compatibility rules apply in two other cases. A function that is declared as taking a pointer to a base class will accept a pointer to a derived class object as an actual parameter. Also, a function that declares a return type of a pointer to a particular class *C* may actually return a pointer to on object of a class derived from *C*.

Using Type Casts with Base Class Pointers

We have seen that a pointer to a particular class *C* may actually be pointing an object of a class derived from *C*. In this case, the class type of the pointer will be different from the class type of the object, and C++ defaults to using the class of the pointer to determine access to the members of the object. As an example, consider the statement

```
Person *pPerson = new Faculty("Donald Knuth", COMPUTER_SCIENCE);
```

Even though this assignment is legal, it does not make the pointer `pPerson` aware of additional members of the `Faculty` class that are not in the `Person` class. Consequently, an attempt to access members of the `Faculty` class through `pPerson`, such as

```
pPerson->setDepartment(BIOLOGY); //  compiler error!
```

is rejected by the compiler. If we do know that `pPerson` actually points to a `Faculty` object, we can use a type cast to get the compiler to accept the statement:

```
static_cast<Faculty *>(pPerson)->setDepartment(BIOLOGY);
```

The type cast informs the compiler that `pPerson` is actually pointing to a `Faculty` object derived from the `Person` base class. Alternatively, we can first cast `pPerson` to a pointer to `Faculty`, and then use the new pointer:

```
Faculty *pFaculty = static_cast<Faculty *>(pPerson);
pFaculty->setDepartment(BIOLOGY);
```

In general, a pointer to a base class that actually points to a derived class object must first be appropriately cast before the additional features of the derived class can be used.

Recall from Chapter 11 that a derived class may override member functions that are defined in its base class. When a pointer to a base class is being used to access a member function that has been overridden by the derived class, the default C++ behavior is to use the version of the function that is defined in the class of the pointer rather than in the class of the object. For example, the code

```
Person *pP = new TFaculty("Indiana Jones", ARCHEOLOGY);
```

sets pP, which is a pointer to the base class `Person`, to point to a `TFaculty` object. Note that `TFaculty` overrides the `getName` function defined in `Person`. In executing the statement

```
cout << pP->getName();
```

the compiler is not aware that the actual class type of the object is `TFaculty`. The compiler sees the class type of the pointer and assumes that the class type of the object is the same as that of the pointer. Therefore, it calls the version of `getName` defined in the `Person` class. Program 15-1 illustrates these concepts.

Program 15-1

```
// This program demonstrates type compatibility within
// an inheritance hierarchy.
#include "inheritance4.h"
#include <iostream>
using namespace std;

int main()
{
   Person *pp;
   Faculty *pf;
   TFaculty *ptf;
   ptf = new TFaculty("Indiana Jones", ARCHEOLOGY, "Dr.");

   // Calling getName through a pointer to TFaculty uses
   // the version of getName in TFaculty.
   cout << "Get name through a pointer to TFaculty: ";
   cout << ptf->getName() << endl;

   // Assignment of derived to base needs no cast.
   pf = ptf;

   // Calling getName through a pointer to Faculty uses the
   // version of getName in Faculty.
   cout << "Get name through a pointer to Faculty: ";
   cout << pf->getName() << endl;

   // Assignment of derived to base needs no cast.
   pp = ptf;

   // Derived class members can be accessed using a cast.
   cout << "Get name through a cast to pointer to TFaculty: ";
   cout << static_cast<TFaculty *>(pp)->getName() << endl;

   // Assigment from base to derived needs a cast.
   TFaculty *ptf1;
   ptf1 = static_cast<TFaculty *>(pp);

   // Access getName through a pointer to TFaculty
   cout << "Get name through a pointer to TFaculty: ";
   cout << ptf1->getName();

   return 0;
}
```

(program continues)

Program 15-1 *(continued)*

Program Output

```
Get name through a pointer to TFaculty: Dr. Indiana Jones
Get name through a pointer to Faculty: Indiana Jones
Get name through a cast to pointer to TFaculty: Dr. Indiana Jones
Get name through a pointer to TFaculty: Dr. Indiana Jones
```

15.2 Polymorphism and Virtual Member Functions

CONCEPT: **Virtual functions allow the most specific version of a member function in an inheritance hierarchy to be selected for execution. Virtual functions make polymorphism possible.**



VideoNote
Polymorphism

A piece of code is said to be *polymorphic* if executing the code with different types of data produces different behavior. For example, a function would be called polymorphic if it executes differently when it is passed different types of parameters.

To illustrate polymorphism, consider the following program. The program creates an array of (pointers to) `Person` objects of type `Student`, `Faculty`, and `TFaculty`. It then prints the names in all the objects using the same code. Because an array can only hold elements of one type, we must use an array of pointers to the base class.

Program 15-2

```
1  // This exhibits the default non-polymorphic behavior of C++.
2  #include "inheritance4.h"
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // Create an array of pointers to Person objects.
9     const int NUM_PEOPLE = 5;
10    Person *arr[NUM_PEOPLE] =
11     {
12       new TFaculty("Indiana Jones", ARCHEOLOGY, "Dr."),
13       new Student("Thomas Cruise", COMPUTER_SCIENCE, NULL),
14       new Faculty("James Stock", BIOLOGY),
15       new TFaculty("Sharon Rock", BIOLOGY, "Professor"),
16       new TFaculty("Nicole Eweman", ARCHEOLOGY, "Dr.")
17     };
18    // Print the names of the Person objects.
19    for (int k = 0; k < NUM_PEOPLE; k++)
20    {
21       cout << arr[k]->getName() << endl;
22    }
23    return 0;
24 }
```

(program continues)

Program 15-2 *(continued)*

Program Output

```
Indiana Jones
Thomas Cruise
James Stock
Sharon Rock
Nicole Eweman
```

Notice that the program calls the `Person` version of the `getName` function for all objects in the array, even though the `TFaculty` objects have their own, more specialized version. This code is obviously not polymorphic, for it executes the same member function for each object, regardless of its type. In other words, it does not behave differently for different types of objects.

To better understand what is happening, we need to take a closer look at each of the five calls

```
arr[k]->getName()
```

used to retrieve the name to be printed. In each of these calls, a pointer `arr[k]` to the base class `Person` is used to invoke the `getName` function in objects of different derived classes. Some of these classes, like `TFaculty`, override `getName` to provide a more specialized version of that function. When `arr[k]` is pointing to a `TFaculty` object, the compiler must choose between the `getName` defined in `Person`, the class of the pointer, and the `getName` defined in `TFaculty`, the class that the object actually belongs to. The default C++ behavior is to use the class type of the pointer rather than that of the object to determine which version of an overridden function to call.

The scenario of invoking a member function of a derived class object through a base class pointer is a common occurrence in object-oriented programming. Let us say that we have a base class `B` with a member function `mfun()` and a base class pointer `ptr` that is pointing to an object of a derived class `D`.

```
class B
{
public:
   void mfun()
   {
      cout << "Base class version";
   }
};
class D : public B
{
public:
   void mfun()
   {
      cout << "Derived class version";
   }
};
Base *ptr = new D();
```

We want to tell the compiler that whenever we write

```
ptr->mfun()
```

the compiler should select the more specialized version of `mfun()` in the derived class. We can do this in C++ by declaring `mfun()` to be a *virtual* function in the base class. Virtual functions are used in C++ to support polymorphic behavior. Thus, to achieve polymorphic behavior for `mfun()` in the class B and all of its derived classes, we must modify the definition of B as follows:

```
class B
{
public:
   virtual void mfun()
   {
      cout << "Base class version";
   }
};
```

The virtual characteristic is inherited: that is, if a member function of a derived class overrides a virtual function in the base class, then that member function is automatically virtual itself. Thus the declaration of `mfun` as virtual in `B` makes `mfun` virtual in `D` and in all classes derived from `D`.

Although it is not necessary, many programmers tag all virtual functions with the key word `virtual` to make it easer to identify them. This is good practice, and accordingly, the definition of `D` should be written as follows:

```
class D : public B
{
public:
   virtual void mfun()
   {
      cout << "Derived class version";
   }
};
```

The following program is a modification of Program 15-2. In it, the `getName` function of the `Person` class has been declared virtual. It includes the `inheritance5.h` file, which is the just the `inheritance4.h` file modified to make the `getName` function in the `Person` class virtual.

Program 15-3

Contents of `Inheritance5.h`

```
1 #include <string>
2 using namespace std;
3
4 enum Discipline { ARCHEOLOGY, BIOLOGY, COMPUTER_SCIENCE };
5 enum Classification { FRESHMAN, SOPHOMORE, JUNIOR, SENIOR };
6
7 // The Person class is modified to make getName
8 // a virtual function.
```

(program continues)

Program 15-3 *(continued)*

```
 9 class Person{
10 protected:
11   string name;
12 public:
13   Person() { setName(""); }
14   Person(string pName) { setName(pName); }
15   void setName(string pName) { name = pName; }
16
17   // Virtual function.
18   virtual string getName() { return name; }
19 };
20
21 class Student:public Person
22 {
23 private:
24   Discipline major;
25   Person *advisor;
26 public:
27   Student(string sname, Discipline d, Person *adv)
28   : Person(sname)
29   {
30     major = d;
31     advisor = adv;
32   }
33   void setMajor(Discipline d) { major = d; }
34   Discipline getMajor() { return major; }
35   void setAdvisor(Person *p) { advisor = p; }
36   Person *getAdvisor() { return advisor; }
37 };
38
39 class Faculty:public Person
40 {
41 private:
42   Discipline department;
43 public:
44   Faculty(string fname, Discipline d) : Person(fname)
45   {
46      department = d;
47   }
48   void setDepartment(Discipline d) { department = d; }
49   Discipline getDepartment( ) { return department; }
50 };
51
52 class TFaculty : public Faculty
53 {
54 private:
55   string title;
56 public:
57   TFaculty(string fname, Discipline d, string title)
58   : Faculty(fname, d)
```

(program continues)

Program 15-3 *(continued)*

```
59   {
60     setTitle(title);
61   }
62
63   void setTitle(string title) { this->title = title; }
64
65   // Virtual function
66   virtual string getName( )
67   {
68     return title + " " + Person::getName();
69   }
70 };
71
```

Contents of Main Program, pr15-03.cpp

```
1  // This demonstrates the polymorphic behavior
2  // of classes with virtual functions.
3  #include "inheritance5.h"
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     // Create an array of Person objects.
10    const int NUM_PEOPLE = 5;
11    Person *arr[NUM_PEOPLE] =
12        {
13          new TFaculty("Indiana Jones", ARCHEOLOGY, "Dr."),
14          new Student("Thomas Cruise", COMPUTER_SCIENCE, NULL),
15          new Faculty("James Stock", BIOLOGY),
16          new TFaculty("Sharon Rock", BIOLOGY, "Professor"),
17          new TFaculty("Nicole Eweman", ARCHEOLOGY, "Dr.")
18        };
19    // Print the names of the Person objects.
20    for (int k = 0; k < NUM_PEOPLE; k++)
21    {
22      cout << arr[k]->getName() << endl;
23    }
24    return 0;
25 }
```

Program Output

```
Dr. Indiana Jones
Thomas Cruise
James Stock
Professor Sharon Rock
Dr. Nicole Eweman
```

Dynamic and Static Binding

The compiler is said to *bind* the name of a function when it selects the code that should be executed when the function name is invoked. In other words, the compiler binds the name to a function definition when the function is called.

Static binding happens at compile time, and binds the name to a fixed function definition, which is then executed each time the name is invoked. For example, in Program 15-2 of the previous section, the compiler used static binding to bind `getName` in the statement

```
for (int k = 0; k < NUM_PEOPLE; k++)
{
   cout << arr[k]->getName() << endl;
}
```

to the definition of `getName` in the `Person` class.

In static binding, the compiler uses type information available at compile time. If the code is operating on objects of different classes within an inheritance hierarchy, the only type information available to the compiler will be the base class pointer type used to access all the objects. Consequently, static binding will always use the base class version of a member function.

In contrast, *dynamic binding* occurs at run time. Dynamic binding works only if the compiler can determine at run time the exact class that a subclass object belongs to. The compiler then uses this run-time type information to call the version of the function defined in that class. To make dynamic binding possible, the compiler stores run-time type information in every object of a class with a virtual function. Dynamic binding always uses the version of the member function in the actual class of the object, regardless of the class of the pointer used to access the object.

More information on dynamic binding and run-time type information can be found in Appendix I on the student CD that comes with this book.

15.3 Abstract Base Classes and Pure Virtual Functions

CONCEPT: Abstract classes and pure virtual functions can be used to define an interface that must be implemented by derived classes.

It is often convenient to have a base class for an inheritance hierarchy that defines a member function that must be implemented in every derived class, but which cannot be implemented by the base class itself because the details needed for a reasonable implementation can only be found in the derived classes. If this is the case, the C++ language permits the programmer to declare the function a *pure virtual function*, that is, a member function for which the class provides no implementation. The C++ way of declaring a pure virtual function is to put the expression `= 0` in the class declaration where the body of the function would otherwise have gone. For example, if a member function `void draw()` is being declared pure virtual, then its declaration in its class looks like

```
void draw() = 0;
```

A pure virtual function is sometimes called an *abstract* function, and a class with at least one pure virtual function is called an *abstract* class. The C++ compiler will not allow you to instantiate an abstract class. Abstract classes can only be subclassed: that is, you can only use them as base classes from which to derive other classes.

A class derived from an abstract class inherits all functions in the base class, and will itself be an abstract class unless it overrides all the abstract functions it inherits. The usefulness of abstract classes lies in the fact that they define an interface that will then have to be supported by objects of all classes derived from it.

You can think of an abstract class as a class which has no instances other than those that belong to some subclass. There are many examples of abstract classes in real life. For example, in the animal kingdom, the class "Animal" of all animals is an abstract class. There are instances of animals that do not actually belong to some subclass. There are animals that are dogs, or chickens, or foxes, but there no animals that are just animals.

Consider a graphics system that consists of a collection of shapes that must be drawn at certain locations on the screen. Each shape object would have some member variables to keep track of its position, and a member function for drawing the shape at the right position. The different shapes supported by the system might include rectangles, hexagons and others. Because a rectangle is a shape, and a hexagon is a shape, it makes sense to have a `Shape` class and have both `Rectangle` and `Hexagon` be classes derived from `Shape`. The `Shape` class will have a member function `setPosition` for setting the position of the shape, as well as a member function `draw` for drawing the shape. However, because `Shape` is an abstract class (there is no shape that is just a "shape," it must be a rectangle, a hexagon, a triangle, or other) the logic for drawing a particular shape must be delegated to an appropriate subclass. Thus the `draw()` function cannot have an implementation in the `Shape` class, and must be made a pure virtual function.

Program 15-4 shows a `Shape` class with two derived classes: `Rectangle` and `Hexagon`. The class declares a pure virtual function `draw()` that is implemented by its two subclasses. The `main` function maintains a collection of `Shape` objects using an array of pointers.

Program 15-4

```
1  // This program demonstrates abstract base
2  // classes and pure virtual functions.
3  #include <iostream>
4  using namespace std;
5
6  class Shape
7  {
8  protected:
9     int posX, posY;
10 public:
11    virtual void draw() = 0;
12    void setPosition(int pX, int pY)
13    {
14       posX = pX;
15       posY = pY;
16    }
17 };
18
19 class Rectangle : public Shape
20 {
```

(program continues)

Program 15-4 *(continued)*

```
21 public:
22   virtual void draw()
23   {
24     cout << "Drawing rectangle at "  << posX << "  "
25          <<  posY << endl;
26   }
27 };
28
29 class Hexagon : public Shape
30 {
31 public:
32   virtual void draw()
33   {
34      cout << "Drawing hexagon at "  << posX << "  "
35           <<  posY << endl;
36   }
37 };
38
39 int main()
40  {
41    // Create array of pointers to Shapes of various types.
42    const int NUM_SHAPES = 3;
43    Shape * shapeArray[] = { new Hexagon(),
44                             new Rectangle(),
45                             new Hexagon()
46                           };
47    // Set positions of all the shapes.
48    int posX = 5, posY = 15;
49    for (int k = 0; k < NUM_SHAPES; k++)
50    {
51      shapeArray[k]->setPosition(posX, posY);
52      posX += 10;
53      posY += 10;
54    };
55
56    // Draw all the shapes at their positions.
57    for (int j = 0; j < NUM_SHAPES; j++)
58    {
59      shapeArray[j]->draw();
60    }
61    return 0;
62  }
```

Program Output

```
Drawing hexagon at 5  15
Drawing rectangle at 15  25
Drawing hexagon at 25  35
```

Program 15-4 affords another demonstration of dynamic binding and polymorphism. Consider in particular the statement

```
shapeArray[j]->draw();
```

which is executed a number of different times in the loop

```
for (int j = 0; j < NUM_SHAPES; j++)
{
   shapeArray[j]->draw();
}
```

The first time the statement is executed, it invokes the `draw` function on a hexagon object, while the second time, it invokes the `draw` function on a rectangle object. Because the two `draw` functions are in different classes, they produce different behavior.

Remember the following points about abstract base classes and pure virtual functions:

- When a class contains a pure virtual function, it is an abstract base class.
- Abstract base classes cannot be instantiated.
- Pure virtual functions are declared with the `= 0` notation, and have no body, or definition.
- Pure virtual functions *must* be overridden in derived classes that need to be instantiated.



Checkpoint

15.1 Explain the difference between static binding and dynamic binding.

15.2 Are virtual functions statically bound or dynamically bound?

15.3 What will the following program display?

```
#include <iostream>
using namespace std;
class First
{
   protected:
      int a;
   public:
      First(int x = 1) { a = x; }
      int getVal() { return a; }
};
class Second : public First
{
   private:
      int b;
   public:
      Second(int y = 5) { b = y; }
      int getVal() { return b; }
};
int main()
{
   First object1;
   Second object2;
   cout << object1.getVal() << endl;
   cout << object2.getVal() << endl;
   return 0;
}
```

15.4 What will the following program display?

```
#include <iostream>
using namespace std;
class First
```

```
{
   protected:
      int a;
   public:
      First(int x = 1) { a = x; }
      void twist() { a *= 2; }
      int getVal() { twist(); return a; }
};
class Second : public First
{
   private:
      int b;
   public:
      Second(int y = 5) { b = y; }
      void twist() { b *= 10; }
};
int main()
{
   First object1;
   Second object2;
   cout << object1.getVal() << endl;
   cout << object2.getVal() << endl;
   return 0;
}
```

15.5 What will the following program display?

```
#include <iostream>
using namespace std;
class First
{
   protected:
      int a;
   public:
      First(int x = 1) { a = x; }
      virtual void twist() { a *= 2; }
      int getVal() { twist(); return a; }
};
class Second : public First
{
   private:
      int b;
   public:
      Second(int y = 5) { b = y; }
      virtual void twist() { b *= 10; }
};
int main()
{
   First object1;
   Second object2;
   cout << object1.getVal() << endl;
   cout << object2.getVal() << endl;
   return 0;
}
```

15.6 What will the following program display?

```
#include <iostream>
using namespace std;
class Base
{
   protected:
      int baseVar;
   public:
      Base(int val = 2) { baseVar = val; }
      int getVar() { return baseVar; }
};
class Derived : public Base
{
   private:
      int deriVar;
   public:
      Derived(int val = 100) { deriVar = val; }
      int getVar() { return deriVar; }
};

int main()
{
   Base *optr;
   Derived object;

   optr = &object;
   cout << optr->getVar() << endl;
   return 0;
}
```

15.7 How can you tell from looking at a class declaration that a virtual member function is pure?

15.8 What makes an abstract class different from other classes?

15.9 Examine the following classes. The table lists the variables that are members of the `Third` class (some are inherited). Complete the table by filling in the access specification each member will have in the `Third` class. Write "inaccessible" if a member is inaccessible to the `Third` class.

```
class First
{
   private:
      int a;
   protected:
      double b;
   public:
      long c;
};
```

```
class Second : protected First
{
   private:
      int d;
   protected:
      double e;
   public:
      long f;
};

class Third : public Second
{
   private:
      int g;
   protected:
      double h;
   public:
      long i;
}
```

Member Variable	Access Specification in `Third` class
a	
b	
c	
d	
e	
f	
g	
h	
i	

15.4 Focus on Object-Oriented Programming: *Composition Versus Inheritance*

CONCEPT: **Inheritance should model an "is a" relation, rather than a "has a" relation, between the derived and base classes.**



VideoNote
Composition Versus Inheritance

Class inheritance in an object-oriented language should be used to model the fact that the type of the derived class is a special case of the type of the base class. Actually, a class can be considered to be the *set* of all objects that can be created from it. Because the derived

class is a special case of the base class, the set of objects that correspond to the derived class will be a *subset* of the set of objects that correspond to the base class. Thus, every object of the derived class is also an object of the base class. In other words, each derived class object *is a* base class object.

Class composition occurs whenever a class contains an object of another class as one of its member variables. Composition was discussed in Chapter 11, where it was pointed out that composition models a *has-a* relation between classes.

Because a derived class inherits all the members of its base class, a derived class effectively contains an object of its base class. Because of this, it is possible to use inheritance where a correct design would call for composition. As an example, consider a program that needs to represent data for a person, say the person's name and street address. The street address might consist of two lines:

123 Main Street
Hometown, 12345

Now suppose we had a class for representing a street address:

```
class StreetAddress
{
   private:
      string line1, line2;
   public:
      void setLine1(string);
      void setLine2(string);
      string getLine1();
      string getLine2();
};
```

Because a person's data has a name and a street address, the proper formulation of a class to represent a person's data would use composition in the following way:

```
class PersonData
{
   private:
      string name;
      StreetAddress address;
   public:
      ...
};
```

We have left off the rest of the class declaration for `PersonData` because we don't need it for our purposes.

It is possible to define this class using inheritance instead of composition. For example, we could define a class `PersonData1` as follows:

```
class PersonData1:public StreetAddress
{
   private:
      string name;
   public:
      ...
};
```

While this new definition would compile correctly, it is conceptually the wrong thing to do because it regards a person's data as a special kind of `StreetAddress`, which it is not. This type of conceptual error in design can result in a program that is confusing to understand and difficult to maintain. It is a good design practice to prefer composition to inheritance whenever possible. One reason is that inheritance breaks the encapsulation of the base class by exposing the base class's protected members to the methods of the derived class.

Let us next consider an example where it makes sense to use inheritance rather than composition. Suppose that we have a class `Dog` that represents the set of all dogs. Assuming that each `Dog` object has a member variable `weight` of type `double` and a member function `void bark()`, we might have the following class:

```
class Dog
{
   protected:
      double weight;
   public:
      Dog(double w)
         { weight = w; }
      virtual void bark()
         {
             cout << "I am dog weighing "
                  << weight << " pounds." << endl;
         }
};
```

The class also has a constructor to allow `Dog` objects to be initialized. Note that we have declared the `bark()` member function as virtual to allow it to be overridden in a derived class.

Suppose that we need to have a class that represents the set of all sheep dogs. Since every sheep dog is also a dog, it makes sense to derive the new `SheepDog` class from the `Dog` class. That way, a `SheepDog` object will inherit every member of the `Dog` class. In addition to having every characteristic that every dog has, a sheep dog can be expected to have other characteristics peculiar to sheep dogs, for example, an integer member `numberSheep` that indicates the maximum number of sheep the dog is trained to herd. In addition, a sheep dog might have a way of barking different from that of a generic dog, perhaps one adapted to the tending of sheep. This is accounted for by overriding the `bark()` member function of the `Dog` class.

```
class SheepDog:public Dog
{
   private:
      int numberSheep;
   public:
      SheepDog(double w, int nSheep) : Dog(w)
      {
         numberSheep = nSheep;
      }
      void bark()
      {
         cout << "I am a sheepdog weighing "
              << weight << " pounds \n   and guarding "
              << numberSheep << " sheep." << endl;
      }
};
```

To demonstrate this class, we will set up an array of dogs with some of the dogs in the array being sheep dogs. To get around the fact that an array cannot hold two different types, we will use an array of pointers to `Dog`. Recall from Section 15.1 that a pointer to a base class (in this case, `Dog`) can point to any derived class object (in this case, `SheepDog`). We can therefore create an array of pointers to `Dog` and have some of those pointers point to `Dog` objects while others point to `SheepDog` objects:

```
Dog *kennel[3] = { new Dog(40.5),
                   new SheepDog(45.3, 50),
                   new Dog(24.7)
                 };
```

Finally, we can use a loop to call the `bark()`member function of each `Dog` object in the array.

```
for (int k = 0; k < 3; k++)
   {
      cout << k+1 << ": ";
      kennel[k]->bark();
   }
```

Because of polymorphism, and because the `bark()` function was declared virtual, the same line of code inside the loop will call the original `bark()` function for a regular dog, but will call the specialized `bark()` function for a sheep dog. The complete program is given in Program 15-5.

Program 15-5

```
1  // This program demonstrates the is-a
2  // relation in inheritance.
3  #include <iostream>
4  using namespace std;
5
6  // Base class.
7  class Dog
8  {
9    protected:
10     double weight;
11   public:
12     Dog(double w)
13      { weight = w; }
14     virtual void bark( )
15      {
16        cout << "I am a dog weighing "
17             << weight << " pounds." << endl;
18      }
19 };
20
```

(program continues)

Program 15-5 *(continued)*

```
21 //A SheepDog is a special type of Dog.
22 class SheepDog:public Dog
23 {
24     int numberSheep;
25   public:
26     SheepDog(double w, int nSheep) : Dog(w)
27     {
28       numberSheep = nSheep;
29     }
30     void bark( )
31     {
32       cout << "I am a sheepdog weighing "
33            << weight << " pounds \n   and guarding "
34            << numberSheep << " sheep." << endl;
35     }
36 };
37
38 int main( )
39 {
40    //Create an array of dogs.
41    const int NUM_DOGS = 3;
42    Dog *kennel[ ] = { new Dog(40.5),
43                       new SheepDog(45.3, 50),
44                       new Dog(24.7)
45                     };
46
47    //Walk by each kennel and make the dog bark.
48    for (int k = 0; k < NUM_DOGS; k++)
49    {
50      cout << k+1 << ": ";
51      kennel[k]->bark( );
52    }
53    return 0;
54 }
```

Program Output

```
1: I am a dog weighing 40.5 pounds.
2: I am a sheepdog weighing 45.3 pounds
   and guarding 50 sheep.
3: I am a dog weighing 24.7 pounds.
```

Inheritance is a better choice than composition in this example, since to use composition would be tantamount to saying that a sheep dog *has a* dog, instead of saying that a sheep dog *is a* dog.

There is a third relationship between classes that some authors talk about: the *uses implementation of* relation. Basically, one class uses the implementation of a second class if it calls a member function of an object of the second class.

How can you know when to use inheritance and when to use composition? Suppose that you have an existing class `C1` and you need to write a definition for another class `C2` that

will need the services of an associated `C1` object. Should you derive `C2` from `C1`, or should you give `C2` a member variable of type `C1`? In general, you should prefer composition to inheritance. To help determine if inheritance may be appropriate, you might ask the following questions:

- Is it natural to think of a `C2` object as a special type of `C1` object? If so, then you should use inheritance.
- Will objects of class `C2` need to be used in places where objects of class `C1` are used? For example, will they need to be passed to functions that take reference parameters of type `C1`, or pointers to `C1`? If so, then you should make `C2` a derived class of `C1`.

15.5 Secure Encryption Systems, Inc., Case Study

Secure Encryption Systems is a recently founded consulting company that advises business and corporations on how to protect their data from unauthorized access. The company is interested in developing a framework that enables the rapid evaluation and testing of different encryption and decryption algorithms to determine their effectiveness and the level of security they offer.

In this section, we will consider the use of virtual functions and abstract classes to build application frameworks. An *application framework* can be regarded as an application in skeletal form: it only needs the user to specify the definition of a few functions to transform the framework into a useful application.

Understanding the Problem

In Chapter 9's *Tying It All Together* section you were introduced to the idea of encoding a message so that it could only be read by someone possessing the right information to decode it. More formally, *encryption* is the process of transforming a message, called *plain text*, into *cipher text*, a form that disguises its true meaning. The encryption algorithm used in Chapter 9 involved substituting a new character for each original character. There are many other methods of encrypting text. For example, a simple encryption algorithm might remove all punctuation and spacing from a message and then mix up all the letters in some predetermined way. The message can later be "unmixed" to allow it to be read. For example, the plain text message

```
attack at dawn
```

might be transformed into the cipher text

```
aadakntwctta
```

The "mixing" scheme is an example of an *encryption key*, while the "unmixing" scheme would be a corresponding *decryption key*. *Decryption* is the process of reversing the encryption transformation that has been performed on a message. Decryption is applied to cipher text to yield the original plain text.

A Simple Encryption / Decryption Framework

To provide a framework for testing encryption and decryption algorithms, we will implement a class that provides all the functionality needed to test such an algorithm, but leaves the function that is used to transform the letters unspecified. The framework will be realized as a class, and the transformation function will be a pure virtual member function of the class. Specific encryption algorithms can then be easily tested by forming a derived class of the framework class and overriding the virtual transformation function.

We use a really simple character transformation algorithm: it just shifts up the character by one in the ASCII code: it does not even wrap around to the beginning of the alphabet when it shifts the letters 'z' or 'Z'. The major part of the program is the `Encryption` class:

```
class Encryption
{
 protected:
   ifstream inFile;
   ofstream outFile;
 public:
   Encryption(char *inFileName, char *outFileName);
   ~Encryption();
   // Pure virtual function.
   virtual char transform(char ch) = 0;
   // Do the actual work.
   void encrypt();
};
```

This class contains the file objects that will be used to access the input and output files. The constructor is passed the names of the two files and does the work of opening the files. The destructor closes the files. The `encrypt` function will read characters from the input file, call the virtual function `transform` to transform the single character, and write the character to the output file.

Because the `transform` function is pure virtual, the `Encryption` class is abstract and cannot be instantiated. All a subclass of `Encryption` needs to do is implement a suitable `transform` function and pass the file names as parameters to its base class constructor. The complete program follows.

Application frameworks are used in many areas of software development to simplify the creation of software. Most application framework rely heavily on virtual functions and abstract classes.

Program 15-6

```
1 // This program demonstrates an application
2 // of pure virtual functions.
3 #include <iostream>
4 #include <fstream>
5 using namespace std;
6
```

(program continues)

Program 15-6 *(continued)*

```
7  class Encryption
8  {
9   protected:
10    ifstream inFile;
11    ofstream outFile;
12   public:
13    Encryption(char *inFileName, char *outFileName);
14    ~Encryption();
15    // Pure virtual function.
16    virtual char transform(char ch) = 0;
17    // Do the actual work.
18    void encrypt();
19 };
20
21 //************************************************
22 // Constructor opens the input and output file.    *
23 //************************************************
24 Encryption::Encryption(char *inFileName, char *outFileName)
25 {
26    inFile.open(inFileName);
27    outFile.open(outFileName);
28    if (!inFile)
29      {
30        cout <<  "The file " << inFileName
31             << " cannot be opened.";
32        exit(1);
33      }
34    if (!outFile)
35     {
36       cout << "The file " << outFileName
37            << " cannot be opened.";
38       exit(1);
39     }
40 }
41
42 //************************************************
43 //Destructor closes files.                        *
44 //************************************************
45 Encryption::~Encryption()
46 {
47    inFile.close();
48    outFile.close();
49 }
50
```

(program continues)

Program 15-6 *(continued)*

```
51 //*************************************************
52 //Encrypt function uses the virtual transform         *
53 //member function to transform individual characters. *
54 //*************************************************
55 void Encryption::encrypt()
56 {
57    char ch;
58    char transCh;
59    inFile.get(ch);
60    while (!inFile.fail())
61      {
62        transCh = transform(ch);
63        outFile.put(transCh);
64        inFile.get(ch);
65      }
66 }
67
68 // The subclass simply overides the virtual
69 // transformation function.
70 class SimpleEncryption : public Encryption
71 {
72 public:
73    char transform(char ch)
74    {
75      return ch + 1;
76    }
77    SimpleEncryption(char *inFileName, char *outFileName)
78    : Encryption(inFileName, outFileName)
79    {
80    }
81 };
82
83 int main()
84 {
85   char inFileName[80], outFileName[80];
86   cout << "Enter name of file to encrypt: ";
87   cin >> inFileName;
88   cout << "Enter name of file to receive "
89        << "the encrypted text: ";
90   cin >> outFileName;
91   SimpleEncryption obfuscate(inFileName, outFileName);
92   obfuscate.encrypt();
93   return 0;
94 }
```

15.6 Tying It All Together: *Let's Move It*

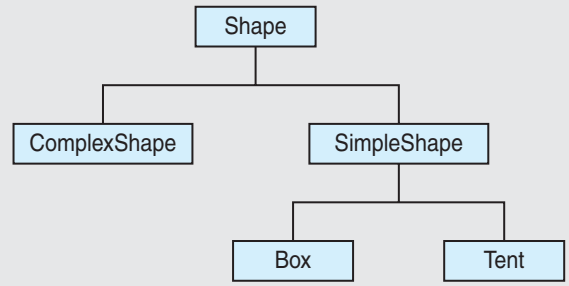
Video game programmers often have to maintain a collection of figures that are simultaneously moving in various directions on the screen. Let's devise a solution to a simplifed version of this problem. We will maintain a collection of geometric shapes and simultaneously animate those shapes. The functions used to directly access the screen and manage the timer are peculiar to Microsoft Windows, but the principles used are very general and are applicable to all operating systems.

We begin with a class that represents a shape that is able to move in any of eight different directions, with each direction being specified by a pair of integer (X, Y) coordinates. Upward or downward motion is indicated by a Y component of ±1, and likewise, motion in a left or rightward direction is indicated by an X component of ±1. A value of 0 for an X or Y coordinate indicates lack of motion in that direction. Thus, a value of (0, 1) for (X, Y) indicates motion straight up, a value of (–1, 0) indicates motion to the left, and (1, 1) is motion that is simultaneously downward and to the right. The `Shape` class can be seen in lines 13–23 of the `ShapeAnimator.h` file.

The `Shape` class has a `move()` function that is pure virtual. This is because a shape is moved by erasing it at its current position and redrawing it at a new position, and it is not possible to know how to draw a shape without knowing what type of shape it is.

Our solution for representing the different shapes will use the five classes `Shape`, `ComplexShape`, `SimpleShape`, `Box`, and `Tent`. These classes form the inheritance hierarchy shown in Figure 15-2.

Figure 15-2



The `SimpleShape` class represents objects that can be drawn at a given position in a specified color. Accordingly, it has member variables for representing position and color and member functions for setting and accessing those values. The `SimpleShape` class appears in lines 26–44 of the `ShapeAnimator.h` file. Notice that the `SimpleShape` class is still abstract because it provides no implementation for the `draw()` method. The class does implement the `move()` method, though. This is because the `move()` method works the same way for all subclasses of SimpleShape: erase the shape at its current position, compute its new position, and draw the shape at the new position. The `draw()` method, however, works differently for each concrete subclass that implements it. Because `draw()` is virtual, the `move()` method will always call the appropriate version of `draw()`, even when the call to `draw()` is through a pointer to the abstract class `Shape`.

The `Box` and `Tent` classes are the concrete classes at the tip of the inheritance hierarchy. They define a specific concrete shape and implement the member function `draw()` that draws the shape at its current position using the shape's specified color. The `Box` class defines a rectangular shape by specifying the position of its top left-hand corner together with its width and height. The `Tent` class defines a triangle with a horizontal base whose two other sides are equal in length and whose height is half the length of the base. A `Tent` object is specified by giving the position of the left end point of its base together with the length of the base. For example, a tent whose base has length 5 would look like this:

```
  *
 ***
*****
```

The `Box` and `Tent` classes can be seen in Lines 46–66 of `ShapeAnimator.h`.

The `ComplexShape` class provides a mechanism for assembling a collection of simple shapes to form a single shape that can be moved using a single command. The class maintains a vector of pointers to `Shape` objects, and implements its `move()` method is by calling the `move()` methods of all the `Shape` objects in its collection. Likewise, `ComplexShape` has a `setDirection()` method that can be used to cause all of its constituent shapes to move in the same direction. The class itself is found in lines 70–78 of the `ShapeAnimator.h` file, and its `move()` method is implemented in lines 124–128 of `ShapeAnimator.cpp`.

Contents of `SortAnimator.h`

```
 1 #include <iostream>
 2 #include <string>
 3 #include <vector>
 4 #include <windows.h>
 5 using namespace std;
 6
 7 // A global constant can be included in more than one cpp file.
 8 // This is the handle to the output console.
 9 const HANDLE outHandle = GetStdHandle(STD_OUTPUT_HANDLE);
10
11 // A shape has a direction and is able to move in that direction.
12 // The move is a virtual member function.
13 class Shape
14 {
15 public:
16     virtual void setDirection(int drow, int dcol)
17       {dRow = drow; dCol = dcol;}
18     void getDirection(int &drow, int &dcol)
19       {drow = dRow; dcol = dCol;}
20     virtual void move()= 0;
21 private:
22     int dRow, dCol; // Direction of motion
23 };
24
25 // A SimpleShape is drawn at a given position in a specified color.
26 class SimpleShape : public Shape
```

```
{
public:
    virtual void draw() = 0;
    void getPosition(int &row, int &col)
    {
        row = rowPos; col = colPos;
    }
    void setPosition(int row, int col)
    {
        rowPos = row; colPos = col;
    }
    void setColor(int c){ color = c; }
    int getColor() {return color; }
    virtual void move();
private:
    int color;
    int rowPos, colPos;
};

// A Box is a rectangular type of shape.
class Box : public SimpleShape
{
public:
    virtual void draw();
    Box(int rowPos, int colPos, int width, int height);
private:
    int width, height;
};

// A Tent is an isosceles triangle whose horizontal base has a
// given length and whose height is half the length of the base.
// The position of the triangle is the left end point of the base.
class Tent : public SimpleShape
{
public:
    virtual void draw();
    Tent(int baseRowPos, int baseColPos, int baseLength);
private:
    int baseLength;
};

// A ComplexShape is made up of simpler shapes. It is represented
// as a vector of pointers to the simpler shapes that make it up.
class ComplexShape : public Shape
{
public:
    ComplexShape(Shape ** shapeCollection, int shapesCount);
    virtual void move();
    virtual void setDirection(int dRow, int dCol);
private:
    vector<Shape *> shapes;
};
```

Contents of ShapeAnimator.cpp

```
#include "ShapeAnimator.h"

//************************************************************
// Moves a simple shape one step by erasing the shape         *
// at its current position, changing its position, and then   *
// redrawing the shape at its new position.                   *
//************************************************************
void SimpleShape::move()
{
    int dRow, dCol; // Direction of motion
    int savedColor = color;
    color = 0;      // Drawing in color 0 erases the shape
    draw();
    // Compute the new position for the shape by adding a step in
    // the proper direction to the current position
    getDirection(dRow, dCol);
    rowPos += dRow;
    colPos += dCol;
    // Draw the shape at its new position in its specified color
    color = savedColor;
    draw();
}
//*********************************
// Draws a tent at its position     *
//*********************************
void Tent:: draw()
{
    int rowPos, colPos;
    COORD pos;
    int currentLength = baseLength;
    // Set the color attribute
    SetConsoleTextAttribute(outHandle, getColor());
    getPosition(rowPos, colPos);
    pos.Y = rowPos; pos.X = colPos;

    // Draw the lines that form the tent beginning with
    // the base and moving up toward the point.
    for (int r = 0; r < (baseLength + 1)/2; r++)
    {
        SetConsoleCursorPosition(outHandle,pos);
        // Draw a horizontal line of a given length
        for (int k = 0; k < currentLength; k++)
        {
            cout << "*";
        }
        cout << endl;
        pos.Y--;
        pos.X ++;
        currentLength -= 2;
    }
    // Restore normal attribute
    SetConsoleTextAttribute(outHandle, 7);
}
```

```
54 //*********************************
55 // Draws a box shape               *
56 //*********************************
57 void Box::draw()
58 {
59     int rowPos, colPos;
60     COORD pos;
61
62     // Set the color attribute for the box
63     SetConsoleTextAttribute(outHandle, getColor());
64     getPosition(rowPos, colPos);
65     pos.X = colPos; pos.Y = rowPos;
66
67     // Draw the lines that make up the  box
68     for (int r = 0; r < height; r++)
69     {
70         SetConsoleCursorPosition(outHandle, pos);
71         for (int c = 0; c < width; c++)
72         {
73             cout << "*";
74         }
75         cout << endl;
76         pos.Y++;
77     }
78     // Restore normal text attribute
79     SetConsoleTextAttribute(outHandle, 7);
80 }
81 //*********************************************
82 // Constructor sets the color, position, and    *
83 // dimensions for a box shape, and draws        *
84 // the box at its initial position              *
85 //*********************************************
86 Box::Box(int rowPos, int colPos, int width, int height)
87 {
88     setColor(4);
89     setPosition(rowPos, colPos);
90     this->width = width;
91     this->height = height;
92     draw();
93 }
94 //*********************************************
95 // Constructor sets the color for a Tent shape, *
96 // sets the position of the tent as well as the *
97 // length of its base and draws it at its       *
98 // initial position                             *
99 //*********************************************
100 Tent::Tent(int baseRowPos, int baseColPos, int baseLength)
101 {
102     setColor(2);
103     setPosition(baseRowPos, baseColPos);
104     this->baseLength = baseLength;
105     draw();
106 }
```

```
107 //****************************************************************
108 // Constructor builds a complex shape by assembling a vector of    *
109 // constituent shapes                                              *
110 //****************************************************************
111 ComplexShape::ComplexShape(Shape ** shapeCollection, int shapesCount)
112 {
113     Shape *p;
114     for (int k = 0; k < shapesCount; k++)
115     {
116         p = shapeCollection[k];
117         shapes.push_back(p);
118     }
119 }
120 //************************************
121 // Moves a complex shape by moving the *
122 // constituent shapes                  *
123 //************************************
124 void ComplexShape::move()
125 {
126     for (int k = 0; k < shapes.size(); k++)
127         shapes[k]->move();
128 }
129 //*****************************************************
130 // Sets the direction of a complex shape by setting the  *
131 // direction of all constituent shapes                   *
132 //*****************************************************
133  void ComplexShape::setDirection(int dRow, int dCol)
134  {
135      // Set direction for the ComplexShape object so
136      // getDirection() will work correctly
137      Shape::setDirection(dRow, dCol);
138
139      // Set the directions for the constituent shapes so
140      // move() will work correctly
141      for (int k = 0; k < shapes.size(); k++)
142          shapes[k]->setDirection(dRow, dCol);
143  }
```

Program 15-7, which follows, illustrates the use of these classes. The program starts out by creating two simple shapes, a tent and a box, in lines 7–8. The tent is created at the left edge of the screen while the box is near the right edge. In lines 14–26, the program moves the tent to the right at the same time that it is moving the box to the left, stopping the motion when the two shapes are within a few coordinates of each other. Lines 27–39 create a complex shape out of the two simple shapes, and then moves the complex shape diagonally downward and to the right. Finally, in lines 40–46, the program moves the box horizontally to the right.

Program 15-7

```
// This program illustrates the use of the various Shape
// classes and subclasses to do graphic animation.
#include "ShapeAnimator.h"
int main()
{
    // Create a tent and a box
    Tent tent(11, 5, 13);
    Box box(5, 65, 4, 7);

    // Draw the tent and the box
    tent.draw();
    box.draw();

    // Set initial direction of motion for the two shapes
    tent.setDirection(0, 1);    // Tent moves horizontally to the right
    box.setDirection(0, -1);    // Box moves horizontally to the left

    // Simultaneously move the tent and the box, this makes them
    // move toward each other
    for (int k = 0; k <= 20; k++)
    {
        Sleep(75);
        tent.move();
        box.move();
    }

    // Create a complex shape composed of the tent and the box
    Shape *myShapes[] = {&tent, &box};
    ComplexShape cS(myShapes, 2);

    // Set direction for the complex shape and  move the
    // complex shape: this moves both the tent and the box
    // diagonally to the right
    cS.setDirection(1, 1);
    for (int k = 0; k < 12; k++)
    {
        Sleep(75);
        cS.move();
    }
    // Move the box by itself horizontally to the right
    box.setDirection(0, 1);
    for (int k = 0; k < 10; k ++)
    {
        Sleep(75);
        box.move();
    }
    return 0;
}
```

Review Questions and Exercises

Fill-in-the-Blank

1. A class that cannot be instantiated is a(n) _____.
2. A member function of a class that is not implemented is called a(n) _____ function.
3. A class with at least one pure virtual member function is called a(n) _____ class.
4. In order to use dynamic binding, a member function of a class needs to be declared as a(n) _____ function.
5. Static binding takes place at _____ time.
6. Dynamic binding takes place at _____ time.
7. The ability of code to execute differently depending on the type of data is called _____.
8. A base class pointer needs a(n) _____ to be assigned to a derived class pointer.
9. The *is-a* relation between classes is best implemented using the mechanism of class _____.
10. The *has-a* relation between classes is best implemented using the mechanism of class _____.
11. If *every* `C1` class object can be used as a `C2` class object, the relationship between the two classes should be implemented using _____.
12. A collection of abstract classes defining an application in skeletal form is called a(n) _____.

C++ Language Elements

Suppose that the classes `Dog` and `Cat` derive from `Animal`, which in turn derives from `Creature`. Suppose further that `pDog`, `pCat`, `pAnimal`, and `pCreature` are pointers to the respective classes. Suppose that `Animal` and `Creature` are both abstract classes.

13. Will the statement

```
Animal a;
```

compile?

14. Will the statement

```
pAnimal = new Cat;
```

compile?

15. Will the statement

```
pCreature = new Dog;
```

compile?

16. Will the statement

```
pCat = new Animal;
```

compile?

17. Rewrite the following two statements to get them to compile correctly.

```
pAnimal = new Dog;
pDog = pAnimal;
```

Algorithm Workbench

18. Write a C++ class that has an array of integers as a member variable, a pure virtual member function

```
bool compare(int x, int y) = 0;
```

that compares its two parameters and returns a boolean value, and a member function

```
void sort()
```

that uses the comparison defined by the compare virtual function to sort the array. The sort function will swap a pair of array elements `a[k]` and `a[j]` if

```
compare (a[k], a[j])
```

returns true. Explain how you can use this class to produce classes that sort arrays in ascending order and descending order.

Find the Errors

19. Find all errors in the following fragment of code.

```
class MyClass
{
public:
   virtual myFun() = 0;
   { cout << "Hello";}
};
```

Soft Skills

20. Suppose that you need to have a class that can sort an array in ascending order or descending order upon request. If an array is already sorted in ascending or descending order, you can easily sort it the other way by reversing it. Now suppose you have two different classes that encapsulate arrays. One provides a member function to reverse its array, while the other provides a member function to sort its array. Can you use multiple inheritance to obtain a quick solution to your problem? Should you? Write a couple of paragraphs explaining whether using multiple inheritance will or will not work to solve this problem, and, if it can, whether this is a good way to solve the problem.

Programming Challenges

1. Analysis of Sorting Algorithms

Design a class `AbstractSort` that can be used to analyze the number of comparisons performed by a sorting algorithm. The class should have a member function compare that is capable of comparing two array elements, and a means of keeping track of the number of comparisons performed. The class should be an abstract class with a pure virtual member function

```
void sort(int arr[ ], int size)
```

which, when overridden, will sort the array by calling the compare function to determine the relative order of pairs of numbers. Create a subclass of `AbstractSort` that uses a simple sorting algorithm to implement the `sort` function. The class should have a member function that can be called after the sorting is done to retrieve the number of comparisons performed.

2. Analysis of Quicksort

Create a subclass of the `AbstractSort` class of Programming Challenge 1 that uses the `Quicksort` algorithm to implement the `sort` function.

3. Sequence Sum

VideoNote
Solving the Sequence Sum Problem

A sequence of integers such as 1, 3, 5, 7, ... can be represented by a function that takes a nonnegative integer as parameter and returns the corresponding term of the sequence. For example, the sequence of odd numbers just cited can be represented by the function

```
int odd(int k) {return 2 * k + 1;}
```

Write an abstract class `AbstractSeq` that has a pure virtual member function

```
virtual int fun(int k) = 0;
```

as a stand-in for an actual sequence, and two member functions

```
void printSeq(int k, int m);
int sumSeq(int k, int m)
```

that are passed two integer parameters k and m, where k < m. The function `printSeq` will print all the terms `fun(k)` through `fun(m)` of the sequence, and likewise, the function `sumSeq` will return the sum of those terms. Demonstrate your `AbstractSeq` class by creating subclasses that you use to sum the terms of at least two different sequences. Determine what kind of output best shows off the operation of these classes, and write a program that produces that kind of output.

4. Flexible Encryption

Write a modification of the encryption program of section 15.5 whose transform function uses an integer key to transform the character passed to it. The function transforms the character by adding the key to it. The key should be represented as a member of the `Encryption` class, and the class should be modified so that it has a member function that sets the encryption key. When the program runs, the main function should ask the user for the input file, the output file, and an encryption key.

Show that with these modifications, the same program can be used for both encryption and decryption.



5. File Filter

A file filter reads an input file, transforms it in some way, and writes the results to an output file. Write an abstract file filter class that defines a pure virtual function for transforming a character. Create one subclass of your file filter class that performs encryption, another that transforms a file to all uppercase, and another that creates an unchanged copy of the original file.

The class should have a member function

```
void doFilter(ifstream &in, ofstream &out)
```

that is called to perform the actual filtering. The member function for transforming a single character should have the prototype

```
char transform(char ch)
```

The encryption class should have a constructor that takes an integer as an argument and uses it as the encrytion key.

6. File Double Spacer

Create a subclass of the abstract filter class of Programming Challenge 5 that double spaces a file: that is, it inserts a blank line between any two lines of the file.

7. Bumper Shapes

Write a program that creates two rectangular shapes and then animates them. The two shapes should start on opposite ends of the screen and then move toward each other. When they meet in the middle of the screen, each shape reverses course and moves toward the edge of the screen. The two shapes keep oscillating and bouncing off of each other in the middle of the screen. The program terminates when the shapes meet each other in the middle for the tenth time.

8. Bow Tie

In this chapter's Tying It All Together we defined a *tent* to be a certain type of triangular shape. Define a *wedge* to be a tent that has been rotated 90 degrees clockwise, and a *reverse wedge* to be a tent rotated 90 degrees counterclockwise. Write a program that creates a wedge and a reverse wedge at the left and right edges of the screen, respectively, and then moves them toward each other until they meet in the middle. The two shapes should form a bow tie when they meet.

This page intentionally left blank

CHAPTER 16

Exceptions, Templates, and the Standard Template Library (STL)

TOPICS

16.1 Exceptions
16.2 Function Templates
16.3 Class Templates
16.4 Class Templates and Inheritance
16.5 Introduction to the Standard Template Library
16.6 Tying It All Together: *Word Transformers Game*

16.1 Exceptions

CONCEPT: **Exceptions are used to signal errors or unexpected events that occur while a program is running.**

Error testing is usually a straightforward process involving `if` statements or other control mechanisms. For example, the following code segment will trap a division-by-zero error before it occurs:

```
if (denominator == 0)
   cout << "ERROR: Cannot divide by zero.\n";
else
   quotient = numerator / denominator;
```

But what if similar code is part of a function that returns the quotient as in the following example:

```
// An unreliable division function
double divide(double numerator, double denominator)
{
   if (denominator == 0)
      {
         cout << "ERROR: Cannot divide by zero.\n";
         return 0;
      }
   else
      return numerator / denominator;
}
```

Functions commonly signal error conditions by returning a predetermined value. In this example, the function returns 0 when division by zero has been attempted. This is unreliable, however, because 0 is a valid result of a division operation. Even though the function displays an error message, the part of the program that calls the function will not know when an error has occurred. Problems like these require more sophisticated error-handling techniques.

Throwing an Exception

One way of handling complex error conditions is with *exceptions*. An exception is a value or an object that signals an error. When the error occurs, an exception is said to be "thrown" because control will pass to a part of the program that catches and handles that type of error. For example, the following code shows the `divide` function, modified to throw an exception when division by zero has been attempted.

```
double divide(double numerator, double denominator)
{
   if (denominator == 0)
      throw "ERROR: Cannot divide by zero.\n";
   else
      return numerator / denominator;
}
```

The following statement causes the exception to be thrown.

```
throw "ERROR: Cannot divide by zero.\n";
```

The `throw` key word is followed by an argument, which can be any value. As you will see, the type of the argument is used to determine the nature of the error. The function above simply throws a string containing an error message.

The line containing a throw statement is known as the *throw point*. When a `throw` statement is executed, control is passed to another part of the program known as an *exception handler.*

Handling an Exception



VideoNote
Throwing and Handling Exceptions

To handle an exception, a program must have a *try/catch* construct. The general format of the try/catch construct is

```
try
{
   // code here calls functions or object member
   // functions that might throw an exception.
}
catch(exception parameter)
{
   // code here handles the exception
}
// Repeat as many catch blocks as needed.
```

The first part of the construct is the *try block*. This starts with the key word `try` and is followed by a block of code executing any statements that might directly or indirectly cause an exception to be thrown. The try block is immediately followed by one or more *catch blocks,* which are the exception handlers. A catch block starts with the key word `catch`, followed by a set of parentheses containing the declaration of an exception parameter. For example, here is a try/catch construct that can be used with the `divide` function:

```
try
{
   quotient = divide(num1, num2);
   cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{
   cout << exceptionString;
}
```

Because the `divide` function throws an exception whose type is a string, there must be an exception handler that catches a string. The catch block shown catches the error message in the `exceptionString` parameter, then displays it with `cout`.

Now let's look at an entire program to see how `throw`, `try`, and `catch` work together. In the first sample run of Program 16-1, valid data is given. This shows how the program should run with no errors. In the second sample run, a denominator of 0 is given. This shows the result of the exception being thrown.

Program 16-1

```
1 // This program illustrates exception handling.
2 #include <iostream>
3 using namespace std;
4
5 // Function prototype
6 double divide(double, double);
7
```

(program continues)

Program 16-1 *(continued)*

```
 8 int main()
 9 {
10    int num1, num2;
11    double quotient;
12
13    cout << "Enter two numbers: ";
14    cin >> num1 >> num2;
15    try
16    {
17       quotient = divide(num1, num2);
18       cout << "The quotient is " << quotient << endl;
19    }
20    catch (char *exceptionString)
21    {
22       cout << exceptionString;
23    }
24    cout << "End of the program.\n";
25    return 0;
26 }
27
28 double divide(double numerator, double denominator)
29 {
30    if (denominator == 0)
31       throw "ERROR: Cannot divide by zero.\n";
32    else
33       return numerator / denominator;
34 }
```

Program Output with Example Input Shown in Bold

```
Enter two numbers: 12 2[Enter]
The quotient is 6
End of the program.
```

Program Output with Example Input Shown in Bold

```
Enter two numbers: 12 0[Enter]
ERROR: Cannot divide by zero.
End of the program.
```

As you can see from the second output screen, the exception caused the program to jump out of the `divide` function and into the catch block. After the catch block has finished, the program resumes with the first statement after the try/catch construct.

What If an Exception Is Not Caught?

There are two possible ways for a thrown exception to go uncaught. The first possibility is for the program to contain no catch blocks with an exception parameter of the right data type. The second possibility is for the exception to be thrown from outside a try block. In either case, the exception will cause the entire program to abort execution.

Object-Oriented Exception Handling with Classes

Now that you have an idea of how the exception mechanism in C++ works, we will examine an object-oriented approach to exception handling. Let's begin by looking at the `IntRange` class:

Contents of `IntRange.h`

```
 1 #ifndef INTRANGE_H
 2 #define INTRANGE_H
 3
 4 #include <iostream>
 5 using namespace std;
 6
 7 class IntRange
 8 {
 9 private:
10    int input;  // For user input
11    int lower;  // Lower limit of range
12    int upper;  // Upper limit of range
13 public:
14    // Exception class.
15    class OutOfRange
16       { };     // Empty class declaration
17    // Member functions.
18    IntRange(int low, int high)   // Constructor
19    {
20       lower = low;
21       upper = high;
22    }
23    int getInput()
24    {
25       cin >> input;
26       if (input < lower || input > upper)
27          throw OutOfRange();
28       return input;
29    }
30 };
31 #endif
```

`IntRange` is a simple class whose member function, `getInput`, lets the user enter an integer value. The value is compared against the member variables `lower` and `upper` (which are initialized by the class constructor). If the value entered is less than `lower` or greater than `upper`, an exception is thrown indicating the value is out of range. Otherwise, the value is returned from the function.

Instead of throwing a character string or or some other value of a primitive type, this function throws an *exception class*. Notice the empty class declaration that appears in the public section:

```
class OutOfRange
   { };  // Empty class declaration
```

Notice that the class has no members. The only important part of this class is its name, which will be used by the exception handling code. Look at the `if` statement in the `getinput` function:

```
if (input < lower || input > upper)
   throw OutOfRange();
```

The throw statement's argument, `OutOfRange()`, causes an instance of the `OutOfRange` class to be created and thrown as an exception. All that remains is for a catch block to handle the exception. Here is an example:

```
catch (IntRange::OutOfRange)
{
   cout << "That value is out of range.\n";
}
```

All that must appear inside the catch block's parentheses is the name of the exception class. The exception class is empty, so there is no need to declare an actual parameter. All the catch block needs to know is the type of the exception.

Since the `OutOfRange` class is declared in the `IntRange` class, its name must be fully qualified with the scope resolution operator. Program 16-2 shows the class at work in a driver program.

Program 16-2

```
1  // This program demonstrates the use of object-oriented
2  // exception handling.
3  #include <iostream>
4  #include "IntRange.h"
5  using namespace std;
6
7  int main()
8  {
9     IntRange range(5, 10);
10    int userValue;
11
12    cout << "Enter a value in the range 5 - 10: ";
13    try
14    {
15       userValue = range.getInput();
16       cout << "You entered " << userValue << endl;
17    }
18    catch (IntRange::OutOfRange)
19    {
20       cout << "That value is out of range.\n";
21    }
22    cout << "End of the program.\n";
23    return 0;
24 }
```

Program Output with Example Input Shown in Bold

```
Enter a value in the range 5 - 10: 12[Enter]
That value is out of range.
End of the program.
```

Multiple Exceptions

The programs we have studied so far test only for a single type of error and throw only a single type of exception. In many cases a program will need to test for several different types of errors and signal which one has occurred. C++ allows you to throw and catch multiple exceptions. The only requirement is that each different exception be of a different type. You then code a separate catch block for each type of exception that may be thrown in the try block.

For example, suppose we wish to expand the `IntRange` class so it throws one type of exception if the user enters a value that is too low, and another type if the user enters a value that is too high. First, we declare two different exception classes, such as

```
// Exception classes
class TooLow
   { };
class TooHigh
   { };
```

An instance of the `TooLow` class will be thrown when the user enters a low value, and an instance of the `TooHigh` class will be thrown when a high value is entered.

Next we modify the `getInput` member function to perform the two error tests and throw the appropriate exception:

```
if (input < lower)
   throw TooLow();
else if (input > upper)
   throw TooHigh();
```

The entire modified class, which is named `IntRange2`, is shown here:

Contents of `IntRange2.h`

```
1  #ifndef INTRANGE2_H
2  #define INTRANGE2_H
3
4  #include <iostream>
5  using namespace std;
6
7  class IntRange2
8  {
9  private:
10    int input;  // For user input
11    int lower;  // Lower limit of range
12    int upper;  // Upper limit of range
13 public:
14    // Exception classes.
15    class TooLow
16       { };
17    class TooHigh
18       { };
19    // Member functions.
```

```
20     IntRange2(int low, int high)  // Constructor
21     {
22          lower = low;
23          upper = high;
24     }
25     int getInput()
26     {
27          cin >> input;
28          if (input < lower)
29             throw TooLow();
30          else if (input > upper)
31             throw TooHigh();
32          return input;
33     }
34 };
35 #endif
```

Program 16-3 is a simple driver that demonstrates this class.

Program 16-3

```
1  // This program demonstrates the IntRange2 class.
2  #include <iostream>
3  #include "IntRange2.h"
4  using namespace std;
5
6  int main()
7  {
8     IntRange2 range(5, 10);
9     int userValue;
10
11    cout << "Enter a value in the range 5 - 10: ";
12    try
13    {
14       userValue = range.getInput();
15       cout << "You entered " << userValue << endl;
16    }
17    catch (IntRange2::TooLow)
18    {
19       cout << "That value is too low.\n";
20    }
21    catch (IntRange2::TooHigh)
22    {
23       cout << "That value is too high.\n";
24    }
25
26    cout << "End of the program.\n";
27    return 0;
28 }
```

Program Output with Example Input Shown in Bold

```
Enter a value in the range 5 - 10: 3[Enter]
That value is too low.
End of the program.
```

Extracting Information from the Exception Class

Sometimes we might want an exception to pass information back to the exception handler. For example, suppose we would like the `IntRange` class not only to signal when an invalid value has been entered, but to pass the value back. This can be accomplished by giving the exception class members in which information can be stored.

`IntRange3`, our next modification of the `IntRange` class, again uses a single exception class: `OutOfRange`. This version of `OutOfRange`, however, has a member variable and a constructor that initializes it:

```
// Exception class
class OutOfRange
{  public:
      int value;
      OutOfRange(int i)
      {  value = i; }
};
```

When we throw this exception, we want to pass the value entered by the user to `OutOfRange`'s constructor. This is done with the following statement:

```
throw OutOfRange(input);
```

This `throw` statement creates an instance of the `OutOfRange` class and passes a copy of the `input` variable to the constructor. The constructor then stores this number in `OutOfRange`'s member variable `value`. The class instance carries this member variable to the catch block that intercepts the exception.

Back in the catch block, the value is extracted:

```
catch (IntRange3::OutOfRange ex)
{
   cout << "That value " << ex.value
        << " is out of range.\n";
}
```

Notice that the catch block declares a parameter object named `ex`. This is necessary, because the exception has a member variable that we want to examine. The entire `IntRange3` class is as follows, and Program 16-4 is a driver that demonstrates it.

Contents of `IntRange3.h`

```
1  #ifndef INTRANGE3_H
2  #define INTRANGE3_H
3
4  #include <iostream>
5  using namespace std;
6
7  class IntRange3
8  {
9  private:
10    int input;  // For user input
11    int lower;  // Lower limit of range
12    int upper;  // Upper limit of range
```

```
public:
   // Exception class.
   class OutOfRange
   {
      public:
      int value;
      OutOfRange(int i)
      {  value = i; }
   };
   // Member functions.
   IntRange3(int low, int high)  // Constructor
   {
      lower = low;
      upper = high;
   }
   int getInput()
   {
      cin >> input;
      if (input < lower || input > upper)
         throw OutOfRange(input);
      return input;
   }
};
#endif
```

Program 16-4

```
// This program demonstrates the IntRange3 class.
#include <iostream>
#include "IntRange3.h"
using namespace std;

int main()
{
   IntRange3 range(5, 10);
   int userValue;

   cout << "Enter a value in the range 5 - 10: ";
   try
   {
      userValue = range.getInput();
      cout << "You entered " << userValue << endl;
   }
   catch (IntRange3::OutOfRange ex)
   {
      cout << "That value " << ex.value
           << " is out of range.\n";
   }
   cout << "End of the program.\n";
   return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter a value in the range 5 - 10: 12[Enter]
That value 12 is out of range.
End of the program.
```

Handling the `bad_alloc` Exception Thrown by `new`

The `new` operator throws a system-defined exception of type `bad_alloc` if it is unable to allocate the requested storage. For example, the following program attempts to allocate an array of two integers using the `new` operator inside a try block. If the allocation fails, the resulting `bad_alloc` exception is caught in the attached catch block and the program is terminated with an appropriate error message. If the allocation succeeds, the program proceeds on to print the two numbers 10 and 20. The `bad_alloc` type is defined in the header file `new`, which must be included in programs that refer to it.

Program 16-5

```
1  // This program demonstrates the use of the bad_alloc
2  // exception.
3  #include <iostream>
4  #include <cstdlib>
5  #include <new>          //needed to use bad_alloc
6  using namespace std;
7
8  int main()
9  {
10    int *p;
11    try
12    {
13       p = new int[2];
14       p[0] = 10;
15       p[1] = 20;
16    }
17    catch(bad_alloc)
18    {
19       cout << "Memory cannot be allocated.";
20       exit(1);
21    }
22    cout << p[0] << "  " << p[1];
23    return 0;
24 }
```

Program Output

```
10 20
```

Unwinding the Stack

If an exception is thrown in a try block that has a catch block capable of handling the exception, control transfers from the throw point to the catch block. Assuming that the catch block executes to completion without throwing further exceptions, returning from the function, or terminating the program, execution will continue at the first statement after the sequence of catch blocks attached to the try block.

If the function does not contain a catch block capable of handling the exception, control passes out of the function, and the exception is automatically rethrown at the point of the call in the calling function. By this process, an exception can propagate backwards along the chain of function calls until the exception is thrown out of a try block that has a catch block that can

handle it. If no such try block is ever found, the exception will eventually be thrown out of the main function, causing the program to be terminated. This process of propagating uncaught exceptions from a function to its caller is called *unwinding the stack* of function calls.

Rethrowing an Exception

It is possible for try blocks to be nested. For example, look at this code segment:

```
void main()
{
   try
   {
      doSomething();
   }
   catch(exception1)
   {
      Code to handle exception 1
   }
   catch(exception2)
   {
      Code to handle exception 2
   }
}
```

In this try block the function `doSomething` is called. There are two catch blocks, one that handles `exception1`, and another that handles `exception2`. If the `doSomething` function contains a try block, then it is nested inside the one shown.

With nested try blocks, it is sometimes necessary for an inner exception handler to pass an exception to an outer exception handler. Sometimes, both an inner and an outer catch block must perform operations when a particular exception is thrown. These situations require that the inner catch block *rethrow* the exception so the outer catch block can catch it.

A catch block can rethrow an exception with the `throw;` statement with no parameter. For example, suppose the `doSomething` function (called in the try block above) executes code that potentially can throw `exception1` or `exception3`. Suppose we do not want to handle the `exception1` error in `doSomething`, but instead want to rethrow it to the outer block. The following code segment illustrates how this is done.

```
void doSomething()
{
   try
   {
      Code that can throw exceptions 1 and 3
   }
   catch(exception1)
   {
      throw; // Rethrow the exception
   }
   catch(exception3)
   {
      Code to handle exception 3
   }
}
```

When the first catch block catches `exception1`, the `throw;` statement simply throws the exception again. The catch block in the outer try/catch construct, in this case the one in the `main` function, will then handle the exception.



Checkpoint

16.1 What is the difference between a try block and a catch block?

16.2 What happens if an exception is thrown, but not caught?

16.3 If multiple exceptions can be thrown, how does the catch block know which exception to catch?

16.4 After the catch block has handled the exception, where does program execution resume?

16.5 How can an exception pass information to the exception handler?

16.2 Function Templates

CONCEPT: A function template is a "generic" function that can work with different data types. The programmer writes the specifications of the function, but substitutes parameters for data types. When the compiler encounters a call to the function, it generates code to handle the specific data type(s) used in the call.

Overloaded functions make programming convenient because only one function name must be remembered for a set of functions that perform similar operations. Each of the functions, however, must still be written individually. For example, consider the following overloaded `square` functions.

```
int square(int number)
{
   return number * number;
}

double square(double number)
{
   return number * number;
}
```

The only differences between these two functions are the data types of their return values and their parameters. In situations like this, it is more convenient to write a *function template* than an overloaded function. Function templates allow you to write a single function definition that works with many different data types, instead of having to write a separate function for each data type used.

A function template is not an actual function, but a "mold" the compiler uses to generate one or more functions. When writing a function template, you do not have to specify actual types for the parameters, return value, or local variables. Instead, you use a *type parameter*

to specify a generic data type. When the compiler encounters a call to the function, it examines the data types of its arguments and generates the function code that will work with those data types.



VideoNote
Writing a Function Template

Here is a function template for the `square` function:

```
template <class T>
T square(T number)
{
   return number * number;
}
```

The beginning of a function template is marked by a *template prefix*, which begins with the key word `template`. Next is a set of angled brackets that contains one or more generic data types used in the template. A generic data type starts with the key word `class`, followed by a parameter name that stands for the data type. The example just given only uses one, which is named `T`. (If there were more, they would be separated by commas.) After this, the function definition is written as usual, except the type parameters are substituted for the actual data type names. In the example the function header reads

```
T square(T number)
```

`T` is the type parameter, or generic data type. The header defines `square` as a function that returns a value of type `T` and uses a parameter, `number`, which is also of type `T`.

As mentioned before, the compiler examines each call to `square` and fills in the appropriate data type for `T`. For example, the following call uses an `int` argument:

```
int y, x = 4;
y = square(x);
```

This code will cause the compiler to generate the function:

```
int square(int number)
{
   return number * number;
}
```

while the statements

```
double y, d = 6.2
y = square(d);
```

will result in the generation of the function

```
double square(double number)
{
   return number * number;
}
```

Program 16-6 demonstrates how this function template is used.

Program 16-6

```
1  // This program uses a function template.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  // Template definition for square function.
7  template <class T>
8  T square(T number)
9  {
10    return number * number;
11 }
12
13 int main()
14 {
15    cout << setprecision(5);
16
17    // Get an integer and compute its square.
18    cout << "Enter an integer: ";
19    int iValue;
20    cin >> iValue;
21
22    // The compiler creates int square(int) at the first
23    // occurrence of a call to square with an int argument.
24    cout << "The square is " <<  square(iValue);
25
26    // Get a double and compute its square.
27    cout << "\nEnter a double: ";
28    double dValue;
29    cin >> dValue;
30
31    // The compiler creates double square(double)at the first
32    // occurrence of a call to square with a double argument.
33    cout << "The square is " << square(dValue) << endl;
34
35    return 0;
36 }
```

Program Output with Example Input Shown in Bold

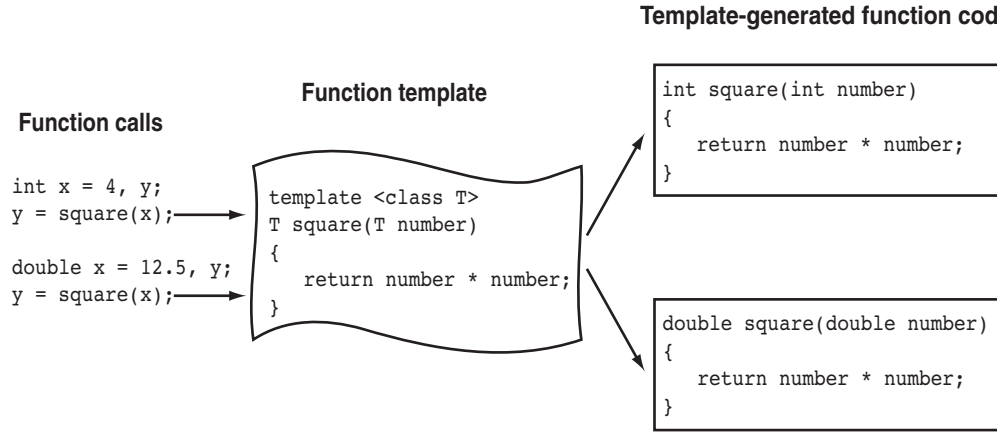
```
Enter an integer: 3[Enter]
The square is 9
Enter a double: 8.3[Enter]
The square is 68.89
```



NOTE: All type parameters defined in a function template must appear at least once in the function parameter list.

Since the compiler encountered two calls to `square` in Program 16-6, each with different parameter types, it generated the code for two instances of the function: one with an `int` parameter and `int` return type, the other with a `double` parameter and `double` return type. This is illustrated in Figure 16-1.

Figure 16-1



Notice in Program 16-6 that the template appears before all calls to `square`. As with regular functions, the compiler must already know the template's contents when it encounters a call to the template function. Templates, therefore, should be placed near the top of the program or in a header file.



NOTE: A function template is merely the specification of a function and by itself does not cause memory to be used. An actual instance of the function is created in memory when the compiler encounters a call to the template function.

The `swap` Function Template

In many applications, there is a need for swapping the contents of two variables of the same type. For example, while sorting an array of integers, there would be a need for the function

```
void swap(int &a, int &b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

whereas while sorting an array string objects, there would be a need for the function

```
void swap(string &a, string &b)
{
   string temp = a;
   a = b;
   b = temp;
}
```

Because the only difference in the coding of these two functions is the type of the variables being swapped, the logic of both functions and all others like them can be captured with one template function:

```
template<class T>
void swap(T &a, T &b)
{
   T temp = a;
   a = b;
   b = temp;
}
```

Such a template function is available in the libraries that come with standard C++ compilers. The function is declared in the `algorithm` header file. Program 16-7 demonstrates the use of this library template function to swap contents of pairs of variables.

Program 16-7

```
1  // This program demonstrates the use of the swap
2  // function template.
3  #include <iostream>
4  #include <string>
5  #include <algorithm>  //needed for swap
6  using namespace std;
7
8  int main()
9  {
10    // Get and swap two chars.
11    char firstChar, secondChar;
12    cout << "Enter two characters: ";
13    cin >> firstChar >> secondChar;
14    swap(firstChar, secondChar);
15    cout << firstChar << " " << secondChar << endl;
16
17    // Get and swap two ints.
18    int firstInt, secondInt;
19    cout << "Enter two integers: ";
20    cin >> firstInt >> secondInt;
21    swap(firstInt, secondInt);
22    cout << firstInt << " " << secondInt << endl;
23
24    // Get and swap two strings.
25    cout << "Enter two strings: ";
26    string firstString, secondString;
27    cin >> firstString >> secondString;
28    swap(firstString, secondString);
29    cout << firstString << " " << secondString << endl;
30    return 0;
31 }
```

(program continues)

Program 16-7 *(continued)*

Program Output With Example Input Shown in Bold

```
Enter two characters: a b[Enter]
b a
Enter two integers: 12 45[Enter]
45 12
Enter two strings: Ronald Reagan[Enter]
Reagan Ronald
```

Using Operators in Function Templates

The `square` function template shown earlier in this section applies the operator `*` to its parameter. The `square` template will work correctly as long as the type of the parameter passed to it supports the `*` operator. For example, it works for numeric types such as `int`, `long`, and `double`, because all these types have a multiplication operator `*`. In addition, the `square` template will work with any user-defined class type that overloads the operator `*`. Errors will result if `square` is used with types that do not support the operator `*`.

Always remember that templates will only work with types that support the operations used by the template. For example, a class can only be used with a template that applies the relational operators such as <, <=, and != to its type parameter if the class overloads those operators.

For example, because the string class overloads all the relational operators, it can be used with template functions that compute the minimum of an array of items. Program 16-8 illustrates this.

Program 16-8

```
1  // This program illustrates the use of function templates.
2  #include <string>
3  #include <iostream>
4  using namespace std;
5
6  // Template for minimum of an array.
7  template <class T>
8  T minimum(T arr[ ], int size)
9  {
10    T smallest = arr[0];
11    for (int k = 1; k < size; k++)
12    {
13       if (arr[k] < smallest)
14          smallest = arr[k];
15    }
16    return smallest;
17 }
18
19 int main()
20 {
21    // The compiler creates int minimum(int [], int)
22    // when you pass an array of int.
23    int arr1[] = {40, 20, 35};
24    cout << "The minimum number is " << minimum(arr1, 3)
25         << endl;
```

(program continues)

Program 16-8 *(continued)*

```
26
27    // The compiler creates string minimum(string [], int)
28    // when you pass an array of string.
29    string arr2[] = {"Zoe", "Snoopy", "Bob", "Waldorf"};
30    cout << "The minimum string is " << minimum(arr2, 4)
31         << endl;
32
33    return 0;
34 }
```

Program Output

```
The minimum number is 20
The minimum string is Bob
```

Function Templates with Multiple Types

More than one generic type may be used in a function template. Here is an example of a function that takes as parameters a list of three values of any printable type, prints out the list in order, and then prints out the list in reverse. The type parameters for the template function are represented using the identifiers `T1`, `T2` and `T3`.

Program 16-9

```
1  // This program illustrates the use of function templates
2  // with multiple types.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  //template function
8  template <class T1, class T2, class T3>
9  void echoAndReverse(T1 a1, T2 a2, T3 a3)
10 {
11    cout << "Original order is: "
12         << a1 << "  " << a2 << "  " << a3 << endl;
13    cout << "Reversed order is: "
14         << a3 << "  " << a2 << "  " << a1 << endl;
15 }
16
17 int main()
18 {
19    echoAndReverse("Computer", 'A', 18);
20    echoAndReverse("One", 4, "All");
21    return 0;
22 }
```

Program Output

```
Original order is: Computer  A  18
Reversed order is: 18  A  Computer
Original order is: One  4  All
Reversed order is: All  4  One
```



NOTE: Each type parameter declared in the template prefix must be used somewhere in the template definition.

Overloading with Function Templates

Function templates may be overloaded. As with regular functions, function templates are overloaded by having different parameter lists. For example, there are two overloaded versions of the `sum` function in Program 16-10. The first version accepts two arguments, and the second version accepts three.

Program 16-10

```
1  // This program demonstrates an overloaded function template.
2  #include <iostream>
3  using namespace std;
4
5  template <class T>
6  T sum(T val1, T val2)
7  {
8     return val1 + val2;
9  }
10
11 template <class T>
12 T sum(T val1, T val2, T val3)
13 {
14    return val1 + val2 + val3;
15 }
16
17 int main()
18 {
19    double num1, num2, num3;
20
21    cout << "Enter two values: ";
22    cin >> num1 >> num2;
23    cout << "Their sum is " << sum(num1, num2) << endl;
24    cout << "Enter three values: ";
25    cin >> num1 >> num2 >> num3;
26    cout << "Their sum is " << sum(num1, num2, num3) << endl;
27    return 0;
28 }
```

Program Output with Example Input Shown in Bold

```
Enter two values: 12.5 6.9[Enter]
Their sum is 19.4
Enter three values: 45.76 98.32 10.51[Enter]
Their sum is 154.59
```

There are other ways to perform overloading with function templates as well. For example, a program might contain a regular (non-template) version of a function as well as a template version. As long as each has a different parameter list, they can coexist as overloaded functions.

Defining Template Functions

In defining template functions, it may be helpful to start by writing a non-template version of the function, and then converting it to a template after it has been tested. The conversion is then achieved by prefixing the function definition with an appropriate template header, say

```
template <class T>
```

and then systematically replacing the relevant type with the generic type `T`. We followed a similar procedure in defining the template for the `swap` function.



Checkpoint

16.6 When does the compiler actually generate code for a function template?

16.7 The function

```
int minPosition(int arr[ ], int size)
```

takes an array of integers of the given size and returns the index of the smallest element of the array. Define a template that works like this function, but permits as parameter arrays of any type that can be compared using the less-than operator <.

16.8 What must you be sure of when passing a class object to a function template that uses an operator, such as * or >?

16.9 What is a good method for writing a function template?

16.3 Class Templates

CONCEPT: Templates may also be used to create generic classes and abstract data types.

Function templates are used whenever we need several different functions with the same problem-solving logic, but which differ only in the types of the parameters they work with. Class templates can be used whenever we need several classes which only differ in the types of some of their data members, or in the types of the parameters of their member functions.

Declaring a class template is similar to declaring a function template: you write the class using identifiers such as `T`, `T1`, `T2` (or whatever other identifier you choose) as generic types, and then prefix the class declaration with an appropriately written template header. For example, suppose that we wish to define a class similar to the `NumberArray` class studied in Chapter 11, that represents an array of a generic type, and adds an overloaded operator `[]` that performs bounds checking. Calling our class `SimpleVector`, and putting in the appropriate data members and constructors, we arrive at the template:

```
template <class T>
class SimpleVector
{
private:
   T *aptr;
   int arraySize;
   void subError();           // Handles subscripts out of range.
```

```
public:
   SimpleVector()
     { aptr = 0; arraySize = 0;}          // Default Constructor
   SimpleVector(int);                     // Constructor
   SimpleVector(const SimpleVector &);    // Copy constructor
   ~SimpleVector();                       // Destructor
   int size()
     { return arraySize; }
   T &operator[](int);                 // Overloaded [] operator
   void print();                       // outputs the array elements
};
```

This class template will store elements of type `T` in a dynamically generated array. This explains why the pointer `aptr`, which will point to the base of this array, is declared to be of type `T*`, a pointer to `T`. Likewise, the overloaded array subscription operator returns a value of type `T`. Notice, however, that the `size` member function and the member `arraySize` that represents the number of elements stored in the array are both of type `int`. This makes sense because the number of elements in an array is always an integer, regardless of the type of element the array stores.

You can think of the `SimpleVector` template as a generic pattern that can be specialized to create classes of `SimpleVector` that hold `double`, `long`, `string`, or any other type that you can define. The rule is that you form the name of such an actual class by appending a list of the actual types, enclosed in angled brackets, to the name of the class template:

- `SimpleVector<double>` is the name of a class that stores arrays of `double`.
- `SimpleVector<string>` is the name of a class that stores arrays of `string`.
- `SimpleVector<char>` is the name of a class that stores arrays of `char`.

Let us look at some examples of defining objects:

```
SimpleVector<int>  iTable1;
SimpleVector<double> dTable1;
SimpleVector<double> dTable2(10);
```

The first two statements use default constructors to create `iTable1` and `dTable1`, which are respectively arrays of `int` and `double` of size 0. The last statement uses the convert constructor to create an array of 10 elements of type `double`.

Defining a member function of a template class inside the class is straightforward: an example is furnished by the definition of the default constructor in the `SimpleVector` class. To define a member function outside the class, you must prefix the definition of the member function with a template header that specifies the list of type parameters, and then within the definition, use the name of the class template followed by a list of the type parameters in angled brackets whenever you need the *name of the class*.

Let us use the operator [] function to illustrate the definition of a member function outside the class.

```
template <class T>
T &SimpleVector<T>::operator[](int sub)
{
   if (sub < 0 || sub >= arraySize)
      subError();
   return aptr[sub];
}
```

In this definition, the name of the class is needed just before the scope resolution operator, so we have `SimpleVector<T>` at that place. As another example, consider the definition of the convert constructor:

```
template <class T>
SimpleVector<T>::SimpleVector(int s)
{
   arraySize = s;
   aptr = new T [s];
   for (int count = 0; count < arraySize; count++)
          aptr[count] = T();
}
```

Here, we need to have `SimpleVector<T>` before the scope resolution operator, but only `SimpleVector`, without the `<T>`, after. This is because what is needed after the scope resolution operator is not the name of the class, but the name of a member function, which in this case happens to be a constructor.

There is an exception to the rule of attaching the list of type parameters to the name of template class. The list, and the angled brackets that enclose it, can be omitted whenever the name of the class is within the scope of the template class. Thus the list can be omitted when the name of a class is being used anywhere within the class itself, or within the local scope of a member function that is being defined outside of the class. For example, the copy constructor

```
template <class T>
SimpleVector<T>::SimpleVector(const SimpleVector &obj)
{
   arraySize = obj.arraySize;
   aptr = new T [arraySize];
   for(int count = 0; count < arraySize; count++)
      aptr[count] = obj[count];
}
```

does not need to append the `<T>` to the `SimpleVector` that denotes the type of its argument.

The convert constructor for `SimpleVector` assumes that the type parameter `T` has a default constructor T() when it executes the assignment `aptr[count] = T();`. If `T` is a primitive type, the C++ compiler will use the default value of 0 in place of `T()`. For example, if `T` were `int`, the assignment is equivalent to `aptr[count] = int();` and a value of 0 will be stored in `aptr[count]`.

The code for the `SimpleVector` template is listed in the `SimpleVector.h` file.

Contents of `SimpleVector.h`

```
1 #ifndef SIMPLEVECTOR_H
2 #define SIMPLEVECTOR_H
3
4 #include <iostream>
5 #include <cstdlib>
6 using namespace std;
7
```

```
template <class T>
class SimpleVector
{
private:
   T *aptr;
   int arraySize;
   void subError();       // Handles subscripts out of range
public:
   SimpleVector(int);                    // Constructor
   SimpleVector(const SimpleVector &);  // Copy constructor
   ~SimpleVector();                      // Destructor
   int size()
      { return arraySize; }
   T &operator[](int);     // Overloaded [] operator
   void print();           // outputs the array elements.
};

//*******************************************************
// Constructor for SimpleVector class. Sets the size    *
// of the array and allocates memory for it.            *
//*******************************************************
template <class T>
SimpleVector<T>::SimpleVector(int s)
{
   arraySize = s;
   aptr = new T [s];
   for (int count = 0; count < arraySize; count++)
      aptr[count] = T();
}
//*******************************************************
// Copy Constructor for SimpleVector class.            *
//*******************************************************
template <class T>
SimpleVector<T>::SimpleVector(const SimpleVector &obj)
{
   arraySize = obj.arraySize;
   aptr = new T [arraySize];
   for(int count = 0; count < arraySize; count++)
      aptr[count] = obj[count];
}
//*******************************************************
// Destructor for SimpleVector class.                  *
//*******************************************************
template <class T>
SimpleVector<T>::~SimpleVector()
{
   if (arraySize > 0)
      delete [] aptr;
}

//*******************************************************
// subError function. Displays an error message and     *
// terminates the program when a subscript is out of    *
// range.                                               *
//*******************************************************
```

```
template <class T>
void SimpleVector<T>::subError()
{
   cout << "ERROR: Subscript out of range.\n";
   exit(0);
}
//*******************************************************
// Overloaded [] operator. The argument is a subscript. *
// This function returns a reference to the element     *
// in the array indexed by the subscript.               *
//*******************************************************
template <class T>
T &SimpleVector<T>::operator[](int sub)
{
   if (sub < 0 || sub >= arraySize)
      subError();
   return aptr[sub];
}
//*******************************************************
// prints all the entries is the array.                 *
//*******************************************************
template <class T>
void SimpleVector<T>::print( )
{
   for (int k = 0; k < arraySize; k++ )
      cout << aptr[k] << "  ";
   cout << endl;
}
#endif
```

Program 16-11 demonstrates the `SimpleVector` template.

Program 16-11

```
// This program demonstrates the SimpleVector template.
#include <iostream>
#include "SimpleVector.h"
using namespace std;

int main()
{
   const int SIZE = 10;

   SimpleVector<int> intTable(SIZE);
   SimpleVector<double> doubleTable(SIZE);

   // Store values in the arrays.
   for (int x = 0; x < SIZE; x++)
   {
      intTable[x] = (x * 2);
      doubleTable[x] = (x * 2.14);
   }

```

(program continues)

Program 16-11 *(continued)*

```
20    // Display the values in the arrays.
21    cout << "These values are in intTable:\n";
22    intTable.print();
23    cout << "These values are in doubleTable:\n";
24
25    doubleTable.print();
26
27    // Use the built-in + operator on array elements.
28    for (int x = 0; x < SIZE; x++)
29    {
30       intTable[x] = intTable[x] + 5;
31       doubleTable[x] = doubleTable[x] + 1.5;
32    }
33    // Display the values in the array.
34    cout << "These values are in intTable:\n";
35    intTable.print();
36    cout << "These values are in doubleTable:\n";
37    doubleTable.print();
38
39    // Use the built-in ++ operator on array elements.
40    for (int x = 0; x < SIZE; x++)
41    {
42       intTable[x]++;
43       doubleTable[x]++;
44    }
45    // Display the values in the array.
46    cout << "These values are in intTable:\n";
47    intTable.print();
48    cout << "These values are in the doubleTable:\n";
49    doubleTable.print();
50    cout << endl;
51    return 0;
52 }
```

Program Output

```
These values are in intTable:
0  2  4  6  8  10  12  14  16  18
These values are in doubleTable:
0  2.14  4.28  6.42  8.56  10.7  12.84  14.98  17.12  19.26
These values are in intTable:
5  7  9  11  13  15  17  19  21  23
These values are in doubleTable:
1.5  3.64  5.78  7.92  10.06  12.2  14.34  16.48  18.62  20.76
These values are in intTable:
6  8  10  12  14  16  18  20  22  24
These values are in the doubleTable:
2.5  4.64  6.78  8.92  11.06  13.2  15.34  17.48  19.62  21.76
```



NOTE: The file that contains the template code has been included in the file that contains the driver code to avoid the complexities of linking separately compiled files that use templates.

16.4 Class Templates and Inheritance

Inheritance can be applied to class templates. For example, in the following template, `SearchableVector` is derived from the `SimpleVector` class.

Contents of `SearchVect.h`

```
#ifndef SEARCHABLEVECTOR_H
#define SEARCHABLEVECTOR_H

#include "SimpleVector.h"

template <class T>
class SearchableVector : public SimpleVector<T>
{
public:
   // Constructor.
   SearchableVector(int s) : SimpleVector<T>(s)
      { }
   // Copy constructor.
   SearchableVector(SearchableVector &);
   // Additional constructor.
   SearchableVector(SimpleVector<T> &obj):
            SimpleVector<T>(obj)
      { }
   int findItem(T);
};

//*****************************************
// Definition of the copy constructor.     *
//*****************************************
template <class T>
SearchableVector<T>::
SearchableVector(SearchableVector &obj) :
SimpleVector<T>(obj)
{
}

//*****************************************
// findItem takes a parameter of type T    *
// and searches for it within the array.   *
//*****************************************
template <class T>
int SearchableVector<T>::findItem(T item)
{
   for (int count = 0; count < this->size(); count++)
   {
      if (this->operator[](count) == item)
         return count;
   }
   return -1;
}
#endif
```

Let us use this example to take a closer look at the derivation of a class from a template base class. First, we have to indicate to the compiler that we are defining a new class template based on an another, already existing class template:

```
template <class T>
class SearchableVector : public SimpleVector<T>
{
   // Members of the class will go here
};
```

Here the new class template being defined is `SearchableVector`, while the existing base class template is `SimpleVector<T>`. The class has three constructors. The first constructor, shown here,

```
SearchableVector(int size) : SimpleVector<T>(size){ }
```

is designed to dynamically allocate an array of `size` elements of type `T`, which it does by invoking the base class constructor and passing it the parameter `size`. This constructor will create an array of the specified size with all elements initialized to default values of type `T`. The class has another constructor,

```
SearchableVector(SimpleVector<T> &obj): SimpleVector<T>(obj){ }
```

which takes as parameter a base class object, a copy of which is to be searched. The constructor simply passes its parameter to the base class copy constructor. The remaining constructor is the copy constructor for the `SearchableVector` class,

```
SearchableVector(SearchableVector<T> &obj): SimpleVector<T>(obj){ }
```

Because the initialization of a `SearchableVector` is the same as that of a `SimpleVector`, the `SearchableVector` copy constructor simply passes its argument to the copy constructor of its base class. The member function `findItem` takes an item of type `T` as its argument and returns the position of the item within the array. If the item is not found in the array, a value of –1 is returned.

Program 16-12 demonstrates the class by storing values in two `SearchableVector` objects and then searching for a specific value in each.

Program 16-12

```
1  // This program demonstrates the SearchableVector template.
2  #include <iostream>
3  #include "searchvect.h"
4  using namespace std;
5
6  int main()
7  {
8     const int SIZE = 10;
9     SearchableVector<int> intTable(SIZE);
10    SearchableVector<double> doubleTable(SIZE);
11
```

(program continues)

Program 16-12 *(continued)*

```
12    // Store values in the vectors.
13    for (int x = 0; x < SIZE; x++)
14    {
15       intTable[x] = (x * 2);
16       doubleTable[x] = (x * 2.14);
17    }
18    // Display the values in the vectors.
19    cout << "These values are in intTable:\n";
20    for (int x = 0; x < SIZE; x++)
21       cout << intTable[x] << " ";
22    cout << endl;
23    cout << "These values are in doubleTable:\n";
24    for (int x = 0; x < SIZE; x++)
25       cout << doubleTable[x] << " ";
26    cout << endl;
27
28    // Now search for values in the vectors.
29    int result;
30    cout << "Searching for 6 in intTable.\n";
31    result = intTable.findItem(6);
32    if (result == -1)
33       cout << "6 was not found in intTable.\n";
34    else
35       cout << "6 was found at subscript "
36            << result << endl;
37
38    cout << "Searching for 12.84 in doubleTable.\n";
39    result = doubleTable.findItem(12.84);
40    if (result == -1)
41       cout << "12.84 was not found in doubleTable.\n";
42    else
43       cout << "12.84 was found at subscript "
44            << result << endl;
45    return 0;
46 }
```

Program Output

```
These values are in intTable:
0 2 4 6 8 10 12 14 16 18
These values are in doubleTable:
0 2.14 4.28 6.42 8.56 10.7 12.84 14.98 17.12 19.26
Searching for 6 in intTable.
6 was found at subscript 3
Searching for 12.84 in doubleTable.
12.84 was found at subscript 6
```

The `SearchableVector` class demonstrates that a class template may be derived from another class template. In addition, class templates may be derived from ordinary classes, and ordinary classes may be derived from class templates.



Checkpoint

16.10 Suppose your program uses a class template named `List`, which is defined as

```
template<class T>
class List
{
   // Members are declared here…
};
```

Give an example of how you would use `int` as the data type in the declaration of a `List` object. (Assume the class has a default constructor.)

16.11 In the following `Rectangle` class declaration, the `width`, `length`, and `area` members are of type `double`. Rewrite the class as a template that will accept any numeric type for these members.

```
class Rectangle
{
   private:
     double width;
     double length;
     double area;
   public:
     void setData(double w, double l)
            { width = w; length = l;}
     void calcArea()
            { area = width * length; }
     double getWidth()
            { return width; }
     double getLength()
            { return length; }
     double getArea()
            { return area; }
};
```

16.5 Introduction to the Standard Template Library

CONCEPT: The Standard Template Library contains many templates for useful algorithms and data structures.

In addition to its run-time library, which you have used throughout this book, C++ also provides a library of templates. The *Standard Template Library* (or *STL*) contains numerous templates for implementing data types and algorithms.

The most important data structures in the STL are the *containers* and *iterators*. A container is a class that stores data and organizes it in some fashion. An iterator is an object that works like a pointer and allows access to items stored in containers.

Sequential Containers

There are two types of container classes in the STL: *sequential* containers and *associative* containers. Sequential containers store items in the form of *sequences*, meaning that there is a natural way to order the items by their position within the container. An array is an example of a sequential container. The STL provides the three sequential containers shown in Table 16-1.

Table 16-1 STL Sequential Containers

Container Name	Description
`vector`	A sequence of items implemented as an array that can automatically grow as needed during program execution. Items can be efficiently added and removed from the vector at its end. Insertions and removals from the middle or beginning of the vector are not as efficient.
`deque`	A sequence of items that has a front and back: items can be efficiently added or removed from the front and back. Insertions and removals in the middle of a deque are not as efficient.
`list`	A sequence of items that allows quick additions and removals from any position.

Because a sequential container organizes the items it stores as a sequence, it can be said to have a *front* and a *back*. A container is said to provide *random access* to its contents if it is possible to specify a position of an item within the container and then jump directly to that item without first having to go through all the items that precede it in the container.

Positions used in random access are usually specified by giving an integer specifying the position of the desired item within the container. The integer may specify a position relative to the beginning of the container, the end of the container, or relative to some other position. Arrays and vectors are examples of sequential containers that provide random access.

Associative Containers

Sequential containers use the position of an item within the sequence to access their data. In contrast, associative containers associate a *key* with each item stored, and then use the key to retrieve the stored item. A telephone book is an example of an associative container; the values stored are telephone numbers, and each telephone number is associated with a name. The name can later be used as a key to look up, or retrieve, the telephone number. The STL provides four associative containers, as shown in Table 16-2.

Table 16-2 STL Associative Containers

Container Name	Description
`set`	Stores a set of keys. No duplicate values are allowed.
`multiset`	Stores a set of keys. Duplicates are allowed.
`map`	Maps a set of keys to data elements. Each key is associated with a unique data element, and duplicate keys are not permitted.
`multimap`	Maps a set of keys to data elements. The same key may be associated with multiple values.

A *map* is a container that requires each value stored to be associated with a key. Each key may be associated with only one value; once a key is used, no other value with the same key may be added to the map. A *multimap* is like a map, except a key may be associated with multiple values.

A *set* is like a map in which only keys are stored, with no associated values. No item may be stored twice in a set: that is, duplicates are not permitted. A *multi set* is like a set in which duplicates are permitted.


VideoNote
Iterators

Iterators

Iterators are objects that behave like pointers. They are used to access items stored in containers. A typical iterator is an object of a class declared inside a container class. The iterator overloads pointer operators such as the increment operator `++` , the decrement operator `--`, and the dereferencing operator `*` in order to provide pointer-like behavior.

Each STL container object provides member functions `begin()` and `end()` that return the beginning and ending iterators for the object. The `begin()` iterator points to the item at the beginning of the container if the container is nonempty, while the `end()` iterator points to just past the end of the container. More details on the use of these iterators will be given later.

Table 16-3 shows the different types of iterators available for use with various STL containers.

Table 16-3 Iterator Types

Iterator Type	Description
Forward	Can only move forward in a container (uses the ++ operator).
Bidirectional	Can move forward or backward in a container (uses the ++ and -- operators).
Random-access	Can move forward and backward, and can jump to a specific data element in a container.
Input	Can be used with `cin` to read information from an input device or a file.
Output	Can be used with `cout` to write information to an output device or a file.

The Use of Iterators

Let us consider the use of iterators, including `begin()` and `end()`, to access items stored in an STL container.

Because an iterator is an object of an inner class called `iterator` that is defined inside a container class, the use of the scope resolution operator is necessary to obtain an iterator for a given container. For example, to define an iterator object `iter` that will work with a class of type `vector<int>`, we write:

```
vector<int>::iterator iter;
```

To define an iterator for a vector of `int` and have it initialized to the beginning of the container `vect`, we write:

```
vector<int> vect;
vector<int>::iterator iter = vect.begin();
```

Once an iterator is defined and made to point to the beginning of the container, the items in the container at the position of the iterator can be accessed by dereferencing the iterator. An iterator can be made to move to the next item in the container by incrementing it, and if the iterator is bidirectional, it can be made to move to the previous element by decrementing it. For example, the statements

```
cout << *iter;
iter++;
```

print the value of an element in the container and then move the iterator to the next element.

Because the end iterator does not point to a legitimate element, an iterator whose value is `end()` should not be dereferenced. Program 16-13 illustrates the use of iterators with vector containers.

Program 16-13

```
1  // This program provides a simple demonstration of the
2  // vector STL template.
3  #include <iostream>
4  #include <vector>  // needed to use vectors
5  using namespace std;
6
7  int main()
8  {
9     vector<int> vect;   // Create a vector of int
10
11    for (int x = 0; x < 10; x++)
12        vect.push_back(x*x);
13
14    //print everything using iterators.
15    vector<int>::iterator iter = vect.begin();
16    while (iter != vect.end())
17    {
18       cout <<  *iter << " ";
19       iter ++;
20    }
21    return 0;
22 }
```

Program Output

```
0 1 4 9 16 25 36 49 64 81
```

Notice in Program 16-13 the inclusion of the `vector` header file, which is required for the `vector` container. The vector container is one of the simplest types of containers in the STL. In the following chapters, you will see examples using other types of containers.

The `vector` Container

Table 16-4 lists a selection of member functions of the vector class template. Some of these accept iterators as parameters, and some of them return iterators as results.

Table 16-4 Selected Member Functions of the Vector Class

Member Function	Description
`at(position)`	Returns the value of the element located at `position` in the vector. Example: `x = vect.at(5);` This statement assigns the value of the element in position 5 of `vect` to `x`.
`back()`	Returns a reference to the last element in the vector. Example: `cout << vect.back() << endl;`
`begin()`	Returns an iterator pointing to the vector's first element. Example: `iter = vect.begin();`
`capacity()`	Returns the maximum number of elements that may be stored in the vector without additional memory being allocated. (This is not the same value as returned by the `size` member function). Example: `x = vect.capacity();` This statement assigns the capacity of `vect` to `x`.
`clear()`	Clears a vector of all its elements. Example: `vect.clear();` This statement removes all the elements from `vect`.

`empty()`	Returns true if the vector is empty. Otherwise, it returns false.
	Example:
	`if (vect.empty())`
	`  cout << "The vector is empty.";`

`end()`	Returns an iterator pointing to just after the last element of the vector. Example: `iter = vect.end();`
`erase(iter)`	Causes the vector element pointed to by the iterator `iter` to be removed. Example: `vect.erase(iter);`
`erase(iter1, iter2)`	Removes all vector elements in the range specified by the iterators `iter1` and `iter2`. Example: `vect.erase(iter1, iter2);`
`front()`	Returns a reference to the vector's first element. Example: `cout << vector.front() << endl;`

(table continues)

Table 16-4 Selected Member Functions of the Vector Class *(continued)*

Member Function	Description
`insert(iter, value)`	Inserts an element into the vector. Example: `vect.insert(iter, x);` This statement inserts the value `x` just before the element pointed to by the iterator `iter`.
`insert(iter, n, value)`	Inserts `n` copies of `value` into the vector, starting just before the position pointed to by the iterator `iter`. Example: `vect.insert(iter, 7, x);` This statement inserts 7 copies of the value `x` just before the element pointed to by the iterator `iter`.
`pop_back()`	Removes the last element from the vector. Example: `vect.pop_back();` This statement removes the last element of `vect`, thus reducing its size by one.
`push_back(value)`	Stores `value` as the new last element of the vector. If the vector is already filled to capacity, it is automatically resized. Example: `vect.push_back(7);` This statement stores 7 as the new last element of `vect`.
`reverse()`	Reverses the order of the elements in the vector (the last element becomes the first element, and the first element becomes the last element.) Example: `vect.reverse();.`
`resize(n)` `resize(n, value)`	Resizes a vector to have `n` elements, where `n` is greater than the vector's current size. If the optional `value` argument is included, each of the new elements will be initialized with that value. Example where vect currently has 4 elements: `vect.resize(6,99);` adds two elements to the end of the vector, each initialized to 99.
`size()`	Returns the number of elements in the `vector`. Example: `cout << vector.size() << endl;`
`swap(vector2)`	Swaps the contents of the vector with the contents of `vector2`. Example: `vect1.swap(vect2);` The statement above swaps the contents of `vect1` and `vect2`.

Algorithms

The algorithms provided by the STL are implemented as function templates and perform various operations on elements of containers. There are many algorithms in the STL; Table 16-5 lists a few of them. (The table gives only general descriptions.)

Table 16-5 STL Algorithms

Algorithm	Description
`binary_search`	Performs a binary search for an object and returns `true` if the object is found, `false` if not. Example: `binary_search(iter1, iter2, value);` In this statement, `iter1` and `iter2` define a range of elements within the container. (`iter1` points to the first element in the range and `iter2` points to just after the last element in the range.) The statement performs a binary search on the range of elements, searching for `value`. The `binary_search` function returns `true` if the element was found and `false` if the element was not found.
`count`	Returns the number of times a value appears in a range. Example: `number = count(iter1, iter2, value);` In this statement, `iter1` and `iter2` define a range of elements within the container. (`iter1` points to the first element in the range and `iter2` points to just after the last element in the range.) The statement returns the number of times `value` appears in the range of elements.
`for_each`	Executes a function for each element in a container. Example: `for_each(iter1, iter2, func);` In this statement, `iter1` and `iter2` define a range of elements within the container. (`iter1` points to the first element in the range and `iter2` points to just after the last element in the range.) The third argument, `func`, is the name of a function. The statement calls the function `func` for each element in the range, passing the element as an argument.
`find`	Finds the first object in a container that matches a value and returns an iterator to it. Example: `iter3 = find(iter1, iter2, value);` In this statement, `iter1` and `iter2` define a range of elements within the container. (`iter1` points to the first element in the range and `iter2` points to just after the last element in the range.) The statement searches the range of elements for `value`. If `value` is found, the function returns an iterator to the element containing it, otherwise, it returns the iterator `iter2`.

(table continues)

Table 16-5 STL Algorithms *(continued)*

Algorithm	Description
`max_element`	Returns an iterator to the largest object in a range. Example: `iter3 = max_element(iter1, iter2);` In this statement, `iter1` and `iter2` define a range of elements within the container. (`iter1` points to the first element in the range and `iter2` points to just after the last element in the range.) The statement returns an iterator to the element containing the largest value in the range.
`min_element`	Returns an iterator to the smallest object in a range. Example: `iter3 = min_element(iter1, iter2);` In this statement, `iter1` and `iter2` define a range of elements within the container. (`iter1` points to the first element in the range and `iter2` points to just after the last element in the range.) The statement returns an iterator to the element containing the smallest value in the range.
`random_shuffle`	Randomly shuffles the elements of a container. Example: `random_shuffle(iter1, iter2);` In this statement, `iter1` and `iter2` define a range of elements within the container. (`iter1` points to the first element in the range and `iter2` points to just after the last element in the range.) The statement randomly reorders the elements in the range.
`sort`	Sorts a range of elements. Example: `sort(iter1, iter2);` In this statement, `iter1` and `iter2` define a range of elements within the container. (`iter1` points to the first element in the range and `iter2` points to just after the last element in the range.) The statement sorts the elements in the range in ascending order.



NOTE: The STL algorithms require the inclusion of the `algorithm` header file.

Program 16-14 demonstrates the use of the `random_shuffle`, `sort`, and `binary_search` algorithm templates.

Program 16-14

```
1 // This program provides a simple demonstration of the
2 // STL algorithms.
3
4 #include <iostream>
5 #include <vector>     // Include the vector header
6 #include <algorithm>  // Required for STL algorithms
7 using namespace std;
8
```

(program continues)

Program 16-14 *(continued)*

```
int main()
{
   vector<int> vect;    // Define a vector object

   // Use push_back to push values into the vector.
   for (int x = 0; x < 10; x++)
      vect.push_back(x*x);
   // Display the vector's elements
   cout << "The collection has " << vect.size()
        << " elements. Here they are:\n";
   for (int x = 0; x < vect.size(); x++)
      cout << vect[x] << " ";
   cout << endl;

   // Randomly shuffle the vector's contents.
   random_shuffle(vect.begin(), vect.end());
   // Display the vector's elements
   cout << "The elements have been shuffled:\n";
   for (int x = 0; x < vect.size(); x++)
      cout << vect[x] << " ";
   cout << endl;

   // Now sort them.
   sort(vect.begin(), vect.end());

   // Display the vector's elements again.
   cout << "The elements have been sorted:\n";
   for (int x = 0; x < vect.size(); x++)
      cout << vect[x] << " ";
   cout << endl;

   // Now search for an element.
   int val = 49;
   if (binary_search(vect.begin(), vect.end(), val))
      cout << "The value " << val
           << " was found in the vector.\n";
   else
      cout << "The value " << val
           << " was not found in the vector.\n";
   return 0;
}
```

Program Output

```
The collection has 10 elements. Here they are:
0 1 4 9 16 25 36 49 64 81
The elements have been shuffled:
64 1 81 4 0 25 49 9 16 36
The elements have been sorted:
0 1 4 9 16 25 36 49 64 81
The value 49 was found in the vector.
```



NOTE: Your run of this program will generate a different random shuffle.

The `random_shuffle` function rearranges the elements of a container. In Program 16-14, it is called in the following manner:

```
random_shuffle(vect.begin(), vect.end());
```

The function takes two arguments, which together represent a range of elements within a container. The first argument is an iterator to the first element in the range. In this case, `vect.begin()` is used. The second argument is an iterator to just after the last element in the range. Here we have used `vect.end()`. These arguments tell `random_shuffle` to rearrange all the elements from the beginning to the end of the `vect` container.

The `sort` algorithm also takes iterators to a range of elements. Here is the function call that appears in Program 16-14:

```
sort(vect.begin(), vect.end());
```

All the elements within the range are sorted in ascending order.

The `binary_search` algorithm searches a range of elements for a value. If the value is found, the function returns `true`. Otherwise, it returns `false`. For example, the following function call searches all the elements in `vect` for the value 7.

```
binary_search(vect.begin(), vect.end(), 7)
```

Program 16-15 demonstrates the `count` algorithm.

Program 16-15

```
1  // This program demonstrates the STL count algorithm.
2  #include <iostream>
3  #include <vector> // Needed to declare the vector
4  #include <algorithm> // Needed for the for_each algorithm
5  using namespace std;
6
7  int main()
8  {
9     vector<int> values;
10    vector<int>::iterator iter;
11
12    // Store some values in the vector.
13    values.push_back(1);
14    values.push_back(2);
15    values.push_back(2);
16    values.push_back(3);
17    values.push_back(3);
18    values.push_back(3);
19
20    // Display the values in the vector.
21    cout << "The values in the vector are:\n";
22    for (iter = values.begin(); iter != values.end(); iter++)
23       cout << *iter << " ";
24    cout << endl << endl;
25
```

(program continues)

Program 16-15 *(continued)*

```
26    // Display the count of each number.
27    cout << "The number of 1s in the vector is ";
28    cout << count(values.begin(), values.end(), 1) << endl;
29    cout << "The number of 2s in the vector is ";
30    cout << count(values.begin(), values.end(), 2) << endl;
31    cout << "The number of 3s in the vector is ";
32    cout << count(values.begin(), values.end(), 3) << endl;
33    return 0;
34 }
```

Program Output

```
The values in the vector are:
1 2 2 3 3 3

The number of 1s in the vector is 1
The number of 2s in the vector is 2
The number of 3s in the vector is 3
```

Program 16-16 demonstrates the `max_element` and `min_element` algorithms.

Program 16-16

```
1  // This program demonstrates the STL max_element
2  // and min_element algorithms.
3  #include <iostream>
4  #include <vector>       // Needed to declare the vector
5  #include <algorithm>    // Needed for the algorithms
6  using namespace std;
7
8  int main()
9  {
10    vector<int> numbers;
11    vector<int>::iterator iter;
12
13    // Store some numbers in the vector.
14    for (int x = 0; x < 10; x++)
15        numbers.push_back(x);
16
17    // Shuffle things up just for fun.
18    random_shuffle(numbers.begin(), numbers.end());
19    // Display the numbers in the vector.
20    cout << "The numbers in the vector are:\n";
21    for (iter = numbers.begin(); iter != numbers.end(); iter++)
22        cout << *iter << " ";
23    cout << endl;
24
25    // Find the largest value in the vector.
26    iter = max_element(numbers.begin(), numbers.end());
27    cout << "The largest value in the vector is "
28         << *iter << endl;
29
```

(program continues)

Program 16-16 *(continued)*

```
30    // Find the smallest value in the vector.
31    iter = min_element(numbers.begin(), numbers.end());
32    cout << "The smallest value in the vector is "
33         << *iter << endl;
34
35    return 0;
36 }
```

Program Output

```
The values in the vector are:
8 1 9 2 0 5 7 3 4 6
The largest value in the vector is 9
The smallest value in the vector is 0
```

Program 16-17 demonstrates the `find` algorithm.

Program 16-17

```
1  // This program demonstrates the STL find algorithm.
2  #include <iostream>
3  #include <vector>    // Needed to declare the vector
4  #include <algorithm> // Needed for the find algorithm
5  using namespace std;
6
7  int main()
8  {
9     vector<int> numbers;
10    vector<int>::iterator iter;
11
12    // Store some numbers in the vector
13    for (int x = 0; x < 10; x++)
14       numbers.push_back(x);
15
16    // Display the numbers in the vector
17    cout << "The numbers in the vector are:\n";
18    for (iter = numbers.begin(); iter != numbers.end(); iter++)
19       cout << *iter << " ";
20    cout << endl << endl;
21
22    // Find 7 in the vector
23    iter = find(numbers.begin(), numbers.end(), 7);
24    cout << "The value searched for is " << *iter << endl;
25    return 0;
26 }
```

Program Output

```
The numbers in the vector are:
0 1 2 3 4 5 6 7 8 9

The value searched for is 7
```

Program 16-18 demonstrates the `for_each` algorithm.

Program 16-18

```
1  // This program demonstrates the for_each find algorithm
2
3  #include <iostream>
4  #include <vector>       // Needed to declare the vector
5  #include <algorithm>    // Needed for the for_each algorithm
6  using namespace std;
7
8  // Function prototype
9  void doubleValue(int &);
10
11 int main()
12 {
13    vector<int> numbers;
14    vector<int>::iterator iter;
15
16    // Store some numbers in the vector.
17    for (int x = 0; x < 10; x++)
18       numbers.push_back(x);
19
20    // Display the numbers in the vector.
21    cout << "The numbers in the vector are:\n";
22    for (iter = numbers.begin(); iter != numbers.end(); iter++)
23       cout << *iter << " ";
24    cout << endl;
25
26    // Double the values in the vector.
27    for_each(numbers.begin(), numbers.end(), doubleValue);
28
29    // Display the numbers in the vector again.
30    cout << "Now the numbers in the vector are:\n";
31    for (iter = numbers.begin(); iter != numbers.end(); iter++)
32       cout << *iter << " ";
33    cout << endl;
34    return 0;
35 }
36
37 //*********************************************************
38 // Function doubleValue. This function accepts an int    *
39 // reference as its argument. The value of the argument  *
40 // is doubled.                                           *
41 //*********************************************************
42 void doubleValue(int &val)
43 {
44    val *= 2;
45 }
```

Program Output

```
The numbers in the vector are:
0 1 2 3 4 5 6 7 8 9
Now the numbers in the vector are:
0 2 4 6 8 10 12 14 16 18
```

In Program 16-18, the following statement calls `for_each`:

```
for_each(numbers.begin(), numbers.end(), doubleValue);
```

The first and second arguments specify a range of elements. In this case, the range is the entire vector. The third argument is the name of a function. The `for_each` algorithm calls the function once for each element in the range, passing the element as an argument to the function.

The programs in this section give you a brief introduction to using the STL by demonstrating simple operations on a `vector`. In the remaining chapters you will be given specific examples of how to use other STL containers, iterators, and algorithms.

16.6 Tying It All Together: *Word Transformers Game*

A software entrepreneur is designing an educational word game for children. A child playing the game is given two words, and must determine if it is possible to rearrange the letters in the first word to form the second. What the program does next depends on the relation between the two words and on the correctness of the player's answer.

When given two words, *the child may claim that transformation of the first word into the second is possible*. In this case, the program asks the child to demonstrate the correctness of the answer by typing a sequence of words starting with the first and ending with the second. Such a sequence is an *acceptable proof sequence* if each word is obtained from its predecessor by swapping a single pair of adjacent letters. For example, the sequence

tops, *tosp*, *tsop*, *stop*, *sotp*, *sopt*, *spot*

proves that the word *tops* can be transformed into spot. If the proof sequence is accepted, the child earns a point and play proceeds to the next round with a fresh pair of words.

A proof sequence is *rejected* if a word cannot be obtained from its predecessor by swapping an adjacent pair of letters, or if the first word in the sequence is not the first word of the pair of words being tested, or if the last word is not the second word in the given pair. When a sequence is rejected, play proceeds to the next round, but the child earns no points.

The child may observe that transformation is not possible. If the child's answer is correct, he or she receives a point and play proceeds to the next round. If the child's answer is incorrect and the transformation is indeed possible, the child receives no points. In this case, however, the program displays a correct proof sequence before moving on to the next round of the game.

A program at the heart of this game must perform several tasks.

1. The program must be able to determine if one of a given pair of words can be transformed into another.
2. The program must be able to determine if one word results from another by swapping an adjacent pair of letters.
3. The program must be able to produce a proof sequence when transformation of one word into another is possible.

How can we write a program for producing proof sequences for a pair of words? One idea is to start with the first word as the current word. Then, swap an adjacent pair of letters to obtain a

new current word. Repeating this strategy will generate a sequence of words. If the current word ever turns out to be the target word, we know we have a proof sequence.

Although the approach we have just outlined is easy to think of, it is difficult to implement. An alternative approach involves sorting. If a transformation between the two words is possible, sorting them will yield the same word. If in addition, we use a sorting method that works by swapping adjacent letters, the sorting process will yield a proof sequence from each of the original words to the same word. For example, sorting *tops* and *spot* yields the same word *opst* with the corresponding sequences

tops, *otps*, *opts*, *opst*

and

spot, *psot*, *post*, *opst*.

Notice that the second sequence is a sequence from *opst* to *spot* in reverse. By concatenating this last sequence to the first and eliminating the duplicate entry in the middle, we obtain the proof sequence from *tops* to *spot*:

tops, *otps*, *opts*, *opst*, *spot*, *psot*, *post*, *spot*

Let us consider some details related to the implementation of this plan. Rather than keep a list of intermediate words generated during the sort, we can keep a list of swaps or *transpositions* performed by the sort. We do this by storing the index *i* for each pair (*i*, *i+1*) of positions of characters swapped by the transposition. Our program uses the well known Bubblesort sorting algorithm. A function

```
sort(char str[], int size, vector<int> &tranpose)
```

is used to sort an array of characters of a given size while saving the list of transpositions performed on the array during the sort. Once both words have been sorted, the resulting two lists of transpositions will be applied to a copy of the first word as previously described, and the words resulting from the application of each transposition will be printed. This strategy is implemented in the following program.

Program 16-19

```
1  // This program solves the word transformation puzzle.
2  #include <iostream>
3  #include <string>
4  #include <vector>
5  #include <algorithm>
6  using namespace std;
7
8  // Prototype
9  void sort(char str[], int size, vector<int>& transpositions);
10
11 int main()
12 {
13       // The two words and a copy of the first word
14       char str1[] = "spot";
15       char str1Copy[] = "spot";
```

(program continues)

Program 16-19 *(continued)*

```
16      char str2[] = "stop";
17
18      // These vectors hold the list of transpositions
19      vector<int> transpose;
20      vector<int> reverse_transpose;
21
22      // Sort the two words
23      cout << "The first word is " << str1 <<  endl
24           << "The second word is " << str2 << endl;
25      sort(str1, 4, transpose);
26      sort(str2, 4, reverse_transpose);
27
28      // Apply the first list of transpositions
29      cout << "The transformation steps are: " << endl;
30      cout << str1Copy <<  " ";
31      for (int k = 0; k < transpose.size(); k++)
32      {
33          int index = transpose[k];
34          swap(str1Copy[index], str1Copy[index + 1]);
35          cout << str1Copy << " ";
36      }
37      // Apply the second list of transpositions in reverse order
38      for (int k = reverse_transpose.size()-1; k >=0 ; k--)
39      {
40          int index = reverse_transpose[k];
41          swap(str1Copy[index], str1Copy[index + 1]);
42          cout << str1Copy << " ";
43      }
44      cout << endl;
45      return 0;
46  }
47
48  //************************************************************
49  // This is a version of Bubblesort that saves a list of all   *
50  // transpositions that are needed to sort the list            *
51  //************************************************************
52  void sort(char str[], int size, vector<int>& transpositions)
53  {
54      // Last index of portion yet to be sorted
55      int upperBound = size-1;
56
57      while (upperBound  > 0)
58      {
59          for (int k = 0; k < upperBound; k++ )
60          {
61              if (str[k] > str[k+1])
```

(program continues)

Program 16-19 *(continued)*

```
62              {
63                  // Save the swap index
64                  transpositions.push_back(k);
65                  swap(str[k], str[k+1]);
66              }
67          }
68          upperBound--;
69      }
70  }
```

Program Output

```
The first word is spot
The second word is stop
The transformation steps are:
spot psot post opst ospt sopt sotp stop
```

Review Questions and Exercises

Fill-in-the-Blank

1. The line containing a throw statement is known as the _____.
2. The _____ block should enclose code that directly or indirectly might cause an exception to be thrown.
3. The _____ block handles an exception.
4. When writing function or class templates, you use a(n) _____ to specify a generic data type.
5. The beginning of a template is marked by a(n) _____.
6. When declaring objects of class templates, the _____ you wish to pass into the type parameter must be specified.
7. A(n)_____ container organizes data in a sequential fashion similar to an array.
8. A(n)_____ container uses keys to rapidly access elements.
9. _____ are pointer-like objects used to access information stored in a container.

C++ Language Elements

10. Modify the `SimpleVector` template presented in this chapter to include an overloaded assignment operator.

Algorithm Workbench

11. Write a function template that takes a generic array of a given size as a parameter and reverses the order of the elements in the array. The first parameter of the function should be the array, the second parameter should be the size of the array.
12. Write a function template that is capable of adding any two numeric values and returning the result.

13. Describe what will happen if you call the function of question 11 and pass it an array of `char`.
14. Describe what will happen if you call the function of question 11 and pass it an array of `string`.

Find the Error

15. Each of the following declarations or code segments has errors. Locate as many as possible.

A)
```
catch
{
   quotient = divide(num1, num2);
   cout << "The quotient is " << quotient << endl;
}
try (char *exceptionString)
{
   cout << exceptionString;
}
```

B)
```
try
{
   quotient = divide(num1, num2);
}
cout << "The quotient is " << quotient << endl;
catch (char *exceptionString)
{
   cout << exceptionString;
}
```

C)
```
template <class T>
T square(T number)
{
   return T * T;
}
```

D)
```
template <class T>
int square(int number)
{
   return number * number;
}
```

E)
```
template <class T1, class T2>
T1 sum(T1 x, T1 y)
{
   return x + y;
}
```

F) Assume the following declaration appears in a program that uses the `SimpleVector` class template presented in this chapter.

```
int <SimpleVector> array(25);
```

G) Assume the following statement appears in a program that has defined `valueSet` as an object of the `SimpleVector` class presented in this chapter. Assume that `valueSet` is a vector of `int`s, and has 20 elements.

```
cout << valueSet<int>[2] << endl;
```

Soft Skills

16. Suppose that you are part of a project team and it becomes clear to you that one of the team members is not "pulling his weight." What should you do if you are the project leader? What should you do if you are not the project leader?

Programming Challenges

myCodeMate

1. String Bound Exceptions

Write a class `BCheckString` that is derived from the STL `string` class. This new class will have two member functions:

A) A `BCheckString(string s)` constructor that receives a string object passed by value and passes it on to the base class constructor.

B) An `char operator[](int k)` function that throws a `BoundsException` object if `k` is negative or is greater than or equal to the length of the string. If `k` is within the bounds of the string, this function will return the character at position `k` in the string.

You will need to write the definition of the `BoundsException` class. Test your class with a main function that attempts to access characters that are within and outside the bounds of a suitably initialized `BCheckString` object.



VideoNote
Solving the Arithmetic Exceptions Problem

2. Arithmetic Exceptions

Write a function that accepts an integer parameter and returns its integer square root. The function should throw an exception if it is passed an integer that is not a perfect square. Demonstrate the function with a suitable driver program.

3. Min/Max Templates

Write templates for the two functions `min` and `max`. `min` should accept two arguments and return the value of the argument that is the lesser of the two. `max` should accept two arguments and return the value of the argument that is the greater of the two. Design a simple driver program that demonstrates the templates with various data types.

myCodeMate

4. Sequence Accumulation

Write a function

```
T accum(vector <T>  v)
```

that forms and returns the "sum" of all items in the vector `v` passed to it. For example, if `T` is a numeric type such as int or double, the numeric sum will be returned, and if `T` represents the STL string type, then the result of concatenation is returned.



NOTE: For any type `T`, the expression `T()` yields the value or object created by the default constructor. For example, `T()` yields the empty string object if `T` is the string class. If `T` represents a numeric type such as `int`, then `T()` yields 0. Use this fact to initialize your "accumulator."

Test your function with a driver program that asks the user to enter 3 integers, uses `accum` to compute the sum, and prints out the sum. The program than asks the user to enter 3 strings, uses `accum` to concatenate the strings, and prints the result.

5. Total Template

Write a template for a function called `total`. The function should keep a running total of values entered by the user, then return the total. The argument sent into the function should be the number of values the function is to read. Test the template in a simple driver program that sends values of various types as arguments and displays the results.

6. Absolute Value Template

Write a function template that accepts an argument and returns its absolute value. The absolute value of a number is its value with no sign. For example, the absolute value of –5 is 5 and the absolute value of 2 is 2. Test the template in a simple driver program.

7. `SimpleVector` Modification

Modify the `SimpleVector` class template, presented in this chapter, to include the member functions `push_back` and `pop_back`. These functions should emulate the STL vector class member functions of the same name. (See Table 16-4.) The `push_back` function should throw an exception if the array is full. The `push_back` function should accept an argument and insert its value at the end of the array. The `pop_back` function should accept no argument and remove the last element from the array. Test the class with a driver program.

8. `SearchableVector` Modification

Modify the `SearchableVector` class template, presented in this chapter, so it performs a binary search instead of a linear search. Test the template in a driver program.

9. `SortableVector` Class Template

Write a class template named `SortableVector`. The class should be derived from the `SimpleVector` class presented in this chapter. It should have a member function that sorts the array elements in ascending order. (Use the sorting algorithm of your choice.) Test the template in a driver program.

10. Two-Dimensional Data

Suppose that data representing a list of people and places they would like to visit is stored in a file as follows:

```
3
0 Paul
1 Peter
2 David

0 3 Chicago Boston Memphis
1 1 Boston
2 0
```

The first number *n* in the file indicates how many people there are in the list. Here *n* is 3, so there are 3 people. Each person in the list is assigned a number in the range 0.. *n*-1 that is

used to identify her. For each person, the file lists the numerical identifier of the person, followed by the number of places the person wants to visit, followed by the names of those places. For example, Boston is the only place that Peter cares to visit, while David wants to visit no places.

Write a program that reads in this type of data from a file and stores it in appropriate STL data structure. For example, you might use vectors, as well as vectors of vectors, to represent this information. The program allows users to type in the name of a person whose list of favorite destinations is to be printed out. The program prints an error message if the person is not in the database.

11. Word Transformers Modification

Modify Program 16-19 so that it keeps lists of intermediate words during the two sorts instead of keeping lists of swap indices.

CHAPTER 17

Linked Lists

TOPICS

17.1 Introduction to the Linked List ADT
17.2 Linked List Operations
17.3 A Linked List Template
17.4 Recursive Linked List Operations
17.5 Variations of the Linked List
17.6 The STL `list` Container
17.7 Reliable Software Systems, Inc., Case Study
17.8 Tying It All Together: *More on Graphics and Animation*

17.1 Introduction to the Linked List ADT

CONCEPT: Dynamically allocated data structures may be linked together in memory to form a chain.

A linked list is a series of connected *nodes*, where each node is a data structure. The nodes of a linked list are usually dynamically allocated, used, and then deleted, allowing the linked list to grow or shrink in size as the program runs. If new information needs to be added to a linked list, the program simply allocates another node and inserts it into the series. If a particular piece of information needs to be removed from the linked list, the program deletes the node containing that information.

Advantages of Linked Lists over Arrays and Vectors

Although linked lists are more complex to code and manage than arrays, they have some distinct advantages. First, a linked list can easily grow or shrink in size. In fact, the programmer doesn't need to know how many nodes will be in the list. They are simply created in memory as they are needed.

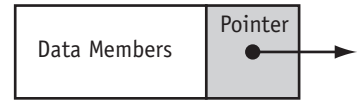
One might argue that linked lists are not superior to vectors (found in the Standard Template Library), because they too can expand or shrink. The advantage that linked lists have over vectors, however, is the speed at which a node may be inserted into or deleted from the list.

To insert a value into the middle of a vector requires all the elements after the insertion point to be moved one position toward the vector's end, thus making room for the new value. Likewise, removing a value from a vector requires all the elements after the removal point to be moved one position toward the vector's beginning. When a node is inserted into or deleted from a linked list, none of the other nodes have to be moved.

The Structure of Linked Lists

Each node in a linked list contains one or more members that hold data. (For example, the data stored in the node may be an inventory record; or it may be a customer information record consisting of the customer's name, address, and telephone number.) In addition to the data, each node contains a *successor* pointer that points to the next node in the list. The makeup of a single node is illustrated in Figure 17-1.

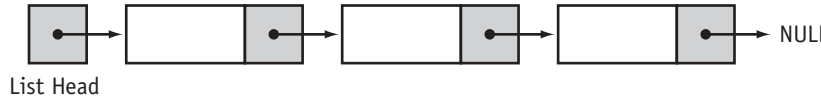
Figure 17-1



The first node of a nonempty linked list is called the *head* of the list. To access the nodes in a linked list, you need to have a pointer to the head of the list. Beginning with the head, you can access the rest of the nodes in the list by following the successor pointers stored in each node. The successor pointer in the last node is set to NULL to indicate the end of the list.

Because the pointer to the head of the list is used to locate the head of the list, we can think of it as representing the list head. The same pointer can also be used to locate the entire list by starting at the head and following the successor pointers, so it is also natural to think of it as representing the entire list. Figure 17-2 illustrates a linked list of three nodes, showing the pointer to the head, the three nodes of the list, and the NULL pointer that signifies the end of the list.

Figure 17-2





NOTE: Figure 17-2 depicts the nodes in the linked list as being very close to each other, neatly arranged in a row. In reality, the nodes may be scattered around various parts of memory.

C++ Representation of Linked Lists

To represent linked lists in C++, we need to have a data type that represents a single node in the list. Looking at Figure 17-1, we see that it is natural to make this data type a structure that contains the data to be stored, together with a pointer to another node of the same

type. Assuming that each node will store a single data item of type `double`, we can declare the following type to hold the node:

```
struct ListNode
{
   double value;
   ListNode *next;
};
```

Here `ListNode` is the type of a node to be stored in the list, the structure member `value` is the data portion of the node, and the structure member `next`, declared as a pointer to `ListNode`, is the successor pointer that points to the next node.

The `ListNode` structure has an interesting property: it contains a pointer to a data structure of the same type, and thus can be said to be a type that contains a reference to itself. Such types are called *self-referential data types*, or *self-referential data structures*.

Having declared a data type to represent a node, we can define an initially empty linked list by defining a pointer to be used as the list head, and initializing it to NULL:

```
ListNode *head = NULL;
```

We can now create a linked list that consists of a single node storing 12.5 as follows:

```
head = new ListNode;     // allocate new node
head->value = 12.5;      // store the value
head->next = NULL;       // signify end of list
```

Now let's see how we can create a new node, store 13.5 in it, and make it the second node in the list. We can use a second pointer to point to a newly allocated node into which the 13.5 will be stored:

```
ListNode *secondPtr = new ListNode;
secondPtr->value = 13.5;
secondPtr->next = NULL;     // second node is end of list
head->next = secondPtr;     // first node points to second
```

Note that we have now made the second node the end of the list by setting its successor pointer, `secondPtr->next`, to NULL, and we have changed the successor pointer of the list head to point to the second node. Program 17-1 illustrates the creation of a simple linked list.

Program 17-1

```
1  // This program illustrates the creation
2  // of linked lists.
3  #include <iostream>
4  using namespace std;
5
6  struct ListNode
7  {
8     double value;
9     ListNode *next;
10 };
11
```

(program continues)

Program 17-1 *(continued)*

```
12 int main()
13 {
14    ListNode *head;
15
16    // Create first node with 12.5
17    head = new ListNode;     // Allocate new node
18    head->value = 12.5;      // Store the value
19    head->next = NULL;       // Signify end of list
20
21    // Create second node with 13.5
22    ListNode *secondPtr = new ListNode;
23    secondPtr->value = 13.5;
24    secondPtr->next = NULL;  // Second node is end of list
25    head->next = secondPtr; // First node points to second
26
27    // Print the list.
28    cout << "First item is " << head->value << endl;
29    cout << "Second item is " << head->next->value << endl;
30    return 0;
31 }
```

Program Output

```
First item is 12.5
Second item is 13.5
```

Using Constructors to Initialize Nodes

Recall that C++ structures can have constructors. It is often convenient to provide the structures that define the type for a list node with one or more constructors, to allow nodes to be initialized as soon as they are created. Recall also that just like regular functions, constructors can be defined with default parameters. It is very common to provide a default parameter of NULL for the successor pointer of a node. Here is an alternative definition of the `ListNode` structure:

```
struct ListNode
{
   double value;
   ListNode *next;
   // constructor
   ListNode(double value1, ListNode *next1 = NULL)
   {
      value = value1;
      next = next1;
   }
};
```

With this declaration, a node can be created in two different ways:

1. by specifying just its `value` part and letting the successor pointer default to NULL, or
2. by specifying both the `value` part and a pointer to the node that is to follow this one in the list.

The first method is useful when we are creating a node to put at the end of a linked list, while the second method is useful when the newly created node is to be inserted at a place in the list where it will have a successor.

Using this new declaration of a node, we can create the previous list of `12.5` followed by `13.5` with much shorter code:

```
ListNode *secondPtr = new ListNode(13.5);
ListNode *head = new ListNode(12.5, secondPtr);
```

We can actually dispense with the second pointer and write the above code as:

```
ListNode *head = new ListNode(13.5);
head = new ListNode(12.5, head);
```

This code is equivalent to what precedes it because the assignment statement

```
head = new ListNode(12.5, head);
```

is evaluated from right to left: first the old value of `head` is used in the constructor, and then the address returned from the new operator is assigned to `head`, becoming its new value.

Building a List

Using the constructor version of `ListNode`, it is very easy to create a list by reading values from a file and adding each newly read value to the beginning of the list of values already accumulated. For example, using `numberList` for the list head, and `numberFile` for the input file object, the following code will read in numbers stored in a text file and arrange them in a list:

```
ListNode *numberList = NULL;
double number;
while (numberFile >> number)
{
   // create a node to hold this number
   numberList = new ListNode(number, numberList);
}
```

Traversing a List

The process of beginning at the head of a list and going through the entire list while doing some processing at each node is called *traversing* the list. For example, we would have to traverse a list if we needed to print the contents of every node in the list. To traverse a list, say one whose list head pointer is `numberList`, we take another pointer `ptr` and point it to the beginning of the list:

```
ListNode *ptr = numberList;
```

We can then process the node pointed to by `ptr` by working with the expression `*ptr`, or by using the structure pointer operator `->`. For example, if we needed to print the value at the node, we could write the code

```
cout << ptr->value;
```

Once the processing at the node is done, we move the pointer to the next node, if there is one, by writing

```
ptr = ptr->next;
```

thus replacing the pointer to a node by the pointer to the successor of the node. Thus to print an entire list we can use code such as

```
ListNode *ptr = numberList;
while (ptr != NULL)
{
   cout << ptr->value << " ";   // process node
   ptr = ptr->next;             // move to next node
}
```

Program 17-2 illustrates these techniques by reading a file of numbers, arranging the numbers in a linked list, and then traversing the list to print the numbers on the screen.

Program 17-2

```
1  // This program illustrates the building
2  // and traversal of a linked list.
3
4  #include <iostream>
5  #include <fstream>
6  using namespace std;
7
8  struct ListNode
9     {
10      double value;
11      ListNode *next;
12      // Constructor.
13      ListNode(double value1, ListNode *next1 = NULL)
14      {
15         value = value1;
16         next = next1;
17      }
18    };
19
20 int main()
21 {
22    // Open the file.
23    ListNode *numberList = NULL;   // List of numbers
24
25    ifstream numberFile("numberFile.dat");
26    if (!numberFile)
27    {
28         cout << "Error in opening the file of numbers.";
29         exit(1);
30    }
31
32    // Read the file into a linked list.
33    double number;                 // Used to read the file
34    cout << "The contents of the file are: " << endl;
35    while (numberFile >> number)
```

(program continues)

Program 17-2 *(continued)*

```
36    {
37       cout << number << "   ";
38       // Create a node to hold this number.
39       numberList = new ListNode(number, numberList);
40    }
41
42    // Traverse the list while printing.
43    cout << endl << "The contents of the list are: " << endl;
44    ListNode *ptr = numberList;
45    while (ptr != NULL)
46    {
47       cout << ptr->value << "   "; // Process node
48       ptr = ptr->next;             // Move to next node
49    }
50    return 0;
51 }
```

Program Output

```
The contents of the file are:
10   20   30   40
The contents of the list are:
40   30   20   10
```



Checkpoint

17.1 Describe the two parts of a node.

17.2 What is a list head?

17.3 What signifies the end of a linked list?

17.4 What is a self-referential data structure?

17.2 Linked List Operations

CONCEPT: **The basic linked list operations are adding an element to a list, removing an element from the list, traversing the list, and destroying the list.**

In this section we develop some simple list classes. The first of these, which we call `NumberList`, will store values of type `double`. It is based on the `ListNode` structure defined in the preceding section, and is shown here.

Contents of `NumberList.h`

```
1  #include <iostream>
2  using namespace std;
3  class NumberList
4  {
5  protected:
6        // Declare a class for the list node.
7        struct ListNode
```

```
8       {
9          double value;
10         ListNode *next;
11         ListNode(double value1, ListNode *next1 = NULL)
12         {
13            value = value1;
14            next = next1;
15         }
16      };
17      ListNode *head;                    // List head pointer
18 public:
19      NumberList() { head = NULL;  }     //Constructor
20      ~NumberList();                     // Destructor
21      void add(double number);
22      void remove(double number);
23      void displayList();
24 };
```

Because `ListNode` does not need to be accessed by any code outside of `NumberList`, we have declared it inside the `NumberList` class. We have also declared `ListNode` in a protected section to make it accessible to classes that may later be derived from `NumberList`.

Notice that the constructor initializes the `head` pointer to `NULL`, thereby indicating that the list starts out empty. The class has an `add` function that takes a value and adds it to the end of the list, as well as a `displayList` function that prints to the screen all values stored in the list. A destructor function destroys the list by deleting all its nodes. With the exception of `remove()`, all of these functions are defined in `NumberList.cpp`. The `remove()` function will be added later.

Contents of `NumberList.cpp`

```
1  #include "NumberList.h"
2  using namespace std;
3
4  //*************************************************
5  // add adds a new element to the end of the list.     *
6  //*************************************************
7  void NumberList::add(double number)
8  {
9      if (head == NULL)
10        head = new ListNode(number);
11     else
12       {
13         // The list is not empty.
14         // Use nodePtr to traverse the list
15         ListNode *nodePtr = head;
16         while (nodePtr->next != NULL)
17            nodePtr = nodePtr->next;
18
19         // nodePtr->next is NULL so nodePtr points to the last node.
20         // Create a new node and put it after the last node.
21         nodePtr->next = new ListNode(number);
22       }
23 }
24
```

```
25 //**************************************************
26 // displayList outputs a sequence of all values     *
27 // currently stored in the list.                    *
28 //**************************************************
29 void NumberList::displayList()
30 {
31     ListNode *nodePtr = head;   // Start at head of list
32     while (nodePtr)
33     {
34        // Print the value in the current node
35        cout << nodePtr->value << "    ";
36        // Move on to the next node
37        nodePtr = nodePtr->next;
38     }
39 }
40
41 //**************************************************
42 // Destructor deallocates the memory used by the list. *
43 //**************************************************
44 NumberList::~NumberList()
45 {
46    ListNode *nodePtr = head;  // Start at head of list
47    while (nodePtr != NULL)
48    {
49        // garbage keeps track of node to be deleted
50        ListNode *garbage = nodePtr;
51        // Move on to the next node, if any
52        nodePtr = nodePtr->next;
53        // Delete the "garbage" node
54        delete garbage;
55    }
56 }
```

Because the `NumberList` class contains pointers to dynamically allocated memory, it needs to be equipped with both a copy constructor and an overloaded assignment operator before it can safely be used in situations that require copies of lists to be made.



VideoNote
Adding an Element to a Linked List

Adding an Element to the List

The `add` member function accepts as an argument a number of type `double`, creates a node containing the number, and adds it to the end of the list. The basic idea is as follows. If the list is empty, the newly created node becomes the only node in the list:

```
head = new ListNode(number);
```

If, on the other hand, the list is not empty, we take a `pointer nodePtr`, set it to the beginning of the list, and walk it down the list until it points to the last node. We will know it is pointing to the last node when `nodePtr->next` equals `NULL`. The code for starting the pointer at the beginning of the list and walking it down to the end is

```
ListNode *nodePtr = head;
while (nodePtr->next != NULL)
       nodePtr = nodePtr->next;
```

Once nodePtr is pointing to the last node, we can add the new node after it by using the code

```
nodePtr->next = new ListNode(number);
```

Putting all of this together, we get the add function shown in lines 7–23 of NumberList.cpp.

Displaying a List

The code for the displayList member function, in lines 29–39, is based on the algorithm for traversing a list presented in the last section.

Destroying the List

It is important for the class's destructor to release all the memory used by the list. It does this by stepping through the list, deleting one node at a time. The code for doing so is found in lines 44–56 of the NumberList.cpp file. A pointer nodePtr starts at the beginning (head) of the list and steps through the list one node at a time. A second pointer, garbage, follows in nodePtr's wake, and is used to delete each node as soon as nodePtr has passed on to the node's successor.

Program 17–3 demonstrates the operation of the member functions of the NumberList class.

Program 17-3

```
1  // This program demonstrates the add and
2  // display linked list operations.
3
4  #include "Numberlist.h"
5  using namespace std;
6
7  int main()
8  {
9       NumberList list;
10      list.add(2.5);
11      list.add(7.9);
12      list.add(12.6);
13      list.displayList();
14      cout << endl;
15      return 0;
16 }
```

Program Output

```
2.5  7.9  12.6
```

Let's step through Program 17-3, observing how the add function builds a linked list to store the three argument values used.

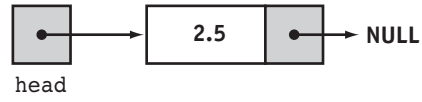
The head pointer, a member variable of the NumberList class, is automatically initialized to NULL by the constructor when the list is created. This indicates that the list is initially empty.

The first call to `add` passes `2.5` as the argument. Because the list is empty at that time, the code

```
head = new ListNode(num);
```

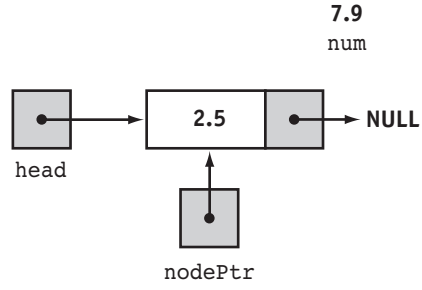
is executed, resulting in the situation depicted in Figure 17-3:

Figure 17-3



There are no more statements to execute, so control returns to function `main`. In the second call to `add`, `7.9` is passed as the argument. The `else` clause of the `if` statement will be executed, setting `nodePtr` to point to the first node of the list, as illustrated in Figure 17-4.

Figure 17-4

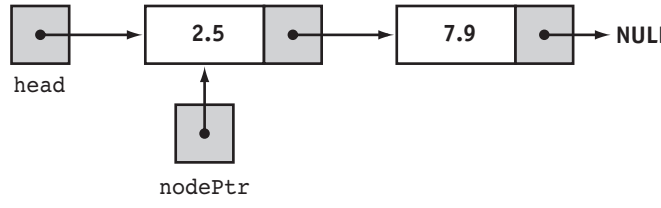


At this point, the pointer `nodePtr->next` has value NULL, and the `while` loop terminates. The statement

```
nodePtr->next = new ListNode(num);
```

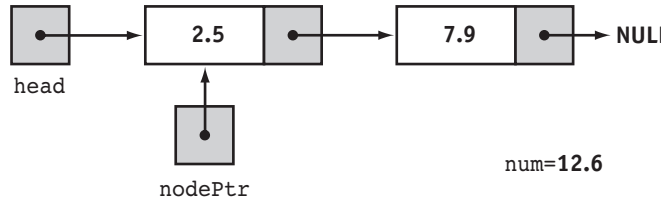
which follows the loop, is then executed, giving the situation depicted in Figure 17.5. The function then returns.

Figure 17-5



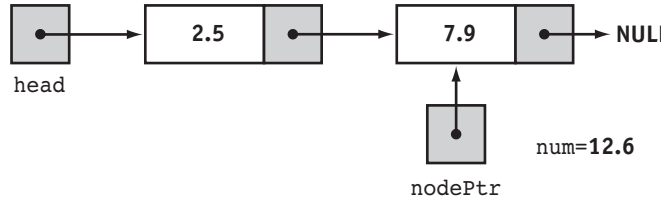
The value `12.6` is passed on the third call to `add`. Again, control will flow to the `else` clause of the if statement because the list is nonempty. The pointer `nodePtr` will be set to the beginning of the list as shown in Figure17-6.

Figure 17-6



Because `nodePtr->next` is not NULL, the `while` loop executes, resulting in the situation illustrated in Figure 17-7.

Figure 17-7

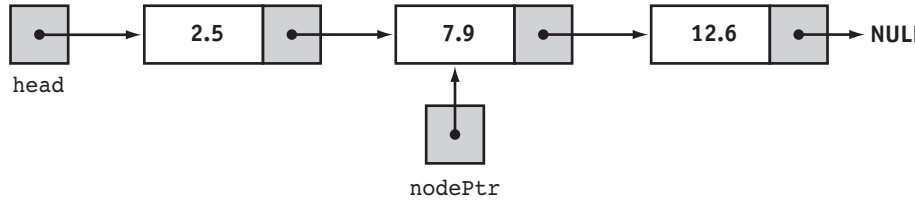


At this point, the `while` loop terminates and the statement

```
nodePtr->next = new ListNode(num);
```

that comes after the `while` loop is executed. This gives the situation depicted in Figure 17-8.

Figure 17-8



Linked Lists in Sorted Order

It is sometimes useful to keep elements added to a linked list in sorted order. For example, the list may maintain its elements in ascending order, meaning that each element in the list is less than or equal to its successor. In these cases, we cannot add elements to the list by putting them at the end of the list as in the add function of the `NumberList` class, because doing so would violate the order of the elements in the list. A different approach is needed.

Consider a class `SortedNumberList` that maintains its elements in ascending order. It is similar to the `NumberList` class, except the `add` function is modified so that it keeps the list in sorted order when placing new elements. Because a sorted list is still a list, it makes sense to use inheritance and derive it from `NumberList`.

Contents of SortedNumberList.h

```
1 #include "NumberList.h"
2 class SortedNumberList : public NumberList
3 {
4 public:
5     void add(double number);
6 };
```

Inserting a Node into a Sorted List

Suppose that we have a linked list of numbers that is sorted in ascending order. We want to write the `add` function so that it inserts its argument `number` in the list at a position that leaves the list sorted.

There are two cases to consider. The first case is when the new number to be inserted should go *before every node already in the list*. This happens when the list is either empty, *or* the first number in the list is greater or equal to `num`:

```
if (head == NULL || head->value >= number)
   head = new ListNode(number, head);
```

Note that the order of these two tests should not be reversed: you should make sure that `head` is not NULL before you try to access `head->value`: trying to evaluate the expression `head->value` will result in a runtime error if `head` is NULL.

The second case that should be considered is when the new number needs to go *after one of the nodes already in the list*. In this case, the new number will need to be placed just *before* the first node that has a value greater or equal to the number. To locate such a node, we use a pointer called `nodePtr`. We will start `nodePtr` at the second node, and then keep moving it forward in the list until it falls off the end of the list (this will happen when `nodePtr` becomes NULL) or it points to a node whose value is greater or equal to `number` (this will happen when the expression `nodePtr->value >= number` becomes true). In order to insert the new node just before `nodePtr`, we will need a pointer to the node that precedes the one that `nodePtr` points to. To this end, we use a pointer `previousNodePtr` that always points to the node previous to the one that `nodePtr` points to. The whole process of finding the insertion point is accomplished by the following code:

```
previousNodePtr = head;
nodePtr = head->next;

// Find the insertion point.
while (nodePtr != NULL && nodePtr->value < number)
{
   previousNodePtr = nodePtr;
   nodePtr = nodePtr->next;
}
```

The entire function, including the code for creating a new node and inserting it at the point just after `previousNodePtr` but before `nodePtr`, is given here:

Contents of SortedNumberList.h

```
1 #include "SortedNumberList.h"
2
3 //*****************************************
4 // Adds a number to the sorted list.      *
5 // This function overrides add in NumberList. *
6 //*****************************************
```

```
 7 void SortedNumberList::add(double number)
 8 {
 9    ListNode *nodePtr, *previousNodePtr;
10
11    if (head == NULL || head->value >= number)
12    {
13       // A new node goes at the beginning of the list.
14       head = new ListNode(number, head);
15    }
16    else
17    {
18       previousNodePtr = head;
19       nodePtr = head->next;
20
21       // Find the insertion point.
22       while (nodePtr != NULL && nodePtr->value < number)
23       {
24          previousNodePtr = nodePtr;
25          nodePtr = nodePtr->next;
26       }
27       // Insert the new node just before nodePtr.
28       previousNodePtr->next = new ListNode(number, nodePtr);
29    }
30 }
```

Here is a program that uses the `add` function. A discussion of how the function works follows the program.

Program 17-4

```
1 // This program illustrates the NumberList append,
2 // insert, and displayList member functions.
3 #include "SortedNumberList.h"
4
5 int main()
6 {
7    SortedNumberList list;
8
9    // Add elements in order
10   list.add(2.5);
11   list.add(7.9);
12   list.add(12.6);
13   // Add a value that should go in the middle of the list
14   list.add(10.5);
15   // Display the list.
16   list.displayList();
17   cout << endl;
18   return 0;
19 }
```

Program Output

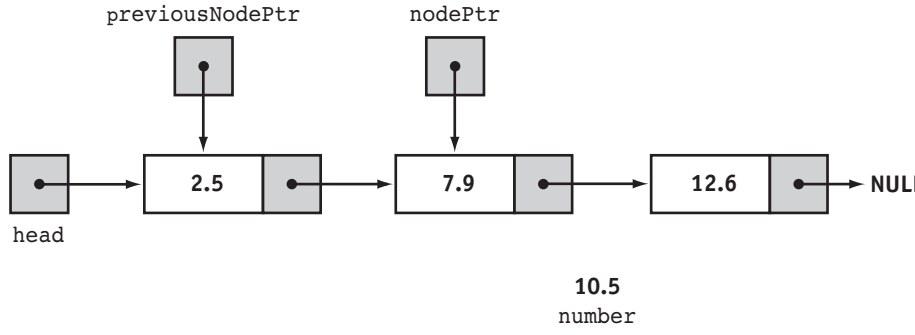
```
2.5  7.9  10.5  12.6
```

Like Program 17-3, Program 17-4 starts out by building a list with the values 2.5, 7.9, and 12.6. Because of the order of addition to the list, each of these values is handled by the if clause in lines 11–15 of `SortedNumberList.cpp`. The `add` function is then called with argument 10.5. This time, the else part in lines 16–26 is executed. The statements

```
previousNodePtr = head;
nodePtr = head->next;
```

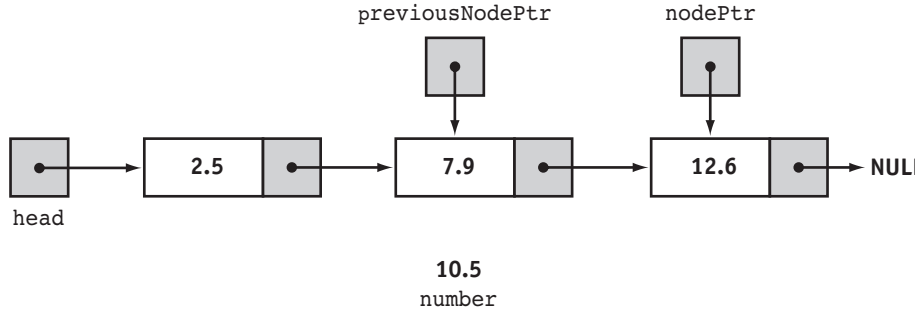
are executed, giving the situation depicted in Figure 17-9.

Figure 17-9



The `while` loop then executes once, leaving the state of the linked list as shown in Figure 17-10.

Figure 17-10

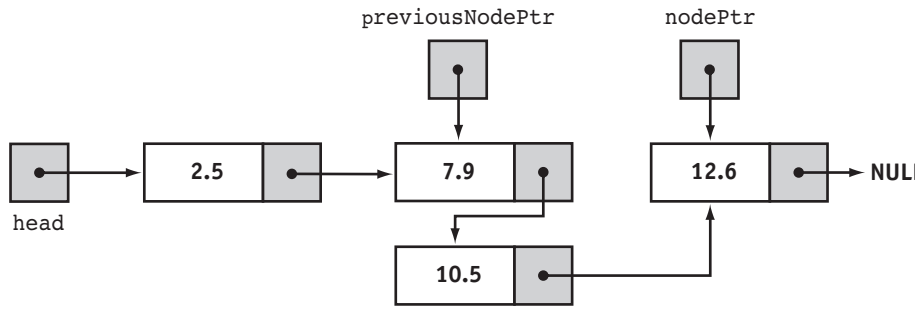


At this point, `nodePtr->value` is greater or equal to `number`, so the loop terminates. The statement after the loop is executed:

```
previousNodePtr->next = new ListNode(number, nodePtr);
```

This final state of the list is illustrated in Figure 17-11.

Figure 17-11



This leaves the list in its final state. If you follow the links, from the `head` pointer to the NULL, you will see that the nodes are stored in the order of their `value` members.

Checkpoint

17.5 What is the difference between appending a node to a list and inserting a node into a list?

17.6 Which is easier to code, appending or inserting?

17.7 Why does the `insertNode` function shown in this section use a `previousNodePtr` pointer?

Removing an Element

VideoNote
Removing an Element from a Linked List

Removing an element from a linked list requires a number of steps:

1. Locating the node containing the element to be removed.
2. Unhooking the node from the list.
3. Deleting the memory allocated to the node.

The `remove` member function uses a pointer `nodePtr` to search for a node containing the value `number` that is to be removed. During this process, a second pointer `previousNodePtr` trails behind `nodePtr`, always pointing to the node preceding the one pointed to by `nodePtr`. When `nodePtr` points to the node to be deleted, `previousNodePtr->next` is set to `nodePtr->next`. This causes the successor pointers in the list to bypass the node containing `number`, allowing its memory to be freed using `delete`. The entire function is shown here:

```
25 //*******************************************
26 // Removes a number from a list. The function  *
27 // does not assume that the list is sorted.    *
28 //*******************************************
29 void NumberList::remove(double number)
30 {
31   ListNode *nodePtr, *previousNodePtr;
32
33    // If the list is empty, do nothing.
34    if (!head) return;
35
36    // Determine if the first node is the one to delete.
37    if (head->value == number)
```

```
38    {
39      nodePtr = head;
40      head = head->next;
41      delete nodePtr;
42    }
43    else
44    {
45      // Initialize nodePtr to the head of the list.
46      nodePtr = head;
47
48      // Skip nodes whose value member is not number
49      while (nodePtr != NULL && nodePtr->value != number)
50      {
51         previousNodePtr = nodePtr;
52         nodePtr = nodePtr->next;
53      }
54      // Link the previous node to the node after
55      // nodePtr, then delete nodePtr.
56      if (nodePtr)
57      {
58         previousNodePtr->next = nodePtr->next;
59         delete nodePtr;
60      }
61    }
62 }
```

Notice that the `remove()` function is a member of `NumberList` rather than `SortedNumberList`. Unlike `add()`, the `remove()` function works with both sorted and unsorted lists, and so does not have to be overridden. The file `RNumberList.cpp`, found on the student CD, is a simple modification of the `NumberList.cpp`: it simply adds the implementation of `remove()`. Program 17-5 demonstrates this new function by first building a list of three values and then removing the values one by one.

Program 17-5

```
1  // This program demonstrates the remove member function.
2  #include "NumberList.h"
3  using namespace std;
4
5  int main()
6  {
7      NumberList list;
8
9      // Build the list.
10     list.add(2.5);
11     list.add(7.9);
12     list.add(12.6);
13
14     // Display the list.
15     cout << "Here are the initial values:\n";
16     list.displayList();
17     cout << "\n\n";
```

(program continues)

Program 17-5 *(continued)*

```
18
19      // Demonstrate the remove function.
20      cout << "Now removing the value in the middle.\n";
21      list.remove(7.9);
22      cout << "Here are the values left.\n";
23      list.displayList();
24      cout << "\n\n";
25
26      cout << "Now removing the last value.\n";
27      list.remove(12.6);
28      cout << "Here are the values left.\n";
29      list.displayList();
30      cout << "\n\n";
31
32      cout << "Now removing the only remaining value.\n";
33      list.remove(2.5);
34      cout << "Here are the values left.\n";
35      list.displayList();
36      cout << endl;
37
38      return 0;
39  }
```

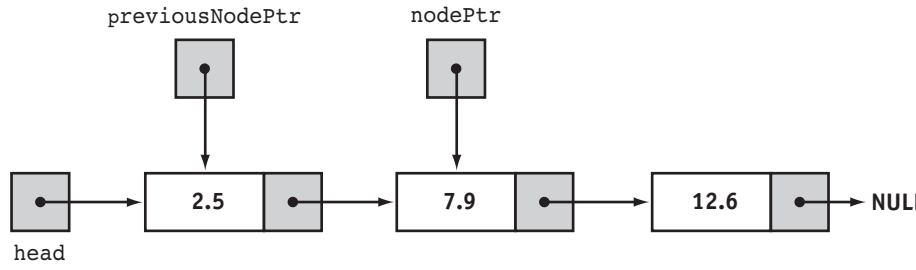
Program Output

```
Here are the initial values:
2.5  7.9  12.6

Now removing the value in the middle.
Here are the values left.
2.5  12.6

Now removing the last value.
Here are the values left.
2.5

Now removing the only remaining value.
Here are the values left.
```

To illustrate how `remove` works, we will step through the first call, the one that removes 7.9 from the list. This is a value that is in the middle of the list.

Look at the else part of the second if statement, lines 44–61. This is where the function will perform its action because the list is not empty and the first node does not contain the value 7.9. Just like the sorted list version of `add()`, this function uses `nodePtr` and `previousNodePtr` to traverse the list. The while loop terminates when the value 7.9 is located. At this point, the list head and the other pointers will be in the state depicted in Figure 17-12.

Figure 17-12

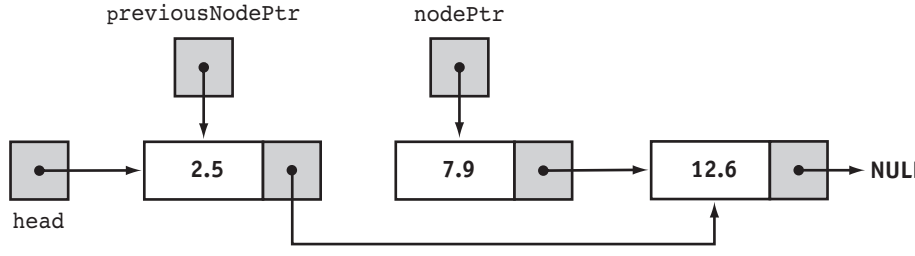


Next, the following statement executes.

```
previousNodePtr->next = nodePtr->next;
```

This statement causes the links in the list to bypass the node that `nodePtr` points to. Although the node still exists in memory, this removes it from the list, as illustrated in Figure 17-13.

Figure 17-13



The last statement uses the `delete` operator to free the memory used by the deleted node.

Checkpoint

17.8 What are the two steps involved in deleting a node from a linked list?

17.9 When deleting a node, why can't you just use the `delete` operator to remove it from memory? Why must you take the steps you listed in response to question 17.8?

17.10 In a program that uses several linked lists, what might eventually happen if the class destructor does not destroy its linked list?

17.3 A Linked List Template*

A major limitation of the `NumberList` class is that it can only hold values of type double. A list class is most useful when it can be used to hold values of different types. The `LinkedList` class, which we will cover next, uses templates to achieve type flexibility. It uses the same logic as the `NumberList` class.

* Note: This section should be skipped if Chapter 16 has not yet been covered.

Contents of LinkedList.h

```
1  #include <iostream>
2  using namespace std;
3  template <class T>
4  class LinkedList
5  {
6  protected:
7      // Declare a class for the list node.
8      struct ListNode
9      {
10        T value;
11        ListNode *next;
12        ListNode(T value1, ListNode *next1 = NULL)
13        {
14           value = value1;
15           next = next1;
16        }
17     };
18     ListNode *head;                    // List head pointer
19 public:
20     LinkedList() { head = NULL;  }     // Constructor
21     ~LinkedList();                     // Destructor
22     void add(T value);
23     void remove(T value);
24     void displayList();
25 };
26
27 //*************************************************
28 // Adds a new element to the end of the list.         *
29 //*************************************************
30 template <class T>
31 void LinkedList<T>::add(T value)
32 {
33    if (head == NULL)
34       head = new ListNode(value);
35    else
36      {
37        // The list is not empty.
38        // Use nodePtr to traverse the list
39        ListNode *nodePtr = head;
40        while (nodePtr->next != NULL)
41           nodePtr = nodePtr->next;
42
43        // nodePtr->next is NULL so nodePtr points to the last node.
44        // Create a new node and put it after the last node.
45        nodePtr->next = new ListNode(value);
46      }
47 }
48
49 //*******************************************
50 // Removes a number from a list. The function  *
51 // does not assume that the list is sorted.    *
52 //*******************************************
53 template <class T>
54 void LinkedList<T>::remove(T value)
```

```
{
  ListNode *nodePtr, *previousNodePtr;

   // If the list is empty, do nothing.
   if (!head)  return;

   // Determine if the first node is the one to delete.
   if (head->value == value)
   {
     nodePtr = head;
     head = head->next;
     delete nodePtr;
   }
   else
   {
     // Initialize nodePtr to the head of the list.
     nodePtr = head;

     // Skip nodes whose value member is not num.
     while (nodePtr != NULL && nodePtr->value != value)
     {
        previousNodePtr = nodePtr;
        nodePtr = nodePtr->next;
     }
     // Link the previous node to the node after
     // nodePtr, then delete nodePtr.
     if (nodePtr)
     {
        previousNodePtr->next = nodePtr->next;
        delete nodePtr;
     }
   }
}

//**************************************************
// displayList outputs a sequence of all values     *
// currently stored in the list.                    *
//**************************************************
template <class T>
void LinkedList<T>::displayList()
{
   ListNode *nodePtr = head;   // Start at head of list
   while (nodePtr)
   {
      // Print the value in the current node
      cout << nodePtr->value << "    ";
      // Move on to the next node
      nodePtr = nodePtr->next;
   }
}

//**************************************************
// Destructor deallocates the memory used by the list. *
//**************************************************
```

```
109 template <class T>
110 LinkedList<T>::~LinkedList()
111 {
112   ListNode *nodePtr = head;  // Start at head of list
113   while (nodePtr != NULL)
114   {
115       // garbage keeps track of node to be deleted
116       ListNode *garbage = nodePtr;
117       // Move on to the next node, if any
118       nodePtr = nodePtr->next;
119       // Delete the "garbage" node
120       delete garbage;
121   }
122 }
```

Notice that the implementation of the class member functions, previously in a separate `.cpp` file, have now been folded into the header file. This has been done to avoid the tremendous complexities of compiling and linking a multifile program that uses templates.

The template class will work for any data type that supports comparison operators such as `==` and `<=`. In particular, it will work for all numeric types and for `string`. Program 17-6 shows the template being used as list of strings.

Program 17-6

```
1  // This program demonstrates the linked list template
2  // being used to create a linked list of strings.
3  #include <string>
4  #include "LinkedList.h"
5  using namespace std;
6
7  int main()
8  {
9        LinkedList<string> list;
10
11         // Build the list.
12       list.add("Alice");
13       list.add("Chuck");
14       list.add("Elaine");
15       list.add("Fran");
16
17       cout << "Here are the initial names:\n";
18       list.displayList();
19       cout << "\n\n";
20
21       cout << "Now removing Elaine.\n\n";
22       list.remove("Elaine");
23       cout << "Here are the remaining elements.\n";
24       list.displayList();
25       cout << endl;
26
27       return 0;
28 }
```

(program continues)

Program 17-6 *(continued)*

Program Output

```
Here are the initial names:
Alice  Chuck  Elaine  Fran

Now removing Elaine.

Here are the remaining elements.
Alice  Chuck  Fran
```

17.4 Recursive Linked List Operations

CONCEPT: Recursion is a useful technique for working with linked lists.

Recursion is a useful approach to solving problems that can be broken down into smaller problems of the same type. Some data structures, such as arrays and linked lists, mirror this property of recursion in that a large array can be split into smaller arrays; and likewise, a nonempty linked list can be reduced to a smaller linked list by removing its first node. Because of this, both array and linked list operations are often well suited to a recursive solution. In this section, we will take a look at the recursive implementation of linked list operations.

Let's take a look at some examples of recursive linked list operations. We will first look at the implementation of recursive stand-alone functions, and then later on in the section, we will look at how member functions of a class can be made recursive. We will use for our examples linked lists of numbers based on the node type

```
struct ListNode
{
   double value;
   ListNode *next;
   ListNode(double value1, ListNode *next1 = NULL)
   {
      value = value1;
      next = next1;
   }
};
```

We have used a structure here to represent the node for ease of presentation only, normally, the node would be a class type to restrict access to its private members.

Recall that the *head* of a nonempty list is the first item on the list. The *tail* of a nonempty list is the list that remains after you remove the head. For example, any list with only one item has the empty list for its tail. A list of numbers `2.5`, `7.9`, `12.6` has the list `7.9`, `12.6` as its tail. With a declaration such as `ListNode`, if a nonempty list is represented by a pointer `ptr`, the tail will be represented by `ptr->next`.

Finally, remember that a good recursive solution must be careful to identify and deal with base cases of the problem, that is, the subproblems resulting from the breaking down process that can be directly solved. In the case of linked lists, the process will often involve breaking

a list down by separating it into its head and tail, and then recursively solving the problem on the tail. The base case will usually be when the list on which the operation is to be performed is empty, or in some cases, has only one item.

Recursive List Functions

Let's write some recursive linked list functions. The function

```
int size(ListNode *ptr)
```

takes as parameter a pointer to the head node of a linked list and returns the number of elements stored in the list. If the list is empty, its size is zero:

```
if (ptr == NULL) return 0;
```

But if a list is nonempty, its size will be one more than the size of its tail:

```
if (ptr != NULL) return 1 + size(ptr->next);
```

Putting these two observations together, we arrive at the following code for the `size()` function:

```
int size(ListNode *ptr)
{
   if (ptr == NULL)
      return 0;
   else
      return 1 + size(ptr->next);
}
```

Consider now a recursive strategy for a function

```
void displayList(ListNode *ptr)
```

that takes a pointer to the head node of a list and prints the list elements. There is nothing to print if the list is empty. To display a nonempty list, we first display the element stored in the head node

```
cout << ptr->value << "  ";
```

and then recursively display the tail of the list. Because the tail of the list is given by `ptr->next`, we arrive at the following code:

```
void displayList(ListNode *ptr)
{
  if (ptr != NULL)
  {
     cout << ptr-> value << "  ";
     displayList(ptr->next);
  }
}
```

Program 17-7 gathers these two functions together and illustrates their use. The program reads data from a file `Numberfile.dat` that can be found on the student CD.

Program 17-7

```
1  // This program illustrates recursion on linked lists.
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  struct ListNode
7    {
8       double value;
9       ListNode *next;
10      // Constructor.
11      ListNode(double value1, ListNode *next1 = NULL)
12      {
13        value = value1;
14        next = next1;
15      }
16   };
17
18 // Function prototypes.
19 int size(ListNode *);
20 void displayList(ListNode *);
21
22 int main()
23 {
24    ListNode *numberList = NULL;   // List of numbers
25
26    // Open the file.
27    ifstream numberFile("numberFile.dat");
28    if (!numberFile)
29    {
30         cout << "Error in opening the file of numbers.";
31         exit(1);
32    }
33
34    // Read the file into a linked list.
35    double number;                  // Used to read the file
36    while (numberFile >> number)
37    {
38       // Create a node to hold this number.
39       numberList = new ListNode(number, numberList);
40    }
41
42    // Print the list.
43    cout << endl << "The contents of the list are: " << endl;
44    displayList(numberList);
45
46    // Print the size of the list.
47    cout << endl << "The number of items in the list is: "
48         << size(numberList);
49    return 0;
50 }
51
```

(program continues)

Program 17-7 *(continued)*

```
52 //*****************************************
53 // length computes the number of nodes in *
54 // a linked list                          *
55 //*****************************************
56 int size(ListNode *ptr)
57 {
58   if (ptr == NULL)
59     return 0;
60   else
61    return 1 + size(ptr->next);
62 }
63
64 //*****************************************
65 // displayList prints all the values stored *
66 // in the list                             *
67 //*****************************************
68 void displayList(ListNode *ptr)
69 {
70   if (ptr != NULL)
71     {
72       cout << ptr-> value << "  ";
73       displayList(ptr->next);
74     }
75 }
```

Program Output:

```
The contents of the list are:
40  30  20  10
The number of items in the list is: 4
```

Recursive Member Functions

Let's write a new version of the `NumberList` class in which the member functions for adding an element, removing an element, and displaying the list have recursive implementations. The class will also have a `size()` function. Here is the class declaration:

Contents of `NumberList2.h`

```
 1 #include <iostream>
 2 using namespace std;
 3 class NumberList2
 4 {
 5 protected:
 6     // Declare a class for the list node.
 7     struct ListNode
 8     {
 9        double value;
10        ListNode *next;
11        ListNode(double value1, ListNode *next1 = NULL)
12        {
13           value = value1;
14           next = next1;
15        }
16     };
```

```
17      ListNode *head;                   // List head pointer
18  public:
19      NumberList2() { head = NULL; }     // Constructor
20      ~NumberList2();                    // Destructor
21      void add(double value) { head = add(head, value);}
22      void remove(double value) {head = remove(head, value);}
23      void displayList() {displayList(head);}
24      int size(){return size(head);}
25  private:
26      // Recursive implementations
27      ListNode *add(ListNode *aList, double value);
28      ListNode *remove(ListNode *aList, double value);
29      void displayList(ListNode *aList);
30      int size(ListNode *aList);
31  };
```

If you look at the class, you will notice that each public member function in lines 20–24 has a corresponding private member function in lines 27–30. The private member functions provide recursive implementations for their public counterparts. Notice that each of the private member functions has a parameter of type `ListNode*`. This parameter is needed for the recursion to work.

You might wonder why we do not make the recursive functions public. The reason is that the parameters of type `ListNode *` are implementation details, and therefore should not be exposed to the users of the class. The user of the public interface of the class does not need to know that the list is internally implemented using a pointer to `ListNode` named `head`.

The Recursive `add` Member Function

Notice that the recursive `add` member function

```
ListNode *add(ListNode *aList, double value);
```

takes as parameters an input list and a value and returns the list that results from adding the value to the input list. Technically, the function takes as its first parameter a pointer to the head of a linked list and returns a pointer to the head of the resulting list. Line 21 of the code listing of `NumberList2.h` shows how the recursive function is called to add a value to the list.

Let's see how the `add` function works. If the input list is empty (base case), the function creates a new node containing the value and returns a pointer to that node:

```
return new ListNode(value);
```

If the list is not empty, the function proceeds as follows. First, it splits the input list into its constituent head node and tail.

```
ListNode *tail = aList->next;  // Fix the tail
aList->next = null;            // aList now points to the head
```

The tail is shorter than the original input list, and is therefore closer to the base case. Using recursion, the function adds the value to the tail of the list, resulting in a "bigger" tail:

```
ListNode *biggerTail = add(tail, value);
```

Finally, the original head, which is being pointed to by `aList`, is reattached to the bigger tail, and a pointer to the original head is returned:

```
aList->next = biggerTail;  // Reattach the head
return aList;              // Return pointer to augmented list
```

Putting all of this together, we get the following code for the `add` function:

```
42 NumberList2::ListNode *NumberList2::add(ListNode *aList, double value)
43 {
44     if (aList == NULL)
45         return new ListNode(value);
46     else
47     {
48         // Split into constituent head and tail
49         ListNode *tail = aList->next;   // tail
50         aList->next = NULL;             // Detached head
51         // Recursively add value to tail
52         ListNode *biggerTail = add(tail, value);
53         // Reattach the head
54         aList->next = biggerTail;
55         // Return pointer to head of bigger list
56         return aList;
57     }
58 }
```

The code in this function can be shortened. First, notice that line 50 is not needed. The head does not have to be detached before making the recursive call on the tail in line 52, as long as it is "reattached" in line 54. Then, we can eliminate the tail variable and just use `aList->next` in line 52. The code in the else clause then gets shortened to

```
ListNode *biggerTail = add(aList->next, value);
aList->next = biggerTail;
return aList;
```

which can in turn be shortened to

```
aList->next = add(aList->next, value);
return  aList;
```

The `add` function can therefore be written as follows:

```
28 NumberList2::ListNode *NumberList2::add(ListNode *aList, double value)
29 {
30     if (aList == NULL)
31         return new ListNode(value);
32     else
33     {
34         // add the value to the end of the tail
35         aList->next = add(aList->next, value);
36         return aList;
37     }
38 }
```

The Recursive `remove` Member Function

The `remove` function

```
ListNode *remove(ListNode *aList, double value)
```

takes as parameter an input list and a value, removes the value from the input list, and returns the resulting list. If the value to be removed is not on the list, the function returns the input list unchanged.

The function works as follows. If the list is empty, the function returns `NULL`.

```
if(aList == NULL) return NULL;
```

Otherwise, the function compares the value to what is stored in the first (head) node of the list. If the value is found there, the head node (pointed to by aList) is deleted and the function returns the tail:

```
if (aList->value == value)
{
   ListNode *tail = aList->next;
   delete aList;
   return tail;
}
```

The last case considered is when the list is not empty and the head of the list does not contain the value to be removed. In this case, the function recursively removes the value from the tail of the list, reattaches the original head to the modified tail, and returns a pointer to the head of the (possibly) modified list. Using the same reasoning as in the `add()` function, we can write this case as

```
aList->next = remove(aList->next, value);
return aList;
```

Again putting it all together, we get the complete function as found lines 10–60 of the implementation file `NumberList2.cpp`.

Contents of `NumberList2.cpp`

```
 1 #include "NumberList2.h"
 2
 3 //****************************************
 4 // Returns the number of elements in a list *
 5 // ****************************************
 6 int NumberList2::size(ListNode *aList)
 7 {
 8     if (aList == NULL)
 9         return 0;
10     else
11         return 1 + size(aList->next);
12 }
13
14 //****************************************
15 // Prints all elements stored in a list     *
16 //****************************************
17 void NumberList2::displayList(ListNode *aList)
```

```
{
    if (aList != NULL)
    {
        cout << aList->value << "  ";
        displayList(aList->next);
    }
}
//*********************************************
// Adds a value at the end of a list           *
//*********************************************
NumberList2::ListNode *NumberList2::add(ListNode *aList, double value)
{
    if (aList == NULL)
        return new ListNode(value);
    else
    {
        // add the value to the end of the tail
        aList->next = add(aList->next, value);
        return aList;
    }
}

NumberList2::ListNode *NumberList2::remove(ListNode *aList, double value)
{
    if (aList == NULL) return NULL;
    // The list is not empty

    // See if value is first on the list.
    // If so, delete the value and return the tail
    if (aList->value == value)
    {
       ListNode *tail = aList->next;
       delete aList;
       return tail;
    }
    else
    {
        // value is not the first on the list.
        // Return the list with the value removed
        // from the tail of the list
        aList->next = remove(aList->next, value);
        return aList;
    }
}

NumberList2::~NumberList2()
{
   ListNode *ptr = head;
   while (ptr != NULL)
```

```
    {
        // Point to the node to be deleted
        ListNode *garbage = ptr;
        // Go on to the next node
        ptr = ptr->next;
        // Delete the current node
        delete garbage;
    }
}
```

The following program demonstrates the use of these member functions.

Program 17-8

```
// This program demonstrates the recursive member
// functions of the NumberList2 class.
#include "NumberList2.h"

int main()
{
    NumberList2 list;
    double number;
    list.add(23);
    list.add(17);
    list.add(59);
    cout << "The members of the list are: ";
    list.displayList();
    cout << "\n";
    cout << "Enter a number to add: ";
    cin >> number;
    list.add(number);
    cout << "The members of the list are: ";
    list.displayList();
    cout << "\n";
    cout << "Enter a number to remove: ";
    cin >> number;
    list.remove(number);
    cout << "The members of the list are: ";
    list.displayList();
    cout << "\n";
    return 0;
}
```

Program Output with Example Input Shown in Bold

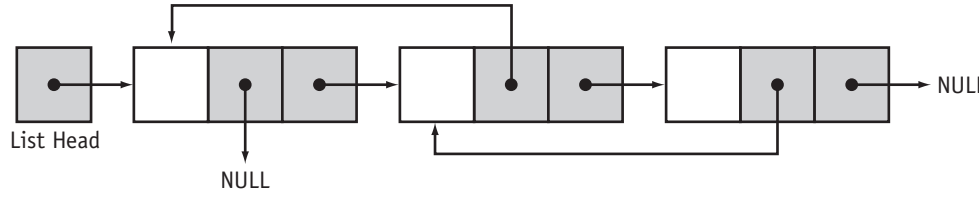
```
The members of the list are: 23  17  59
Enter a number to add: 89
The members of the list are: 23  17  59  89
Enter a number to remove: 17
The members of the list are: 23  59  89
```

17.5 Variations of the Linked List

CONCEPT: **There are many ways to link dynamically allocated data structures together. Two variations of the linked list are the doubly linked list and the circular linked list.**

The linked list examples that we have discussed are *singly linked lists*: Each node is linked to a single other node. A variation of this is the *doubly linked list*. In this type of list, each node not only points to the next node, but also to the previous one. This is illustrated in Figure 17-14.

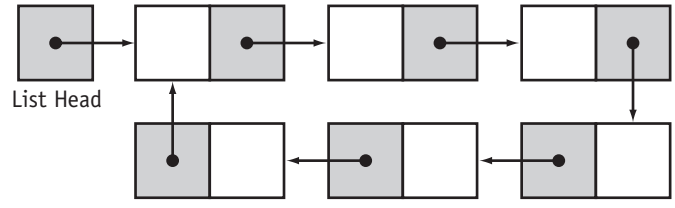
Figure 17-14



In Figure 17-14, the last node and the first node in the list have pointers to the NULL address. When the program traverses the list it knows when it has reached either end.

Another variation is the *circular linked list*. The last node in this type of list points to the first, as shown in Figure 17-15.

Figure 17-15



17.6 The STL `list` Container*

CONCEPT: **The Standard Template Library provides a linked list container.**

The `list` container, found in the Standard Template Library, is a template version of a doubly linked list. STL `lists` can insert elements or add elements within the list more quickly than vectors can, because lists do not have to shift the other elements. Lists are also efficient at adding elements at their back because they have a built-in pointer to the last element in the list (no traversal required).

* Note: This section should be skipped if Chapter 16 has not yet been covered.

Table 17-1 describes some of the list member functions.

Table 17-1 Selected List Member Functions

Member Function	Examples and Description
back	`cout << list.back() << endl;` The back member function returns a reference to the last element in the list.

erase	`list.erase(iter);`
	`list.erase(firstIter, lastIter)`
	The first form causes the list element pointed to by the iterator iter to be removed. The second form causes all of the list elements from firstIter to lastIter to be removed.

empty	`if (list.empty())` The empty member function returns true if the list is empty. It returns false if the list has elements.
end	`iter = list.end();` end returns an iterator to the end of the list.
front	`cout << list.front() << endl;` front returns a reference to the first element of the list.
insert	`list.insert(iter, x)` The insert member function inserts an element into the list. The example inserts an element with the value x, just before the element pointed to by iter.
merge	`list1.merge(list2);` The merge member function expects both list1 and list2 to be already sorted. Every element of list2 will be inserted into list1 in such a way that the expanded list1 remains sorted.
pop_back	`list.pop_back();` pop_back removes the last element of the list.
pop_front	`list.pop_front();` pop_front removes the first element of the list.
push_back	`list.push_back(x);` push_back inserts an element with value x at the end of the list.
push_front	`list.push_front(x);` push_front inserts an element with value x at the beginning of the list.
reverse	`list.reverse();` reverse reverses the order in which the elements appear in the list.
size()	Returns the number of elements in the list.
swap	`list1.swap(list2)` The swap member function swaps the elements stored in two lists. For example, assuming list1 and list2 are lists, the statement shown will exchange the values in the two lists.
unique	`list.unique();` unique eliminates duplicate values by removing any element that has the same value as the element before it.

Program 17-9 demonstrates some simple operations with the STL lists.

Program 17-9

```
1  // This program demonstrates the STL list container.
2  #include <iostream>
3  #include <list>     // Include the list header
4  using namespace std;
5
6  int main()
7  {
8     list<int> myList;
9     list<int>::iterator iter;
10
11    // Add values to the list.
12    for (int x = 0; x < 100; x += 10)
13       myList.push_back(x);
14
15    // Display the values.
16    for (iter = myList.begin(); iter != myList.end(); iter++)
17       cout << *iter << " ";
18    cout << endl;
19
20    // Now reverse the order of the elements.
21    myList.reverse();
22
23    // Display the values again.
24    for (iter = myList.begin(); iter != myList.end(); iter++)
25       cout << *iter << " ";
26    cout << endl;
27    return 0;
28 }
```

Program Output

```
0 10 20 30 40 50 60 70 80 90
90 80 70 60 50 40 30 20 10 0
```

17.7 Reliable Software Systems, Inc., Case Study

Problem Statement

Reliable Software Systems, Inc., writes and markets C++ class libraries for use by programmers worldwide. One of the company's products is a library package that includes the `NumberList` class introduced in section 2 of this chapter. Its customers need to use the class in programs in which copies and assignment of `NumberList` objects will occur. You have been asked to modify the class to support a copy constructor and an assignment operator.

Planning for the Changes and Class Design

At least two functions need to be added to the `NumberList` class. Rather than modifying the original class, you opt to use inheritance to create a new class with the requested enhancements. Both copy constructor and assignment need to make a copy of the linked list of nodes inside of the `NumberList` object being copied. To avoid duplication of code, we will add a member function

```
ListNode *copyList(ListNode *aList);
```

that creates and returns a distinct copy of a list of nodes. In addition, the assignment operator, when applied as in the statement

```
x = y;
```

will need to deallocate storage allocated to the linked list in the `NumberList` object `x`. Accordingly, we add a member function

```
void destroyList(ListNode *aList);
```

to the class. The result of this design work is the `ReliableNumberList` class shown in the listing of the `ReliableNumberList.h` file.

Contents of `ReliableNumberList.h`

```
 1 #include "numberlist.h"
 2
 3 class ReliableNumberList : public NumberList
 4 {
 5 public:
 6     // Copy constructor
 7     ReliableNumberList(const ReliableNumberList& original);
 8     // Now we need a default constructor
 9     ReliableNumberList(){}
10     // Assignment operator
11     ReliableNumberList& operator=(ReliableNumberList right);
12 private:
13     static ListNode* copyList(ListNode *aList);
14     static void destroyList(ListNode *aList);
15 };
```

We have added a default constructor (Line 9) to allow lists that are initially empty to be created. Notice that the auxiliary functions `copyList` and `destroyList` are declared static. This is because they are generic utility functions that do not require access to specific `NumberList` objects to do their job.

Implementation of Class Member Functions

We adopt a recursive strategy for implementing `copyList` and `destroyList`. If a list is empty, `copyList` returns `NULL`. If a list is not empty, then the function creates a copy of the head node, attaches it to recursively created copy of the tail, and returns the resulting list.

Consider now the working of `destroyList`. There is nothing to destroy if the argument list is empty. If the argument list is nonempty, the functions recursively destroys the tail and then deallocates the storage for the head node. The coding details for both `copyList` and `destroyList` can be seen in the listing of `ReliableNumberList.cpp` that follows.

Having the `copyList` function makes writing the copy constructor almost trivial: the constructor simply copies the linked list in the existing object and assigns the result to the head pointer of the object being created:

```
head = copylist(original.head);
```

Coding the assignment operator is not much harder. The operator first deallocates the storage for the linked list in the calling object, and then assigns a copy of the list in its right operand to the head member of the calling object.

```
destroyList(head);
head = copyList(right.head);
```

You can find the full implementation details and an illustration of the use of this new class in the following listing.

Contents of `ReliableNumberList.h`

```
#include "reliablenumberlist.h"

//*************************************************
// Copy constructor                               *
//*************************************************
ReliableNumberList::
ReliableNumberList(const ReliableNumberList& original)
{
    head = copyList(original.head);
}

//*************************************************
// Overloaded Assignment operator                 *
//*************************************************
ReliableNumberList&
ReliableNumberList::operator=(ReliableNumberList right)
{
    // First destroy the linked list in this object
    destroyList(head);
    // Assign a copy of the linked list in other object
    head = copyList(right.head);
}

//*************************************************
// Make a separate copy of the linked list inside *
// a ReliableNumberList object                    *
//*************************************************
NumberList::ListNode *
ReliableNumberList::copyList(ListNode *aList)
{
    if (aList == NULL)
        return NULL;
    else
```

```
    {
        // First copy the tail
        ListNode *tailCopy = copyList(aList->next);
        // Return copy of head attached to copy of tail
        return new ListNode(aList->value, tailCopy);
    }
}

//*****************************************************
// Destroy a list by deallocating all of its nodes     *
//*****************************************************
void ReliableNumberList::destroyList(ListNode *aList)
{
   if (aList != NULL)
   {
      ListNode *tail = aList->next;
      // deallocate the head and then destroy the tail
      delete aList;
      destroyList(tail);
   }
}
```

Program 17-10

```
// This program demonstrates the copy constructor
// and assignment operator added to NumberList.
#include "reliablenumberlist.h"
int main()
{
    ReliableNumberList squareList, cubeList;
    // Store values in the two lists
    for (int k = 1; k <= 5; k++)
    {
        squareList.add(k*k);
        cubeList.add(k*k*k);
    }

    // Use copy constructor to create a third list
    ReliableNumberList otherList(squareList);
    cout << "Result of the copy constructor is:  ";
    otherList.displayList();
    cout << endl;

    // Use the assignment operator
    otherList = cubeList;
    cout << "Result of assignment is:            ";
    otherList.displayList();
    cout << endl;
    return 0;
}
```

(program continues)

Program 17-10 *(continued)*

Program Output

```
Result of the copy constructor is:  1  4  9   16  25
Result of assignment is:            1  8  27  64  125
```

17.8 Tying It All Together: *More on Graphics and Animation*

In previous chapters you learned how to use text-based graphics to draw and animate simple geometric shapes like straight lines, rectangles, and triangles. The techniques you learned can be extended to more complex shapes and figures.

Before you can draw a shape, you must determine the screen coordinates of the characters that will form both its outline and interior. For example, consider the slanted line segment shown in Figure 17-16.

Figure 17-16



The line starts at (0, 0) and ends at (3, 3), and is drawn by placing asterisks at the screen coordinates (0, 0), (1, 1), (2, 2), and (3, 3).

Representing Shapes with Image Maps

More generally, a figure or shape may consist of several parts. Each of the individual parts making up the figure may be a line segment, a geometric shape such as a rectangle or triangle, or some other type of shape. It is convenient to use an array of coordinates to specify a part of a multi-part figure, and then combine the arrays into a single list that defines the whole figure. We use the term *image map* to refer to the list of coordinates that specifies a shape to be drawn.

Let us design the class that will be used to represent image maps. An image map is a list of coordinates, so we make the class a subclass of the STL type `list<COORD>`. In addition, we define a member function

```
void add(COORD coordArray[]);
```

to allow us to add an array of coordinates to the list. We mark the end of the coordinate array by storing a `COORD` value of (-1, -1) as the last element of the array.

Here is a preliminary declaration of an `ImageMap` class:

```
class ImageMap: list<COORD>
{
public:
    // Add an array of coordinates to the image map
    void add(COORD coordArray[])
```

```
        {
            for(int k = 0; coordArray[k].X != -1; k++)
            {
               push_back(coordArray[k]);
            }
        }
    }
```

As an example, the line from (0, 0) to (3, 3) would be represented by the code

```
ImageMap line;
COORD lineCoords[] = {{0,0}, {1,1}, {2,2}, {3,3}, {-1,-1}};
line.add(lineCoords);
```

Initializing an array of coordinates in this manner and then adding it to the image map is very handy, and is a vast improvement over the alternative of using `push_back` to add the coordinates to the image map one at a time:

```
ImageMap line;
COORD pos;
pos.X = 0;
pos.Y = 0;
line.push_back(pos);
pos.X = 1;
pos.Y = 1;
line.push_back(pos);
// rest of the code is omitted
```

The braces { } that go around a single COORD object in the initialization of the `lineCoords` array are tedious to insert, particularly when the array has a lot of elements. We will therefore consider an alternative notation for initializing the image map. The alternative will allow us to use an array of short integers to initialize an array of coordinates.

An array of two short integers and a `COORD` object both consist of two short integers and C++ compilers store both in memory the *same way*. Once stored in memory, an array of `5` `COORD` objects is indistinguishable from an array of 10 short integers. If we write

```
short int lineShorts[] = {0, 0, 1, 1, 2, 2, 3, 3, -1, -1};
```

then the array `lineShorts` is indistinguishable from `lineCoords` in the way the two arrays are stored in memory. We can use this fact to find an alternative way of initializing image maps that does not require as many braces. We add a second `add` member function to `ImageMap`, one that takes an array of `short int` as a parameter. The new `add` function uses a cast to convert its parameter to an array of COORD, and then calls the first `add` function.

```
void add(short *coordAsShorts)
{
    COORD *pCoord = reinterpret_cast<COORD *>(coordAsShorts);
    add(pCoord);
}
```

It will help you to understand why this code works if you remember that an array of `COORD`, which is what the member function `add(COORD arr[])` expects, has the same type as a pointer to `COORD`.

Basics of Animation

Now consider a video game in which a person has to run across the screen. The effect of running will be achieved by creating image maps of a person in successive running position as in Figure 17-17.

Figure 17-17



The first image is displayed briefly at a certain position and then erased. Next, the second image is briefly displayed a little to the right of the first position and then erased. By successively displaying and erasing a progression of images at a series of positions in left to right order, we obtain the appearance of a person running.

Implementation Details

Once an `ImageMap` object is created, a programmer can use its `add` methods to incrementally build the list of coordinates that comprises the shape. Starting with an empty image map, the programmer can initialize arrays of short integers to represent different parts of the human body. In this way, arms, legs, torsos, and other parts of the body can be represented and added to the image map to form the shape of a complete person. Two additional methods,

```
void displayAt(char ch, int col, int row);
void displayAt(int col, int row);
```

can be used to display the image map's shape at a given position. The first of the two methods specifies a fill character to be used for the outline and interior of the shape. The second is a convenience method—it calls the first display method and passes it the asterisk as fill character. Finally, the method

```
void eraseAt(int col, int row);
```

is used to erase the image map's shape at a specified position. The full implementation of the ImageMap class, and an illustration of its use to achieve graphics animation, are shown in the listings that follow.

Contents of `Imagemap.h`

```
1 #include <iostream>
2 #include <list>
3 #include <windows.h>
4 using namespace std;
5
6 const HANDLE console = GetStdHandle(STD_OUTPUT_HANDLE);
```

```
7
8  class ImageMap:list<COORD>{
9
10 public:
11     // Add an array of coordinates to the image map
12     void add(COORD coordArray[]);
13     // Convenience method for adding an array of coordinates
14     void add(short *coordAsShorts);
15     // Display a given character at a specified position
16     void displayAt(char ch, int col, int row);
17     // Display an asterisk at a given position
18     void displayAt(int col, int row)
19     {
20         displayAt('*', col, row);
21     }
22     // Erase whatever character is at a given position
23     void eraseAt(int col, int row)
24     {
25         displayAt(' ', col, row);
26     }
27 };
```

Contents of `Imagemap.cpp`

```
1  #include "ImageMap.h"
2
3  //************************************************
4  // Adds an array of coordinates to the image map   *
5  //************************************************
6  void ImageMap::add(COORD coordArray[])
7   {
8     for(int k = 0; coordArray[k].X != -1; k++)
9     {
10       push_back(coordArray[k]);
11    }
12 }
13
14 //************************************************
15 // Allows an array of shorts to be converted to     *
16 // an array of COORD. That simplifies the           *
17 // initialization process for an image              *
18 //************************************************
19  void ImageMap::add(short *coordAsShorts)
20  {
21     COORD *pCoord = reinterpret_cast<COORD *>(coordAsShorts);
22     add(pCoord);
23  }
24
25  //*****************************************************
26  // Shows an image at a given position. The image is    *
27  // is drawn using the character ch                     *
28  //*****************************************************
29  void ImageMap::displayAt(char ch, int col, int row)
30  {
31     list<COORD>::iterator iter = this->begin();
32     for (; iter != this->end(); iter++)
```

```
33      {
34          COORD currentPos;
35          currentPos.Y = row + iter->Y;
36          currentPos.X = col + iter->X;
37          SetConsoleCursorPosition(console, currentPos);
38          cout << ch << endl;
39      }
40  }
```

Program 17-11

```
1  // This program illustrates animation using the
2  // ImageMap class.
3  #include "ImageMap.h"
4
5  int main()
6  {
7       // Figure 1 - a snapshot of a person running
8       ImageMap figure1;
9
10      // Set up the coordinates for the various body parts
11      // of the person in the first running position
12      short int lowerLeg1[] = { 1, 10, 2, 10, 3, 10, -1, -1};
13      short int thigh1[] = { 4, 9, 5, 8, 6, 7, 7, 6, -1, -1};
14      short int thigh2[] = { 6, 7, 7, 8, 8, 9, -1, -1};
15      short int lowerLeg2[] = {8, 10, 8, 11, -1, -1};
16      short int torso[] = { 8, 5, 9, 4, 10, 3, 11, 2, -1, -1};
17      short int upperArms[] = { 7, 2, 8, 3, 9,
18                                4, 10, 5, 11, 6, -1, -1
19                              };
20      short int foreArm1[] = { 12, 5, 13, 4, -1, -1};
21      short int foreArm2[] = {6, 3, 5, 4, -1, -1};
22      short int  * figure1AllParts [ ] =
23                   {
24                    lowerLeg1, lowerLeg2, thigh1, thigh2, torso,
25                    upperArms, foreArm1, foreArm2, 0
26                   };
27      // Add the coordinates that make up the various body
28      // parts to the image map for the first running position
29      int k = 0;
30      for (int k = 0; figure1AllParts[k] != 0; k++)
31          figure1.add(figure1AllParts[k]);
32
33      // Figure 2- a snapshot of the person in a
34      //  different running position
35      ImageMap figure2;
36      short int p2LowerLeg1[] = {1, 11, 2, 10, 3, 9, -1, -1};
37      short int p2thigh1[] = {3, 9, 3, 8, 3, 7, -1, -1};
38      short int p2thigh2[] = {4, 7, 5, 7, 6, 7, -1, -1};
39      short int p2LowerLeg2[] = {6, 8, 6, 9, -1, -1};
```

(program continues)

Program 17-11 *(continued)*

```
40      short int p2torso[] = {3, 6, 3, 5, 3, 4, 3, 3,
41                             3, 2, 3, 1, -1, -1
42                            };
43      short int p2UpperArms[] = {1, 3, 2, 3, 4, 3, 5, 3, -1, -1};
44      short int p2foreArm1[] = { 1, 4, 1, 5, -1, -1};
45      short int p2foreArm2[] = { 5, 2, 5, 1, -1, -1};
46      short int *figure2AllParts[] =
47                {
48                  p2LowerLeg1, p2thigh1, p2thigh2, p2LowerLeg2,
49                  p2torso, p2UpperArms, p2foreArm1, p2foreArm2, 0
50                 };
51      for (int k = 0; figure2AllParts[k] != 0; k++)
52          figure2.add(figure2AllParts[k]);
53
54      // Figure 3- a snapshot of a person in
55      // yet another running position
56      ImageMap figure3;
57      short int p3torso[] = {4, 7, 4, 6, 4, 5, 4, 4,
58                             4, 3, 4, 2, 4, 1, -1, -1
59                            };
60      short int p3Thigh1[] = {5, 8, 6, 9, -1, -1};
61      short int p3Thigh2[] = {3, 8, 2, 9,  -1, -1};
62      short int p3LowerLeg1[] = {6, 10, 6, 11, -1, -1};
63      short int p3LowerLeg2[] = {1, 8, 0, 7, -1, -1};
64      short int p3UpperArm1[] = {3, 4, 2, 5, -1, -1};
65      short int p3UpperArm2[] = {5, 4, 6, 5, -1, -1};
66      short int p3ForeArm1[] = {3, 6, 4, 7, -1, -1};
67      short int p3ForeArm2[] = {7, 4, 8, 3, -1, -1};
68      short int * figure3AllParts[] =
69                  {
70                    p3torso, p3Thigh1, p3Thigh2, p3LowerLeg1,
71                    p3LowerLeg2,p3UpperArm1, p3UpperArm2,
72                    p3ForeArm1, p3ForeArm2, 0
73                  };
74
75      for (int k = 0; figure3AllParts[k] != 0; k++)
76          figure3.add(figure3AllParts[k]);
77
78      // Ask Microsoft Windows to clear the screen
79      system("cls");
80      // Form an array of all three figures
81      ImageMap *sequence[3] = {&figure1, &figure2, &figure3};
82
83      // Animate to create the appearance of
84      // running across the screen
85      k = 0;
86      int pos = 0;
87      while (pos <= 60 )
88      {
89          // Show the current image at the current position
90          sequence[k]->displayAt(pos, 3);
91          Sleep(400);
```

(program continues)

Program 17-11 *(continued)*

```
92          // Erase the current image
93          sequence[k]->eraseAt(pos, 3);
94          // Move to next image in the rotation and next position
95          k = (k+1) % 3;
96          pos = pos + 8;
97       }
98       sequence[k]->displayAt(pos, 3);
99       return 0;
100 }
```

Review Questions and Exercises

Fill-in-the-Blank

1. The _____ points to the first node in a linked list.
2. A data structure that points to an object of the same type as itself is known as a(n) _____ data structure.
3. To indicate that a linked list is empty, you should set the pointer to its head to the value _____.
4. _____ a node means adding it to the end of a list.
5. _____ a node means adding it to a list, but not necessarily to the end.
6. _____ a list means traveling through the list.
7. In a(n) _____ list, the last node has a pointer to the first node.
8. In a(n) _____ list, each node has a pointer to the one before it and the one after it.

Algorithm Workbench

9. Using the `ListNode` structure introduced in this chapter, write a function

   ```
   void printFirst(ListNode *ptr)
   ```

   that prints the value stored in the first node of a list passed to it as parameter. The function should print an error message and terminate the program if the list passed to it is empty.

10. Write a function

    ```
    void printSecond(ListNode *ptr)
    ```

    that prints the value stored in the second node of a list passed to it as parameter. The function should print an error message and terminate the program if the list passed to it has less than two nodes.

11. Write a function

    ```
    double lastValue(ListNode *ptr)
    ```

    that returns the value stored in the last node of a nonempty list passed to it as parameter. The function should print an error message and terminate the program if the list passed to it is empty.

12. Write a function

```
ListNode *removeFirst(ListNode *ptr)
```

that is passed a linked list as parameter, and returns the tail of the list: that is, it removes the first node and returns what is left. The function should deallocate the storage of the removed node. The function returns NULL if the list passed to it is empty.

13. Write a function

```
ListNode *ListConcat(ListNode *list1, ListNode *list2)
```

That concatenates the items in `list2` to the end of `list1` and returns the resulting list.

Predict the Output

For each of the following program fragments, predict what the output will be.

14.
```
ListNode *p = new ListNode(56.4);
p = new ListNode(34.2, p);
cout << (*p).value <<  endl << p->value;
```

15.
```
ListNode *p = new ListNode(56.4);
p = new ListNode(34.2, p);
ListNode *q = p->next;
cout << q->value;
```

16.
```
ListNode *p = new ListNode(56.4, new ListNode(31.5));
ListNode *q = p;
while (q->next->next != NULL)
q = q->next;
cout << q->value;
```

Find the Errors

17. Each of the following member functions for performing an operation on a linked list of type `NumberList` has at least one error. Explain what is wrong and how to fix it.

A)
```
NumberList::printList( )
{
   while(head)
   {
      cout << head->value;
      head = head->next;
   }
}
```

B)
```
NumberList::printList( )
{
   ListNode *p = head;
   while (p->next)
   {
      cout << p->value;
      p = p->next;
   }
}
```

C)
```
NumberList::printList( )
{
   ListNode *p = head;
   while(p)
   {
      cout << p->value;
      p++;
   }
}
```

D)
```
NumberList::~NumberList()
{
   ListNode *nodePtr, *nextNode;

   nodePtr = head;
   while (nodePtr != NULL)
   {
      nextNode = nodePtr->next;
      nodePtr->next = NULL;
      nodePtr = nextNode;
   }
}
```

Soft Skills

18. You are the leader of a programming team. You want the programmers on your team to attend a two-day workshop on linked lists, stacks and queues. One of the managers points out that the STL already supplies each one of those data structures, making it unnecessary for your programmers to write their own. Write the manager a short memo that justifies the need for the workshop.

Programming Challenges

myCodeMate

1. Simple Linked List Class

Using an appropriate definition of `ListNode`, design a simple linked list class with only two member functions and a default constructor:

```
void add(double x);
boolean isMember(double x);
LinkedList( );
```

The add function adds a new node containing `x` to the front (head) of the list, while the `isMember` function tests to see if the list contains a node with the value `x`. Test your linked list class by adding various numbers to the list and then testing for membership.

2. List Copy Constructor

Modify your list class of Programming Challenge 1 to add a copy constructor. Test your class by making a copy of a list and then testing membership on the copy.

myCodeMate

3. List Print

Modify the list class you created in the previous programming challenges to add a `print` member function. Test the class by starting with an empty list, adding some elements, and then printing the resulting list out.

4. Recursive Member Check

Modify the list class you created in the previous programming challenges to use a recursive method to check for list membership. Test your class.

5. List Member Deletion

Modify the list class you created in the previous programming challenges by adding a function to remove an item from the list, and by adding a destructor:

```
void remove(double x);
~LinkedList();
```

Test the class by adding by a sequence of instructions that mixes operations for adding items, removing items, and printing the list.

6. List Reverse

Modify the list class you created in the previous programming challenges by adding a member function for reversing the list:

```
void reverse();
```

The member function rearranges the nodes in the list so that their order is reversed. You should do this without creating or destroying nodes.



7. List Search

Modify the list class of Programming Challenge 1 (or later) to include a member function

```
int search(double x)
```

that returns the position of a number `x` on the list. The first node in the list is at position 0, the second node is at position 1, and so on. If `x` is not found on the list, the search should return -1. Test the new member function using an appropriate driver program.

8. Member Insertion By Position



VideoNote
Solving the Member Insertion By Position Problem

Modify the list class you created in the previous programming challenges by adding a member function for inserting a new item at a specified position:

```
void insert(double x, int pos);
```

A position of `0` means that `x` will become the first item on the list, a position of 1 means that `x` will become the second item on the list, and so on. A position equal to, or greater than, the length of the list means that the `x` is placed at the end of the list.

9. Member Removal by Position

Modify the list class you created in the previous programming challenges by adding a member function for deleting a node at a specified position:

```
void remove(int pos);
```

A value of 0 for the position means that the first node on the list (the current head) is deleted. The function does nothing if the value passed for `pos` is greater or equal to the length of the list.

10. List Sort

Modify the list class you created in the previous programming challenges by adding a member function that will sort the list into ascending order by the numeric value of the item stored in the node.

```
void sort(  );
```

You should sort the list by moving pointers rather than by copying or swapping the contents of the nodes.

11. List Template

Program 17-10 destroys linked lists in two different functions: in the destructor for `NumberList` and in the `destroyList` method of the `ReliableNumberList` class. Improve the program by eliminating this duplication of code.

12. Running Back

Program 17-11 makes a person run from across the screen, starting near the left edge of the screen and ending near the right edge. Modify the program so that the person turns around and runs back to the starting point.

CHAPTER 18

Stacks and Queues

TOPICS

18.1 Introduction to the Stack ADT
18.2 Dynamic Stacks
18.3 The STL `stack` Container
18.4 Introduction to the Queue ADT
18.5 Dynamic Queues
18.6 The STL `deque` and `queue` Containers
18.7 Focus on Problem Solving and Program Design: *Eliminating Recursion*
18.8 Tying It All Together: *Converting Postfix Expressions to Infix*

18.1 Introduction to the Stack ADT

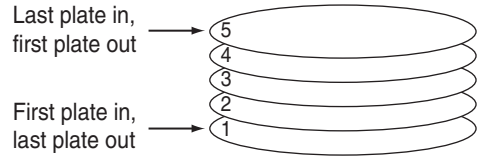
CONCEPT: **A stack is a data structure that stores and retrieves items in a last-in-first-out manner.**

Definition

Like an array or a linked list, a stack is a data structure that holds a sequence of elements. Unlike arrays and lists, however, stacks are *last-in-first-out (LIFO)* structures. This means that when a program retrieves elements from a stack, the last element inserted into the stack is the first one retrieved (and likewise, the first element inserted is the last one retrieved).

When visualizing the way a stack works, think of a stack of plates at the beginning of a cafeteria line. When a cafeteria worker replenishes the supply of plates, the first one he or she puts on the stack is the last one taken off. This is illustrated in Figure 18-1.

Figure 18-1



The LIFO characteristic of a stack of plates in a cafeteria is also the primary characteristic of a stack data structure. The last data element placed on the stack is the first data retrieved from the stack.

Applications of Stacks

Stacks are useful data structures for algorithms that work first with the last saved element of a series. For example, computer systems use stacks while executing programs. When a function is called, they save the program's return address on a stack. They also create local variables on a stack. When the function terminates, the local variables are removed from the stack and the return address is retrieved. Also, some calculators use a stack for performing mathematical operations.

Static and Dynamic Stacks

There are two types of stack data structure: static and dynamic. Static stacks have a fixed size and are implemented as arrays. Dynamic stacks grow in size as needed and are implemented as linked lists. In this section you will see examples of both static and dynamic stacks.

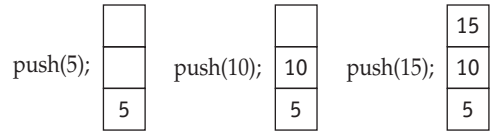
Stack Operations

A stack has two primary operations: *push* and *pop*. The push operation causes a value to be stored, or pushed onto the stack. For example, suppose we have an empty integer stack that is capable of holding a maximum of three values. With that stack we execute the following push operations.

```
push(5);
push(10);
push(15);
```

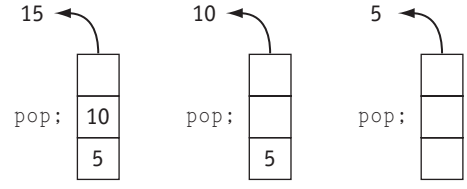
Figure 18-2 illustrates the state of the stack after each of these push operations.

Figure 18-2



The pop operation retrieves (and hence, removes) a value from the stack. Suppose we execute three consecutive pop operations on the stack shown in Figure 18-2. Figure 18-3 depicts the results.

Figure 18-3



As you can see from Figure 18-3, the last pop operation leaves the stack empty.

For a static stack (one with a fixed size), we will need a boolean *isFull* operation. The isFull operation returns true if the stack is full, and false otherwise. This operation is necessary to prevent a stack overflow in the event a push operation is attempted when all the stack's elements have values stored in them.

For both static and dynamic stacks we will need a boolean *isEmpty* operation. The isEmpty operation returns true when the stack is empty, and false otherwise. This prevents an error from occurring when a pop operation is attempted on an empty stack.

A Static Stack Class

Now we examine a class `IntStack` that stores a static stack of integers and performs the stack operations we have discussed. The class has the member variables described in Table 18-1.

Table 18-1 Members Variables of the Stack Class

Member Variable	Description
`stackArray`	A pointer to `int`. When the constructor is executed, it uses `stackArray` to dynamically allocate an array for storage.
`capacity`	An integer that holds the size of the stack. This is the maximum number of elements the stack can hold, not the number of elements currently in the stack.
`top`	An integer that is used to mark the top of the stack. It specifies the position of the next item that will be added to the stack.

The class's member functions are listed in Table 18-2.

Table 18-2 Members Functions of the Stack Class

Member Functions	Description
Constructor	The class constructor accepts an integer argument, which specifies the size of the stack. An integer array of this size is dynamically allocated and assigned to `stackArray`. Also, the variable `top` is initialized to 0 to indicate that the stack is currently empty.
`push`	The `push` function accepts an integer argument, which is pushed onto the top of the stack.
`pop`	The `pop` function uses an integer reference parameter. The value at the top of the stack is removed and copied into the reference parameter.
`isEmpty`	Returns `true` if the stack is empty, and `false` otherwise. The stack is empty when `top` is set to 0.



NOTE: Even though the constructor dynamically allocates the stack array, it is still considered a static stack since the size of the stack does not change once it is allocated.

The code for the class is shown here:

Contents of `IntStack.h`

```
 1 class IntStack
 2 {
 3 private:
 4      int *stackArray;
 5      int capacity;
 6      int top;
 7 public:
 8      IntStack(int capacity);  // Constructor
 9      ~IntStack() { delete[] stackArray; }
10      void push(int value);
11      void pop(int &value);
12      bool isEmpty();
13
14      // Stack Exceptions
15      class Overflow {};
16      class Underflow {};
17 };
```

In addition to the members of the stack described in Table 18-2, the `IntStack` class defines two inner classes named `Overflow` and `Underflow` to be used as stack exceptions. Exceptions are covered in Chapter 16, but we will briefly explain them here for the benefit of those who may have skipped that chapter. A section of code is said to cause an *exception* when, in the course of execution, it encounters conditions that make it impossible to perform the task the code was designed to do. In the case of a static stack, an *overflow* exception occurs during a call to `push` if there is no more room on the stack. Likewise, an *underflow* exception occurs in a call to `pop` if there is nothing on the stack for `pop` to return.

Code that detects the occurrence of an exception can notify the rest of the program by creating a value that describes the exception and passing that value to the rest of the program

using a `throw` statement. For example, the `push` function announces the occurrence of an overflow exception by executing the statement

```
throw InstStack::Overflow();
```

and the `pop` function executes the statement

```
throw IntStack::Underflow();
```

to notify the program that the underflow exception has occurred. By default, a program terminates with an error message when any part of it throws an exception. This default behavior can be changed through a process known as catching the exception. You can learn more about exceptions in Chapter 16.

The `IntStack` constructor allocates an array of a specified capacity and sets the member variable `top` to 0. All stack functions use `top` in such a way that it always points to the next available slot in the stack's array. When `top` equals `capacity`, there are no more slots available to store values, and the next call to `push` throws an exception. Likewise, when `top` is zero, the stack is empty and a call to `pop` throws an exception. Because there are no provisions in the program to catch either exception, the occurrence of either one will terminate the program with an error message. Notice that `push` increments `top` after adding a value to the stack, and `pop` decrements `top` before returning the value stored at `stackArray[top]`.

Contents of `IntStack.cpp`

```
#include "intstack.h"
//***********************************
// Constructor                      *
//***********************************
IntStack::IntStack(int capacity)
{
     this->capacity = capacity;
     stackArray = new int[capacity];
     top = 0;
}

//***********************************
// Adds a value to the stack        *
//***********************************
void IntStack::push(int value)
{
     if (top == capacity) throw IntStack::Overflow();
     stackArray[top] = value;
     top++;
}

//***************************************
// Determines whether the stack is empty *
//***************************************
bool IntStack::isEmpty()
{
     if (top == 0)
         return true;
     else
         return false;
}
```

```
32
33 //*********************************************
34 // Removes a value from the stack and returns it *
35 //*********************************************
36 void IntStack::pop(int &value)
37 {
38      if (isEmpty()) throw IntStack::Underflow();
39      top --;
40      value = stackArray[top];
41 }
```

Program 18-1 illustrates the stack class and its member functions. Notice that the values pushed onto the stack come off in reverse order when they are popped.

Program 18-1

```
1  // This program illustrates the IntStack class.
2  #include "intstack.h"
3  #include <iostream>
4  using namespace std;
5  int main()
6  {
7      IntStack  stack(5);
8      int values[] = {5, 10, 15, 20, 25};
9      int value;
10
11     cout << "Pushing...\n";
12     for (int k = 0; k < 5; k++)
13     {
14         cout << values[k] << "  ";
15         stack.push(values[k]);
16     }
17     cout << "\nPopping...\n";
18     while (!stack.isEmpty())
19     {
20         stack.pop(value);
21         cout << value << "  ";
22     }
23     cout << endl;
24     return 0;
25 }
```

Program Output

```
Pushing...
5  10  15  20  25
Popping...
25  20  15  10  5
```

In Program 18-1, the constructor is called with the argument 5. This sets up the member variables, as shown in Figure 18-4. Since `top` is set to 0, the stack is empty.

Figure 18-4

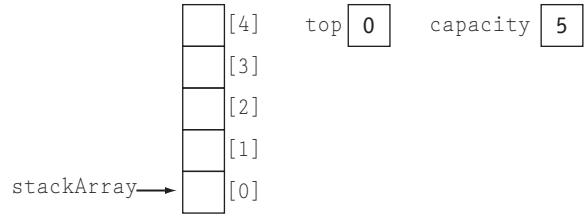


Figure 18-5 shows the state of the member variables after the `push` function is called the first time (with 5 as its argument). The value of `top` is now 1.

Figure 18-5

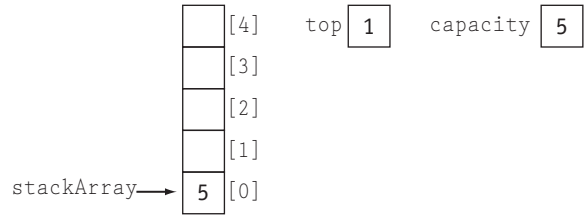
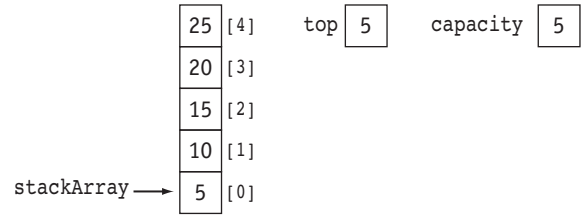


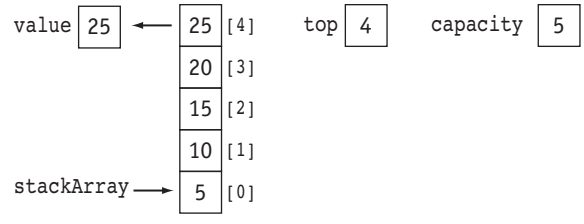
Figure 18-6 shows the state of the member variables after all five calls to the `push` function. Now `top` has value 5, and the stack is full.

Figure 18-6



Notice that the `pop` function uses a reference parameter, `value`. The value that is popped off the stack is copied into `value` so it can be used later in the program. Figure 18-7 depicts the state of the class members and the `value` parameter, just after the first value is popped off the stack.

Figure 18-7



The program continues to call the `pop` function until all the values have been removed from the stack.

Handling Stack Exceptions

As you learned in Chapter 16, the C++ `try/catch` statement can be used to catch and recover from exceptions, thereby allowing the program to avoid being terminated. The following program shows how a program using the `IntStack` class can catch the exceptions that it throws. The program tries to store in the stack more values than the stack can handle, causing push to throw the `Overflow` exception. The main function catches the exception and prints an explanatory error message.

Program 18-2

```
1  // This program illustrates IntStack exception handling.
2  #include "intstack.h"
3  #include <iostream>
4  using namespace std;
5  int main()
6  {
7      IntStack  stack(5);
8      int values[] = {5, 10, 15, 20, 25};
9      int value;
10     try
11     {
12         cout << "Pushing...\n";
13         for (int k = 0; k < 5; k++)
14         {
15             cout << values[k] << "  ";
16             stack.push(values[k]);
17         }
18         cout << "\nPushing value after stack is full..";
19         stack.push(30);
20         cout << "\nYou should not see this!!";
21         cout << endl;
22     }
23     catch(IntStack::Overflow)
24     {
25         cout << "\nAn Overflow exception occurred.\n";
26     }
27     return 0;
28 }
```

Program Output

```
Pushing...
5  10  15  20  25
Pushing value after stack is full..
An Overflow exception occurred.
```

There is a significant difference between a stack filling up and a stack becoming empty, and between the stack overflow and stack underflow exceptions. In stack overflow, a program has a value to push on a stack, but cannot continue execution because the stack is full. There is a sense in which overflow is unexpected, because a program does not normally expect to use up every slot in the stack. On the other hand, programs are usually written to remove and process all items stored on a stack, so they do expect that the stack will eventually become empty. Indeed, most algorithms that use a stack have a loop that continues to iterate as long as the stack is not empty. This is why a stack always needs an `isEmpty`

function, but does not need an `isFull` function. A well-written stack-based algorithm will normally call `isEmpty` to make sure the stack is not empty before calling `pop`.

The difference between overflow and underflow can be summarized as follows. Stack overflow in `push` notifies the caller that the stack has run out of resources, while stack underflow notifies the caller of an error in the program's logic. Unlike overflow, stack underflow can be avoided by careful programming. Therefore, programs should not use `try/catch` to handle underflow. Instead, they should ensure that underflow cannot occur by calling `isEmpty` before calling `pop`.

Stack Templates

The stack classes shown in this chapter work only with integers. A stack template can be easily designed to work with any data type. This is left as a Programming Challenge for you to complete.

18.2 Dynamic Stacks

CONCEPT: **A stack may be implemented as a linked list, and expand or shrink with each push or pop operation.**

A dynamic stack is built on a linked list instead of on an array. A stack based on a linked list offers two advantages over a stack based on an array. First, there is no need to specify the starting size of the stack. A dynamic stack simply starts as an empty linked list, then expands by one node each time a value is pushed. Second, a dynamic stack will never be full, as long as the system has enough free memory.

In this section we will look at a dynamic stack class, `DynIntStack`. This class is a dynamic version of the `IntStack` class previously discussed. The class declaration is shown here:

Contents of `DynIntStack.h`

```
1  #ifndef DYNINTSTACK_H
2  #define DYNINTSTACK_H
3
4  class DynIntStack
5  {
6  private:
7     class StackNode
8     {
9        friend class DynIntStack;
10       int value;
11       StackNode *next;
12       // Constructor.
13       StackNode(int value1, StackNode *next1 = NULL)
14        {
15          value = value1;
16          next = next1;
17        }
18    };
19    StackNode *top;
20 public:
21    DynIntStack() { top = NULL; }
```

```
   void push(int);
   void pop(int &);
   bool isEmpty();
};
#endif
```

The `StackNode` class is the data type of each node in the linked list. Because it is easy to add and remove items at the beginning of the list, we make the beginning of the linked list the *top* of the stack, and use a pointer `top` to point to the first node in the list. This pointer is initialized to NULL by the stack constructor, to signify that the stack is created empty.

The member functions of this stack class are shown here:

Contents of `DynIntStack.cpp`

```
#include <iostream>
#include "DynIntStack.h"
using namespace std;

//************************************************
// Member function push pushes the argument onto   *
// the stack.                                       *
//************************************************
void DynIntStack::push(int num)
{
   top = new StackNode(num, top);
}

//************************************************
// Member function pop removes the value at the top   *
// of the stack and copies it into the variable       *
// passed as an argument.                             *
//************************************************
void DynIntStack::pop(int &num)
{
   StackNode *temp;

   if (isEmpty())
   {
      cout << "The stack is empty.\n";
      exit(1);
   }
   else   // Pop value off top of stack
   {
      num = top->value;
      temp = top;
      top = top->next;
      delete temp;
   }
}

//************************************************
// Member function isEmpty returns true if the stack  *
// is empty, or false otherwise.                      *
//************************************************
```

```
41 bool DynIntStack::isEmpty()
42 {
43    if (!top)
44        return true;
45    else
46        return false;
47 }
```

The `push` function is particularly simple. It simply creates a new node whose value is the number to be pushed on the stack, and whose successor pointer is the node that is currently the top of the stack, and then makes the newly created node the new top of the stack:

```
top = new StackNode(num, top);
```

Note that this works correctly even if the stack was empty previous to the push operation, because in that case the successor to the new node at the top of the stack will be correctly set to NULL.

Now let's look at the `pop` function. Just as the `push` function must insert nodes at the head of the list, `pop` must delete nodes at the head of the list. First, the function calls `isEmpty` to determine whether there are any nodes in the stack. If not, an error message is displayed and the program is terminated.

```
if (isEmpty())
{
   cout << "The stack is empty.\n";
   exit(1);
}
```

If `isEmpty` returns `false`, then the following statements are executed.

```
else   // Pop value off top of stack
{
   num = top->value;
   temp = top;
   top = top->next;
   delete temp;
}
```

First, a copy of the `value` member of the node at the top of the stack is saved in the `num` reference parameter. A temporary pointer `temp` is then set to point to the node that is to be deleted, that is, the node currently at the top of the stack. The `top` pointer is then set to point to the node after the one that is currently at the top. The same code will set `top` to NULL if there are no nodes after the one that is currently at the top of the stack. It is then safe to delete the top node through the temporary pointer.

Program 18-3 is a driver that demostrates the `DynIntStack` class.

Program 18-3

```
1 // This program demonstrates the dynamic stack
2 // class DynIntStack.
3 #include <iostream>
4 #include "DynIntStack.h"
5 using namespace std;
6
```

(program continues)

Program 18-3 *(continued)*

```
 7 int main()
 8 {
 9    DynIntStack stack;
10    int catchVar;
11
12    cout << "Pushing 5\n";
13    stack.push(5);
14    cout << "Pushing 10\n";
15    stack.push(10);
16    cout << "Pushing 15\n";
17    stack.push(15);
18
19    cout << "Popping...\n";
20    stack.pop(catchVar);
21    cout << catchVar << endl;
22    stack.pop(catchVar);
23    cout << catchVar << endl;
24    stack.pop(catchVar);
25    cout << catchVar << endl;
26
27    cout << "\nAttempting to pop again... ";
28    stack.pop(catchVar);
29    return 0;
30 }
```

Program Output

```
Pushing 5
Pushing 10
Pushing 15
Popping...
15
10
5

Attempting to pop again... The stack is empty.
```

18.3 The STL `stack` Container*

CONCEPT: The Standard Template Library offers a stack template, that may be implemented as a `vector`, a `list`, or a `deque`.

So far, the STL containers you have learned about are `vectors` and `lists`. The STL `stack` container may be implemented as a `vector` or a `list`. (It may also be implemented as a `deque`, which you will learn about later in this chapter.) One class is said to *adapt* another class if it provides a new interface for it. The purpose of the new interface is to make it

* This section should be skipped if Chapter 16 has not yet been covered.

more convenient to use the class for specialized tasks. Because the stack container is used to adapt the list, vector, and deque containers, it is often referred to as a *container adapter.*



VideoNote
Storing Objects in an STL Stack

Here are examples of how to declare a stack of ints, implemented as a vector, a list, and a deque.

```
stack< int, vector<int> > iStack; // Vector stack
stack< int, list<int> > iStack    // List stack
stack< int > iStack;              // Deque stack (the default)
```



NOTE: Be sure to put spaces between the angled brackets that appear next to each other. This will prevent the compiler from mistaking >> for the stream extraction operator, >>.

Table 18-3 lists and describes some of the stack template's member functions.

Table 18-3 STL Stack Member Fiunctions

Member Function	Examples and Description
empty	`if (myStack.empty())` The `empty` member function returns `true` if the stack is empty. It returns `false` if the stack has elements.
pop	`myStack.pop();` The pop function removes the element at the top of the stack
push	`myStack.push(x);` The `push` function pushes an element with the value `x` onto the stack.
size	`cout << myStack.size() << endl;` The `size` function returns the number of elements currently in the stack.
top	`x = myStack.top();` The `top` function returns a reference to the element at the top of the stack.



NOTE: The pop function in the stack template does not retrieve the value from the top of the stack, it only removes it. To retrieve the value, you must call the `top` function first.

Program 18-4 is a driver that demonstrates an STL stack implemented as a vector.

Program 18-4

```
1 // This program demonstrates the STL stack
2 // container adapter.
3 #include <iostream>
4 #include <vector>
5 #include <stack>
6 using namespace std;
7
```

(program continues)

Program 18-4 *(continued)*

```
 8 int main()
 9 {
10    stack< int, vector<int> > iStack;
11
12    for (int x = 2; x < 8; x += 2)
13    {
14       cout << "Pushing " << x << endl;
15       iStack.push(x);
16    }
17
18    cout << "The size of the stack is ";
19    cout << iStack.size() << endl;
20
21    // Print items and pop until the stack is empty.
22    while (!iStack.empty())
23    {
24       cout << "Popping " << iStack.top() << endl;
25       iStack.pop();
26    }
27    return 0;
28 }
```

Program Output

```
Pushing 2
Pushing 4
Pushing 6
The size of the stack is 3
Popping 6
Popping 4
Popping 2
```



Checkpoint

18.1 Describe what LIFO means.

18.2 What is the difference between static and dynamic stacks? What advantages do dynamic stacks have over static stacks?

18.3 What are the two primary stack operations? Describe them both.

18.4 What STL types does the STL stack container adapt?

18.4 Introduction to the Queue ADT

CONCEPT: **A queue is a data structure that stores and retrieves items in a first-in-first-out manner.**

Definition

Like a stack, a *queue* (pronounced "cue") is a data structure that holds a sequence of elements. A queue, however, provides access to its elements in *first-in, first-out (FIFO)*

order. The elements in a queue are processed like customers standing in a grocery checkout line: The first customer in line is the first one served.

Application of Queues

Queue data structures are commonly used in computer operating systems. They are especially important in multiuser/multitasking environments where several users or tasks may be requesting the same resource simultaneously. Printing, for example, is controlled by a queue because only one document may be printed at a time. A queue is used to hold print jobs submitted by users of the system, while the printer services those jobs one at a time.

Communications software also uses queues to hold information received over networks and dial-up connections. Sometimes information is transmitted to a system faster than it can be processed, so it is placed in a queue when it is received.

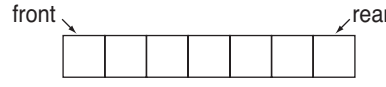
Static and Dynamic Queues

Queues, like stacks, can be implemented as arrays or linked lists. Dynamic queues offer the same advantages over static queues that dynamic stacks offer over static stacks. In fact, the primary difference between queues and stacks is the way data elements are accessed in each structure.

Queue Operations

A queue has a front and a rear like a checkout line in a grocery store. This is illustrated in Figure 18-8. When an element is added to a queue, it is added to the rear. When an element is removed from a queue, it is removed from the front. The two primary queue operations are enqueuing and dequeuing. To *enqueue* means to insert an element at the rear of a queue, and to *dequeue* means to remove an element from the front of a queue. There are several algorithms for implementing these operations. We will begin by looking at the simplest.

Figure 18-8

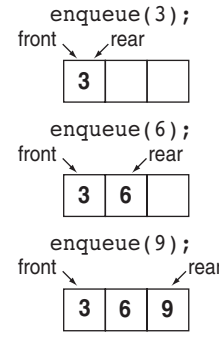


Suppose we have an empty static integer queue that is capable of holding a maximum of three values. With that queue we execute the following enqueue operations:

```
enqueue(3);
enqueue(6);
enqueue(9);
```

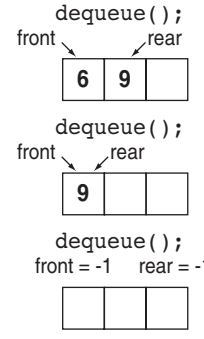
Figure 18-9 illustrates the state of the queue after each of these enqueue operations.

Figure 18-9



Notice that the front index (which is a variable holding a subscript or perhaps a pointer) always references the same physical element. The rear index moves in the array as items are enqueued. Now let's see how dequeue operations are performed. Figure 18-10 illustrates the state of the queue after each of three consecutive dequeue operations.

Figure 18-10



In the dequeuing operation, the element at the front of the queue is removed. This is done by moving all the elements after it forward by one position. After the first dequeue operation, the value 3 is removed from the queue and the value 6 is at the front. After the second dequeue operation, the value 6 is removed and the value 9 is at the front. Notice that when only one value is stored in the queue, that value is at both the front and the rear.

When the last dequeue operation is performed in Figure 18-10, the queue is empty. An empty queue can be signified by setting both front and rear indices to –1.

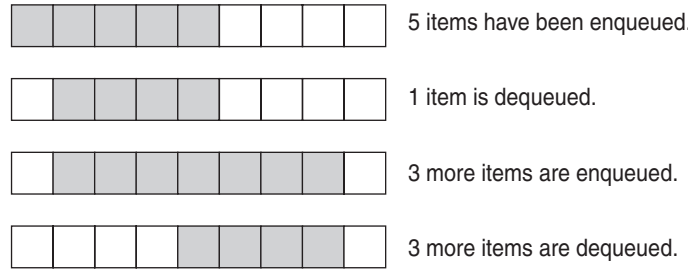
The problem with this algorithm is its inefficiency. Each time an item is dequeued, the remaining items in the queue are copied forward to their neighboring element. The more items there are in the queue, the longer each successive dequeue operation will take.

Here is one way to overcome the problem: Make both the front and rear indices move in the array. As before, when an item is enqueued, the rear index is moved to make room for it. But in this design, when an item is dequeued, the front index moves by one element

toward the rear of the queue. This logically removes the front item from the queue and eleminates the need to copy the remaining items to their neighboring elements.

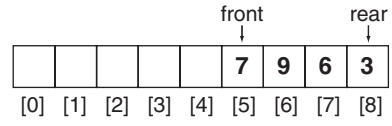
With this approach, as items are added and removed, the queue gradually "crawls" toward the end of the array. This is illustrated in Figure 18-11. The shaded squares represent the queue elements (between the front and rear).

Figure 18-11



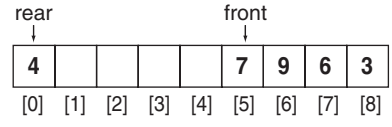
The problem with this approach is that the rear index cannot move beyond the last element in the array. The solution is to think of the array as circular instead of linear. When an item moves past the end of a circular array, it simply wraps around to the beginning. For example, consider the queue depicted in Figure 18-12.

Figure 18-12



The value 3 is at the rear of the queue, and the value 7 is at the front of the queue. Now, suppose an enqueue operation is performed, inserting the value 4 into the queue. Figure 18-13 shows how the rear of the queue wraps around to the beginning of the array.

Figure 18-13



So, what is the code for wrapping the rear marker around to the opposite end of the array? One straightforward approach is to use an `if` statement such as

```
if (rear == queueSize - 1)
   rear = 0;
else
   rear++;
```

Another approach is with modular arithmetic:

```
rear = (rear + 1) % queueSize;
```

This statement uses the `%` operator to adjust the value in `rear` to the proper position. Although this approach appears more elegant, the choice of which code to use is yours.

Detecting Full and Empty Queues with Circular Arrays

In our implementation of a queue using a circular array, we have adopted the convention that the front and rear indices both reference items that are *still in the queue*, and that the front and rear indices will both be set to –1 to indicate an *empty* queue. To preserve this convention, the operation for dequeueing an element must set both `front` and `rear` to –1 after removing an element from a queue with only one item. The dequeuing operation can test for a queue with only one item by testing whether `front` is equal to `rear`. To avoid overflowing the queue, the operation for enqueuing must first check that the queue is not already full before adding another element. We can check to see if the queue is full by testing the expression

```
(rear + 1) % queueSize == front
```

to see if it is true.

There is another way for detecting full and empty queues: A counter variable can be used to keep a count of the number of items currently stored in the queue. With this convention, the counter is incremented with each enqueue operation and decremented with each dequeue operation. The queue is empty when the counter is zero, and is full when the counter equals the size allocated for the queue.

Because it might be helpful to keep a count of items in the queue anyway, we will use the second method in our implementation. Accordingly, we introduce the variables

```
int *queueArray;
int queueSize;
int front;
int rear;
int numItems;
```

with `numItems` being the counter variable, and `queueArray` the pointer to a dynamically allocated array of size `queueSize`. We adopt the following two conventions:

- `rear` points to the place in the queue holding the item that was last added to the queue.
- `front` points to the place in the queue that used to hold the item that was last removed from the queue.

Because of the convention on where the rear index is pointing to, the enqueue operation must first (circularly) move `rear` one place to the right before adding a new item `num`:

```
rear =  (rear + 1) % queueSize;
queueArray[rear] = num;
numItems ++;
```

Similarly, because whatever is at `front` has already been removed, the dequeue operation must first move `front` before retrieving a queue item.

A Static Queue Class

The declaration of the `IntQueue` class is as follows:

Contents of `IntQueue.h`

```
1  #ifndef INTQUEUE_H
2  #define INTQUEUE_H
3
4  class IntQueue
5  {
6  private:
7     int *queueArray;
8     int queueSize;
9     int front;
10    int rear;
11    int numItems;
12 public:
13    IntQueue(int);
14    ~IntQueue();
15    void enqueue(int);
16    void dequeue(int &);
17    bool isEmpty();
18    bool isFull();
19    void clear();
20 };
21 #endif
```

Notice that in addition to the operations discussed in this section, the class also declares a member function named `clear`. This function clears the queue by resetting the `front` and `rear` indices, and setting the `numItems` member to 0. The member function definitions are listed here:

Contents of `IntQueue.cpp`

```
1  #include <iostream>
2  #include "IntQueue.h"
3  using namespace std;
4
5  //************************
6  // Constructor.          *
7  //************************
8  IntQueue::IntQueue(int s)
9  {
10    queueArray = new int[s];
11    queueSize = s;
12    front = -1;
13    rear = -1;
14    numItems = 0;
15 }
16
17 //************************
18 // Destructor.           *
19 //************************
20 IntQueue::~IntQueue()
21 {
22    delete [] queueArray;
23 }
```

```
24
25 //*****************************************
26 // Function enqueue inserts the value in num *
27 // at the rear of the queue.                 *
28 //*****************************************
29 void IntQueue::enqueue(int num)
30 {
31    if (isFull())
32       {
33          cout << "The queue is full.\n";
34          exit(1);
35       }
36    else
37    {
38       // Calculate the new rear position.
39       rear = (rear + 1) % queueSize;
40       // Insert new item.
41       queueArray[rear] = num;
42       // Update item count.
43       numItems++;
44    }
45 }
46
47 //*****************************************
48 // Function dequeue removes the value at the   *
49 // front of the queue, and copies it into num. *
50 //*****************************************
51 void IntQueue::dequeue(int &num)
52 {
53    if (isEmpty())
54    {
55       cout << "The queue is empty.\n";
56       exit(1);
57    }
58    else
59    {
60       // Move front.
61       front = (front + 1) % queueSize;
62       // Retrieve the front item.
63       num = queueArray[front];
64       // Update item count.
65       numItems--;
66    }
67 }
68
69 //*****************************************
70 // Function isEmpty returns true if the queue *
71 // is empty, and false otherwise.             *
72 //*****************************************
73 bool IntQueue::isEmpty()
74 {
75    if (numItems > 0)
76       return false;
77    else
78       return true;
79 }
```

```
80
81 //*****************************************
82 // Function isFull returns true if the queue *
83 // is full, and false otherwise.             *
84 //*****************************************
85 bool IntQueue::isFull()
86 {
87    if (numItems < queueSize)
88       return false;
89    else
90       return true;
91 }
92
93 //*****************************************
94 // Function clear resets the front and rear *
95 // indices, and sets numItems to 0.         *
96 //*****************************************
97 void IntQueue::clear()
98 {
99    front =  - 1;
100   rear =  - 1;
101   numItems = 0;
102 }
```

Program 18-5 is a driver that demonstrates the `IntQueue` class.

Program 18-5

```
1  // This program demonstrates the IntQueue class
2  #include <iostream>
3  #include "IntQueue.h"
4  using namespace std;
5
6  int main()
7  {
8     IntQueue iQueue(5);
9
10    cout << "Enqueuing 5 items...\n";
11
12    // Enqueue 5 items.
13    for (int k = 1; k <= 5; k++)
14       iQueue.enqueue(k*k);
15
16    // Deqeue and retrieve all items in the queue.
17    cout << "The values in the queue were: ";
18    while (!iQueue.isEmpty())
19    {
20       int value;
21       iQueue.dequeue(value);
22       cout << value << "  ";
23    }
24    cout << endl;
25    return 0;
26 }
```

(program continues)

Program 18-5 *(continued)*

Program Output

```
Enqueuing 5 items...
The values in the queue were: 1  4  9  16  25
```

Overflow and Underflow Exceptions in a Static Queue

The `enqueue` and `dequeue` functions in our queue class terminate the calling program when they cannot perform the task they are called to do. But terminating the caller is not always the right thing to do. A better course of action is to throw an exception and allow the caller who is prepared to handle such an exception to take appropriate action. Upon catching such an exception, some callers may indeed decide to terminate the program. Other callers, however, may be able to recover and continue execution. For example, a program that catches a queue overflow exception might be able to create a bigger queue and switch to the new queue.

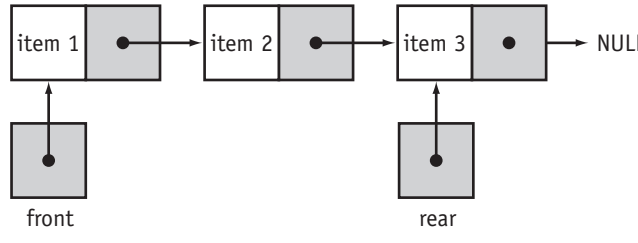
A better design for a static queue is to have `enqueue` and `dequeue` throw `overflow` and `underflow` exceptions. Having `enqueue` throw `overflow` eliminates the need for a public `isFull` function because the caller can use a `try/catch` block to handle queue overflows if and when they occur. By putting all calls to `enqueue` within the try block, the caller is able to put the code to handle an exception thrown by any of those calls in a single place: the catch block. Without exception handling, every call to `enqueue` would have to be preceded by a call to `isFull` and have code attached to it to recover in the event that `isFull` returns true. One of the programming challenges at the end of this chapter asks you to modify the queue class to use exceptions.

18.5 Dynamic Queues

CONCEPT: A queue may be implemented as a linked list, and expand or shrink with each enqueue or dequeue operation.

Dynamic queues, which are built around linked lists, are much more intuitive to understand than static queues. A dynamic queue starts as an empty linked list. With the first enqueue operation, a node is added, which is pointed to by the `front` and `rear` pointers. As each new item is added to the queue, a new node is added to the rear of the list, and the `rear` pointer is updated to point to the new node. As each item is dequeued, front is made to point to the next mode in the list, and then the node that was previously at the front of the list is deleted. Figure 18-14 shows the structure of a dynamic queue.

Figure 18-14



A dynamic integer queue class is listed here:

Contents of DynIntQueue.h

```
#ifndef DYNINTQUEUE_H
#define DYNINTQUEUE_H

class DynIntQueue
{
private:
   class QueueNode
   {
      friend class DynIntQueue;
      int value;
      QueueNode *next;
      QueueNode(int value1, QueueNode *next1 = NULL)
      {
         value = value1;
         next = next1;
      }
   };
   // These track the front and rear of the queue.
   QueueNode *front;
   QueueNode *rear;
public:
   // Constructor and Destructor.
   DynIntQueue();
   ~DynIntQueue();

   // Member functions.
   void enqueue(int);
   void dequeue(int &);
   bool isEmpty();
   void clear();
};
#endif
```

Contents of DynIntQueue.cpp

```
#include <iostream>
#include "DynIntQueue.h"
using namespace std;

//************************
// Constructor.          *
//************************
DynIntQueue::DynIntQueue()
{
   front = NULL;
   rear = NULL;
}

//************************
// Destructor.           *
//************************
```

```
17 DynIntQueue::~DynIntQueue()
18 {
19    clear();
20 }
21
22 //*****************************************
23 // Function enqueue inserts the value in num *
24 // at the rear of the queue.                 *
25 //*****************************************
26 void DynIntQueue::enqueue(int num)
27 {
28    if (isEmpty())
29    {
30       front = new QueueNode(num);
31       rear = front;
32    }
33    else
34    {
35       rear->next = new QueueNode(num);
36       rear = rear->next;
37    }
38 }
39
40 //*****************************************
41 // Function dequeue removes the value at the   *
42 // front of the queue, and copies it into num. *
43 //*****************************************
44 void DynIntQueue::dequeue(int &num)
45 {
46    QueueNode *temp;
47    if (isEmpty())
48    {
49       cout << "The queue is empty.\n";
50       exit(1);
51    }
52    else
53    {
54       num = front->value;
55       temp = front;
56       front = front->next;
57       delete temp;
58    }
59 }
60
61 //*****************************************
62 // Function isEmpty returns true if the queue *
63 // is empty, and false otherwise.             *
64 //*****************************************
65 bool DynIntQueue::isEmpty()
66 {
67    if (front == NULL)
68       return true;
69    else
70       return false;
71 }
72
```

```
73 //*****************************************
74 // Function clear dequeues all the elements  *
75 // in the queue.                             *
76 //*****************************************
77 void DynIntQueue::clear()
78 {
79    int value;   // Dummy variable for dequeue
80
81    while(!isEmpty())
82       dequeue(value);
83 }
```

Program 18-6 is a driver that demonstrates the `DynIntQueue` class.

Program 18-6

```
1  // This program demonstrates the DynIntQeue class.
2  #include <iostream>
3  #include "DynIntQueue.h"
4  using namespace std;
5
6  int main()
7  {
8     DynIntQueue iQueue;
9
10    cout << "Enqueuing 5 items...\n";
11
12    // Enqueue 5 items.
13    for (int k = 1; k < = 5; k++)
14       iQueue.enqueue(k*k);
15
16    // Dequeue and retrieve all items in the queue.
17    cout << "The values in the queue were:\n";
18    while (!iQueue.isEmpty())
19    {
20       int value;
21       iQueue.dequeue(value);
22       cout << value << "  ";
23    }
24    return 0;
25 }
```

Program Ouput

```
Enqueuing 5 items...
The values in the queue were:
1  4  9  16  25
```

18.6 The STL deque and queue Containers*

CONCEPT: The Standard Template Library provides two containers, deque and queue, for implementing queue-like data structures.

In this section we will examine two ADTs offered by the Standard Template Library: deque and queue. A deque (pronounced "deck" or "deek") is a double-ended queue. It similar to a vector, but allows efficient access to values at both the front and the rear. The queue ADT is like the stack ADT: It is actually a container adapter.

The deque Container

Think of the deque container as a vector that provides quick access to the element at its front as well as at the back. (Like vector, deque also provides access to its elements with the [] operator.)

Programs that use the deque ADT must include the deque header. Since we are concentrating on its queue-like characteristics, we will focus our attention on the push_back, pop_front, and front member functions. Table 18-4 describes them.

Table 18-4 deque Member Functions

Member Function	Examples and Description
push_back	`iDeque.push_back(7);` Accepts as an argument a value to be inserted into the deque. The argument is inserted after the last element (pushed onto the back of the deque).
pop_front	`iDeque.pop_front();` Removes the first element of the deque and discards it.
front	`cout << iDeque.front() << endl;` front returns a reference to the first element of the deque.

Program 18-7 demonstrates the deque container.

Program 18-7

```
1 // This program demonstrates the STL deque
2 // container.
3 #include <iostream>
4 #include <deque>
5 using namespace std;
6
```

(program continues)

* Note: This section should be skipped if Chapter 16 has not yet been covered.

Program 18-7 *(continued)*

```
7  int main()
8  {
9     deque<int> iDeque;
10
11    cout << "I will now enqueue items...\n";
12    for (int x = 2; x < 8; x += 2)
13    {
14       cout << "Pushing " << x << endl;
15       iDeque.push_back(x);
16    }
17
18    cout << "I will now dequeue items...\n";
19    while (!iDeque.empty())
20    {
21       cout << "Popping " << iDeque.front() << endl;
22       iDeque.pop_front();
23    }
24    return 0;
25 }
```

Program Output

```
I will now enqueue items...
Pushing 2
Pushing 4
Pushing 6
I will now dequeue items...
Popping 2
Popping 4
Popping 6
```

The queue Container Adapter



VideoNote
Storing Objects in an STL Queue

The queue container adapter can be built upon vectors, lists, or deques. By default, it uses a deque as its base.

The insertion and removal operations supported by queue are similar to those supported by the stack ADT: push, pop, and front. There are differences in their behavior, however. The queue version of push always inserts an element at the rear of the queue. The queue version of pop always removes an element from the structure's front. The front function returns the value of the element at the front of the queue.

Program 18-8 demonstrates a queue. Since the declaration of the queue does not specify which type of container is being adapted, the queue will be built on a deque.

Program 18-8

```
1  // This program demonstrates the STL queue
2  // container adapter.
3  #include <iostream>
4  #include <queue>
5  using namespace std;
6
```

(program continues)

Program 18-8 *(continued)*

```
7  int main()
8  {
9     queue<int> iQueue;
10
11    cout << "I will now enqueue items...\n";
12    for (int x = 2; x < 8; x += 2)
13    {
14       cout << "Pushing " << x << endl;
15       iQueue.push(x);
16    }
17    cout << "I will now dequeue items...\n";
18    while(!iQueue.empty())
19    {
20       cout << "Popping " << iQueue.front() << endl;
21       iQueue.pop();
22    }
23    return 0;
24 }
```

Program Output

```
I will now enqueue items...
Pushing 2
Pushing 4
Pushing 6
I will now dequeue items...
Popping 2
Popping 4
Popping 6
```

18.7 Focus on Problem Solving and Program Design: *Eliminating Recursion*

Although recursion is a very useful programming technique, it carries the overhead of the necessity to make numerous function calls during the process of solving the problem. The efficiency of a recursive solution can often be greatly improved by reformulating a recursive algorithm to eliminate the recursion. In this section, we look at how a stack can be used to eliminate recursion from the Quicksort algorithm.

The main problem in Quicksort is that of sorting a *range*, or a segment of an array `arr`, between two indices `start` and `end`. Naturally, this has to be done only if `start` is less than `end`. As learned in Chapter 14, this is accomplished by calling a procedure `partition`, which determines an integer `pivot` such that

1. All array items in the segment to the left of the pivot are less than the element at the pivot: that is,

   `arr[k] < arr[pivot]` for all `k` in the range `start .. pivot-1`

2. All array items in the segment to the right of the pivot are greater than or equal to the element at the pivot: that is,

   `arr[k] >= arr[pivot]` for all `k` in the range `pivot+1..end`

Once this is done, the array item at the pivot is in its sorted position. Thus an important effect of the `partition` procedure is that it gets the pivot element in its final sorted position. By keeping track of the left and right subranges when we call `partition`, and then later calling `partition` on those subranges, we can sort the entire array without using recursion. We need to keep track of these subranges, and eventually partition them in the order in which the recursive calls to Quicksort would have done them. Because the recursive calls to Quicksort are invoked and return in LIFO (last-in-first-out) order, we use a stack to keep track of the ranges that are waiting to be partitioned.

The main idea of our solution is to define a class

```
class Range
{
   int start;
   int end;
public:
   Range(int s, int e)
   {
      start = s;
      end = e;
   }
};
```

to keep track of the ranges of the array that remain to be partitioned. Accordingly, we use the STL stack class to define a stack of these ranges:

```
stack<Range> qStack;
```

We then use a function `qSort(int arr[ ], int size)` that sorts the array `arr` by initially pushing the range from `0` to `size-1` onto the stack, and then repeatedly removing ranges from the stack, partitioning the range, and putting the left and right subrange back onto the stack. Empty subranges removed from the stack are discarded. The algorithm is

```
push Range(0, size-1) onto stack
While stack not empty
   pop  a range r from the stack
   If r is not empty
      partition the range r into two smaller ranges about the pivot
      push the two smaller ranges onto the stack
   End if
End While
```

The complete solution, which reuses the partition function from Chapter 14, is given in Program 18-9. Notice that we declare the `qSort` function to be a friend of `Range`, to allow access to the private members of `Range`.



NOTE: The statement `qStack.push(Range(0, size-1));` creates a `Range` object by invoking the constructor. The `Range` object is then pushed onto the stack.

Program 18-9

```
// This program illustrates the use of a stack to
// implement a nonrecursive quicksort.
#include <stack>
#include <iostream>
#include <fstream>
#include <algorithm>  // needed for swap
using namespace std;

// Function prototypes.
void qSort(int a[ ], int size);
void outputArray(int a[ ], int size);
int partition(int a[ ], int, int);

// Range is used to indicate a segment
// of an array that is still to be sorted.
class Range
{
   // Make qSort a friend.
   friend void qSort(int a[], int);
   int start;
   int end;
public:
   Range(int s, int e)
   {
     start = s;
     end = e;
   }
};

const int MAX = 100;
int main()
{
  ifstream inputFile;
  string filename = "sort.dat" ;
    int array[MAX];
  int size;
  inputFile.open(filename.data());
  if (!inputFile)
    {
      cout << "The file  " << filename << " cannot be "
           << "opened .";
      exit(1);
    }

   // Read the file and count the number of items in the
   // file. Take care not to overrun the array.
   size = 0;
   while (inputFile >> array[size])
   {
      size ++;
      if (size == MAX)
         break;
   }
```

(program continues)

Program 18-9 *(continued)*

```
54    // Echo the inputted array.
55    cout << "The original array is :" << endl;
56    outputArray(array, size);
57
58    // Perform the sort and output the result.
59    qSort(array, size);
60    cout << "The sorted array is: " << endl;
61    outputArray(array, size);
62    return 0;
63 }
64
65 //*******************************************************
66 // qSort performs a nonrecursive quicksort              *
67 // on the array a[ ] of the given size                  *
68 //*******************************************************
69 void qSort(int arr[ ], int size)
70 {
71   // qStack holds segments of the array that have not
72   // yet been sorted.
73   stack<Range> qStack;
74   int pivot, start, end;
75
76   qStack.push(Range(0, size-1));
77   // As long as there is a range waiting to be sorted,
78   // take it off the stack, partition it, and then
79   // put the resulting two smaller ranges onto the stack.
80
81   while (!qStack.empty())
82   {
83      Range currentRange = qStack.top();
84      qStack.pop();
85
86      // Get the endpoints of the current Range
87      // and partition it.
88
89      start = currentRange.start;
90      end = currentRange.end;
91      if (start < end)
92        {
93           pivot = partition(arr, start, end);
94           // Store the resulting smaller ranges for later
95           // processing.
96           qStack.push(Range(start, pivot-1));
97           qStack.push(Range(pivot + 1, end));
98        }
99   }
100 }
101
102 //*******************************************************
103 // partition rearranges the entries in the array arr from   *
104 // start to end so all values greater than or equal to the  *
105 // pivot are on the right of the pivot and all values less  *
106 // than are on the left of the pivot.                       *
107 //*******************************************************
```

(program continues)

Program 18-9 *(continued)*

```
108 int partition(int arr[], int start, int end)
109 {
110    // The pivot element is taken to be the element at
111    // the start of the subrange to be partitioned.
112    int pivotValue = arr[start];
113    int pivotPosition = start;
114
115    // Rearrange the rest of the array elements to
116    // partition the subrange from start to end.
117    for (int pos = start + 1; pos <= end; pos++)
118    {
119       if (arr[pos] < pivotValue)
120       {
121          // arr[scan] is the "current" item.
122          // Swap the current item with the item to the
123          // right of the pivot element.
124          swap(arr[pivotPosition + 1], arr[pos]);
125          // Swap the current item with the pivot element.
126          swap(arr[pivotPosition], arr[pivotPosition + 1]);
127          // Adjust the pivot position so it stays with the
128          // pivot element.
129          pivotPosition ++;
130       }
131    }
132    return pivotPosition;
133 }
134
135 //*****************************************************
136 // Output an array's elements.                        *
137 //*****************************************************
138 void outputArray(int arr[ ], int size)
139 {
140    for (int k = 0; k < size; k++)
141      cout << arr[k] << "  ";
142    cout << endl;
143 }
```

Program Output

```
The original array is :
34  -45  78  32  90  45
The sorted array is:
-45  32  34  45  78  90
```



NOTE: The friend concept should be used with caution, since it circumvents the protection afforded the members of the class by declaring them private. Notice that in our case, the `start` and `end` members of the `Range` class are never modified by the friend function `qSort`.

18.8 Tying It All Together: *Converting Postfix Expressions to Infix*

Stacks can be used to evaluate postfix expressions. Let's see how this can be done. We confine ourselves to postfix expressions that contain only numbers and the binary operators +, -, *, and /.

Recall from Chapter 14 that a postfix expression is either a single number or two postfix expressions followed by an operator. Evaluation of a single-number postfix expression is easy: we just return the number. For the nonsimple case, we must evaluate the two postfix expressions in order and save their values. Then, when we come to the operator, we retrieve the two previously saved values and apply the operator.

To see how the method works consider the example

2 5 -

Because 2 and 5 are single-number postfix expressions, we simply save their values for later use. Then, when we encounter the minus operator, we retrieve the two saved values and apply the operator, yielding -3 as the value of the entire expression. In general, any postfix expression can be evaluated by reading it in left to right order. Whenever a value is encountered, it is pushed onto the stack to await application by an operator at a later stage. Whenever an operator is encountered, its two operands are popped off the stack, and the operator is applied to them to yield a value. This value is in turn pushed onto the stack. The procedure ends when all of the input expression has been read. At that time, there should be only one value on the stack. The value on the stack is the value of the postfix expression.

This same idea can be used to convert postfix expressions to infix. Again, we read the input postfix expression from left to right. This time, though, we use a stack of strings instead of a stack of integer. Any number that is read must be an operand: it is converted to a string and pushed onto the stack. If an operator is encountered, the two strings at the top of the stack are popped and the operator is placed between them. Parentheses are then placed around the resulting string, and the parenthesized string is pushed back onto the stack. Thus, for example, the above input postfix expression would result on the following sequence of pushes of strings onto the stack:

```
"2"
"2"     "5"
"(2 – 5)"
```

These ideas are used in Program 18–10, which follows.

Program 18-10

```
1 // This program converts postfix expressions to infix.
2 #include <string>
3 #include <iostream>
4 #include <sstream>
```

(program continues)

Program 18-10 *(continued)*

```
#include <stack>

using namespace std;

string postfixExpr(istream & inputStream);

int main()
{
    string input;
    cout << "Enter a postfix expression to convert to infix,"
         << " \nor a blank line to quit the program:";
    getline(cin, input);
    while (input.size() != 0)
    {
        // Convert string to a string stream
        istringstream inputExpr(input);
        cout << "The infix equivalent  is "
             << postfixExpr(inputExpr) << endl;
        cout << "Enter a postfix expression to evaluate: ";
        getline(cin, input);
    }
    return 0;
}

//*************************************************************
// Takes an istream that contains a single postfix expression p *
// and returns a string representing the infix equivalent of p  *
//*************************************************************
string postfixExpr(istream & in)
{
   // Holds intermediate values in computation
   stack<string> infixStack;
   // Used to read characters in the expression
   char ch;
   // Used to read numbers in the expression
   int number;
   // Used to remove infix expressions from the stack
   string lExpr, rExpr;

   ch = in.peek();
   while (ch != EOF)
   {
       // If we have a whitespace character skip it and
       // continue with the next iteration of this loop
       if (isspace(ch))
       {
           ch = in.get();
           ch = in.peek();
           continue;  // Go back to top of loop
       }
```

(program continues)

Program 18-10 *(continued)*

```
55          // Nonspace character is next in input stream
56          // If the next character is a number, read it, convert
57          // to string, and push the string onto the infix stack
58          if (isdigit(ch))
59          {
60             in >> number;
61             // Use to convert number to string
62             ostringstream numberStr;
63             // Convert number to string using stream
64             numberStr << number;
65             // Push the string representing the expression onto the stack
66             infixStack.push(numberStr.str());
67             ch = in.peek();
68             continue;
69          }
70          // If the next character is an operator,
71          // pop the two top infix expresssions stored on the
72          // stack, put the operator between the two infix expressions,
73          // and then push the result on the stack
74
75          rExpr = infixStack.top();
76          infixStack.pop();
77          lExpr = infixStack.top();
78          infixStack.pop();
79          if (ch == '+' || ch == '-' || + ch == '*' || ch == '/')
80              infixStack.push("(" + lExpr + " " + ch + " " + rExpr + ")");
81          else
82          {
83              cout << "Error in the input expression" << endl;
84              exit(1);
85          }
86          ch = in.get();  // Actually read the operator character
87          ch = in.peek(); // Prepare for the next iteration of the loop
88       }
89       return infixStack.top();
90    }
```

Program Output with Example Input Shown in Bold

```
Enter a postfix expression to convert to infix,
or a blank line to quit the program: 56
The infix equivalent  is 56
Enter a postfix expression to evaluate: 56 2 +
The infix equivalent  is (56 + 2)
Enter a postfix expression to evaluate: 56 2 + 12 9 - *
The infix equivalent  is ((56 + 2) * (12 - 9))
Enter a postfix expression to evaluate: [Enter]
```

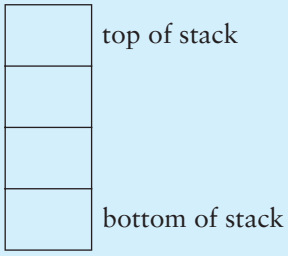
Review Questions and Exercises

Short Answer

1. What does LIFO mean?
2. What element is retrieved from a stack by the pop operation?
3. What is the difference between a static stack and a dynamic stack?
4. Describe two operations that all stacks perform.
5. The STL `stack` is considered a container adapter. What does that mean?
6. What types may the STL `stack` be based on? By default, what type is an STL `stack` based on?
7. What does FIFO mean?
8. When an element is added to a queue, where is it added?
9. When an element is removed from a queue, where is it removed from?
10. Describe two operations that all queues perform.
11. What two queue-like containers does the STL offer?
12. Suppose the following operations were performed on an empty stack:

```
push(0);
push(9);
push(12);
push(1);
```

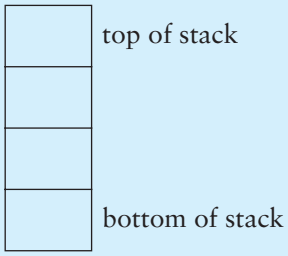
Insert numbers in the following diagram to show what will be stored in the static stack after the operations have executed.



13. Suppose the following operations were performed on an empty stack:

```
push(8);
push(7);
pop();
push(19);
push(21);
pop();
```

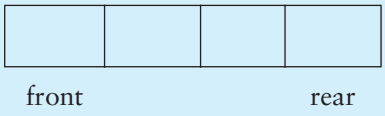
Insert numbers in the following diagram to show what will be stored in the static stack after the operations have executed.



14. Suppose the following operations are performed on an empty queue:

```
enqueue(5);
enqueue(7);
enqueue(9);
enqueue(12);
```

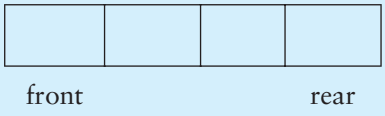
Insert numbers in the following diagram to show what will be stored in the static queue after the operations have executed.



15. Suppose the following operations are performed on an empty queue:

```
enqueue(5);
enqueue(7);
dequeue();
enqueue(9);
enqueue(12);
dequeue();
enqueue(10);
```

Insert numbers in the following diagram to show what will be stored in the static queue after the operations have executed.



16. What problem is overcome by using a circular array for a static queue?

Algorithm Workbench

17. Give pseudocode that implements a queue using two stacks. The queue operations *enqueue*, *dequeue*, and *empty* must be implemented in terms of the *push*, *pop*, and *empty* stack operations.

Soft Skills

18. A common real-life example used to explain stacks is the stack of plates in a cafeteria. Find at least two other real-life examples in which items are added and removed from a container in last-in-first-out order, and use these examples to explain the concept of a stack.

Programming Challenges



1. Static Stack Template

In this chapter you studied `IntStack`, a class that implements a static stack of integers. Write a template that will create a static stack of any data type. Demonstrate the class with a driver program.

2. Dynamic Stack Template

In this chapter you studied `DynIntStack`, a class that implements a dynamic stack of integers. Write a template that will create a dynamic stack of any data type. Demonstrate the class with a driver program.

3. Static Queue Template

In this chapter you studied `IntQueue`, a class that implements a static queue of integers. Write a template that will create a static queue of any data type. Demonstrate the class with a driver program.

4. Dynamic Queue Template

In this chapter you studied `DynIntQueue`, a class that implements a dynamic queue of integers. Write a template that will create a dynamic queue of any data type. Demonstrate the class with a driver program.

5. Error Testing

The `DynIntStack` and `DynIntQueue` classes shown in this chapter are abstract data types using a dynamic stack and dynamic queue, respectively. The classes do not currently test for memory allocaton errors. Modify the classes so they determine if new nodes cannot be created, and handle the error condition in an appropriate way. (You will need to catch the predefined exception `bad_alloc`.)



NOTE: If you have already done Programming Challenges 2 and 4, modify the templates you created.

6. Dynamic String Queue

Design a class that stores strings on a dynamic queue. The strings should not be fixed in length. Demonstrate the class with a driver program.

7. Queue Exceptions

Modify the static queue class used in Program 18-5 as follows.

1. Make the `isFull` member private.
2. Define a queue *overflow* exception and modify `enqueue` so that it throws this exception when the queue runs out of space.
3. Define a queue *underflow* exception and modify `dequeue` so that it throws this exception when the queue is empty.
4. Rewrite the main program so that it catches overflow exceptions when they occur. The exception handler for queue overflow should print an appropriate error message and then terminate the program.

8. Evaluating Postfix Expressions

Write a program that reads postfix expressions and prints their values. Each input expression should be entered on its own line, and the program should terminate when the user enters a blank line.

Assume only binary operators, and that the expressions contain no variables. Note that you will need to use parentheses to indicate the order of application of the operators in the expression. Here are sample input–output pairs:

```
78                                    78
78 6 +                                84
78 6 +  9  2 -  /                     12
```



VideoNote
Solving the File Reverser Problem

9. File Reverser

Write a program that opens a text file and reads its contents into a stack of characters. The program should then pop the characters from the stack and save them in a second text file. The order of the characters saved in the second file should be the reverse of their order in the first file.



10. Balanced Parentheses

A string of characters has balanced parentheses if each right parentheses occurring in the string is matched with a preceding left parentheses in the same way each right brace in a C++ program is matched with a preceding left brace. Write a program that uses a stack to determine whether a string entered at the keyboard has balanced parentheses.

11. Balanced Multiple Delimiters

A string may use more than one type of delimiter to bracket information into "blocks." For example, A string may use braces { }, parentheses ( ), and brackets [ ] as delimiters. A string is properly delimited if each right delimiter is matched with a preceding left delimiter of the same type in such a way that the either the resulting blocks of information are disjoint, or one of them is completely nested within the other. Write a program that uses a single stack to check whether a string containing braces, parentheses, and brackets is properly delimited.

12. Inventory Bin Stack

Design an inventory class that stores the following members.

`serialNum`: an integer that holds a part's serial number
`manufactDate`: a member that holds the date the part was manufactured
`lotNum`: an integer that holds the part's lot number

The class should have appropriate member functions for storing data into, and retrieving data from, these members.

Next, design a stack class that can hold objects of the class. If you wish, you may use the template you designed in Programming Challenge 1 or 2.

Last, design a program that uses the stack class. The program should have a loop that asks the user whether he or she wishes to add a part to inventory or take a part from inventory. The loop should repeat until the user is finished.

If the user wishes to add a part to inventory, the program should ask for the serial number, date of manufacture, and lot number. The information should be stored in an inventory object and pushed onto the stack.

If the user wishes to take a part from inventory, the program should pop the topmost part from the stack and display the contents of its member variables.

When the user finishes, the program should display the contents of the member values of all the objects that remain on the stack

13. Inventory Bin Queue

Modify the program you wrote for Programming Challenge 12 so it uses a queue instead of a stack. Compare the order in which the parts are removed from the bin for each program.

CHAPTER 19

Binary Trees

TOPICS

19.1 Definition and Applications of Binary Trees
19.2 Binary Search Tree Operations
19.3 Template Considerations for Binary Search Trees
19.4 Tying It All Together: *Genealogy Trees*

19.1 Definition and Applications of Binary Trees

CONCEPT: Binary trees differ from linked lists in that where a node in a linked list may have at most one successor, a node in a binary tree can have up to two successors.

A *binary tree* is a collection of nodes in which each node is associated with up to two successor nodes, respectively called the *left* and *right child*. Not every node in the binary tree will have two children: one or both nodes may be omitted. A node in a binary tree that has no children is called a *leaf node*.

A node that has children is said to be the *parent* of its children. For a nonempty collection of nodes to qualify as a binary tree, every node must have at most one parent, and there must be exactly one node with no parent. The one node that has no parent is called the *root* of the binary tree. An empty collection of nodes is regarded as constituting an empty binary tree.

There is some similarity between a linked list and a binary tree. The root of a binary tree corresponds to the head of a list, a child of a binary tree node corresponds to a successor node in a list, and the parent of a binary tree node corresponds to the *predecessor* of a node in the list. And of course, the analog of the empty list is the empty binary tree.

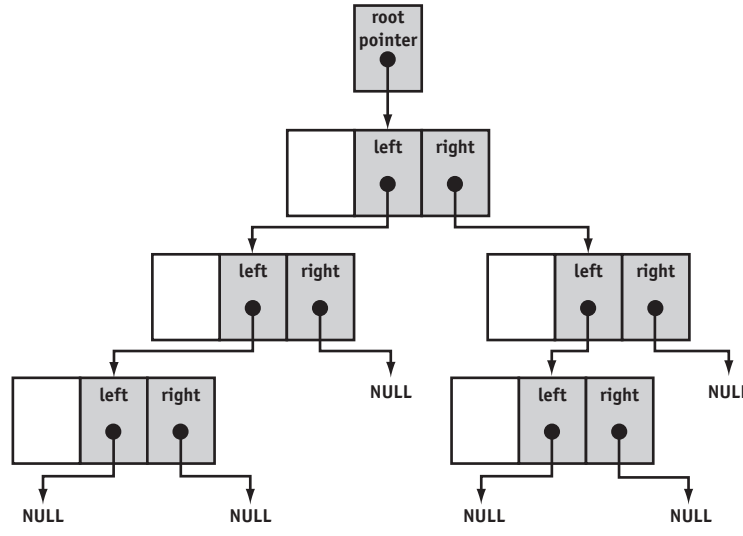
Implementation of Binary Trees

Binary trees are used to store values in their nodes. A node in a binary tree will therefore be a structure or class object that contains a member for storing the value, as well as two members that point to nodes that are the left and right children of that node:

```
struct TreeNode
{
   int value;
   TreeNode *left;
   TreeNode *right;
};
```

A binary tree is itself represented by a pointer to the node that is the root of the tree. An example binary tree, with the values stored in the nodes not shown, is illustrated in Figure 19-1. The `left` or `right` pointer in a node is set to NULL if that node does not possess the corresponding child.

Figure 19-1

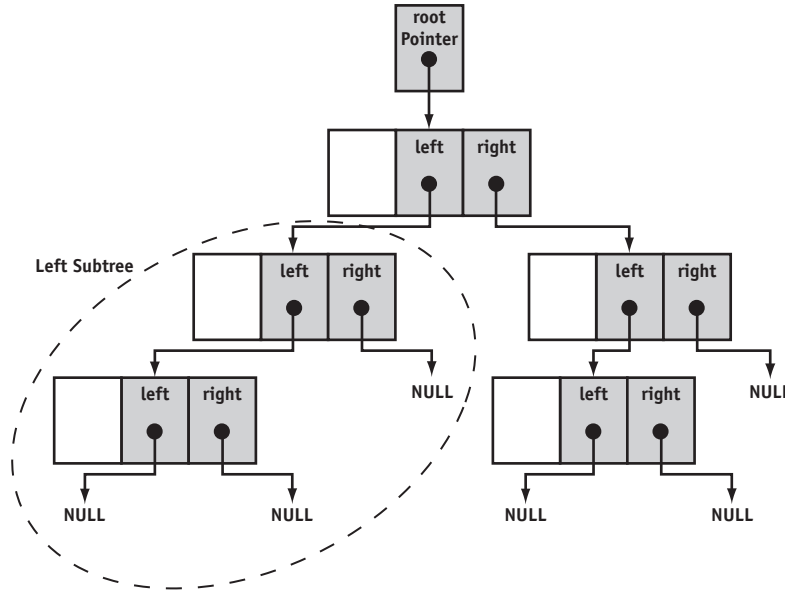


Binary trees are called trees because they resemble an upside-down tree. Any nonempty tree can be partitioned into its root node, its *left subtree*, and its *right subtree*. Intuitively, a subtree is an entire branch of the tree, from one particular node down. Figure 19-2 shows the left subtree of the binary tree shown in Figure 19-1.

Applications of Binary Trees

Searching any linear data structure, such as an array or a standard linked list, is slow when the structure holds a large amount of information. This is because of the sequential nature of linear data structures. Binary trees and their generalizations are excellent data structures for searching large amounts of information. They are commonly used in database applications to organize key values that index database records. When used to facilitate searches,

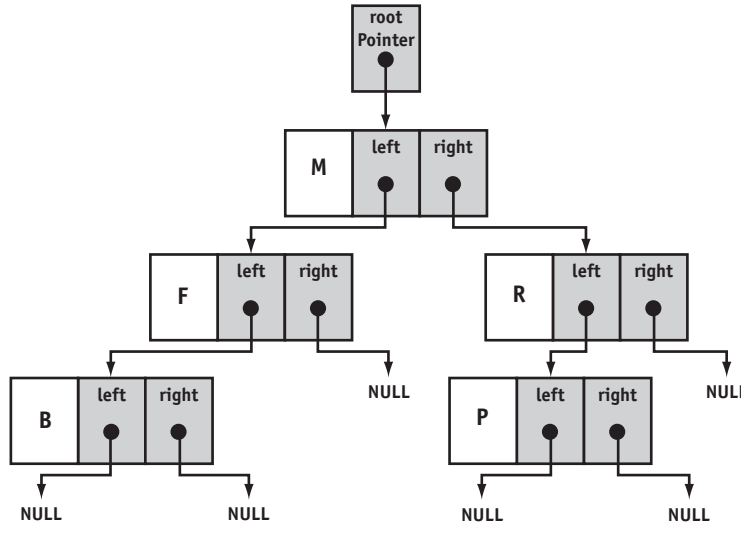
Figure 19-2



a binary tree is called a *binary search tree*. Binary search trees are the primary focus of this chapter.

Information is stored in binary search trees in a way that makes searching for information in the tree simple. For example, look at Figure 19-3.

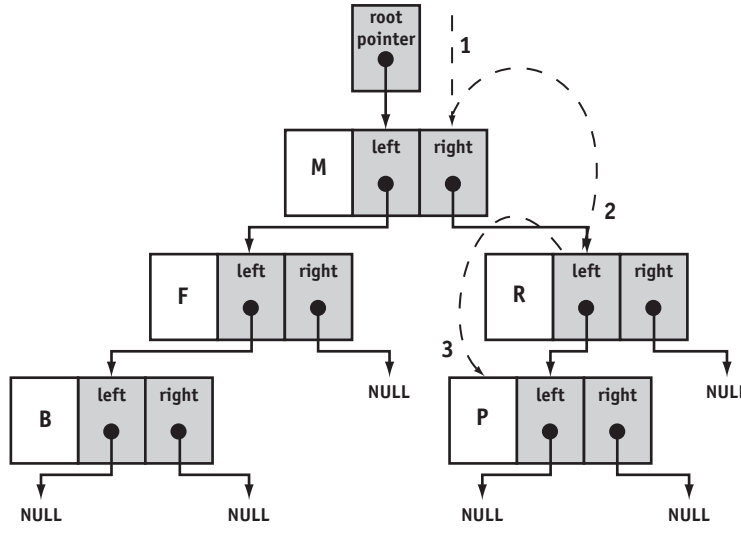
Figure 19-3



The figure depicts a binary search tree where each node stores a letter of the alphabet. Notice that the root node holds the letter M. The left child of the root node holds the letter F, and the right child holds R. Values are stored in a binary search tree so that a node's left child holds data whose value is less than the node's data, and the node's right child holds data whose value is greater than the node's data. This is true for all nodes in the tree that have children.

In fact, in a binary search tree, *all* the nodes to the left of a node hold values less than the node's value. Likewise, all the nodes to the right of a node hold values that are greater than the node's data. When an application is searching a binary search tree, it starts at the root node. If the root node does not hold the search value, the application branches either to the left or right child, depending on whether the search value is less than or greater than the value at the root node. This process continues until the value is found or it is determined that the search value is not in the tree. Figure 19-4 illustrates the search pattern for finding the letter P in the binary tree shown.

Figure 19-4



This manner of searching a binary tree is reminiscent of the binary search technique that is used on sorted arrays. Assuming that the binary tree is balanced (meaning that at each node, the left and right subtrees have approximately the same number of nodes), the search will reduce the size of the tree remaining to be searched by one half at each step. This makes it possible to search trees with very large amounts of information in a relatively small number of steps.

Checkpoint

19.1 Describe the difference between a binary tree and a linked list.

19.2 What is a root node?

19.3 What is a child node?

19.4 What is a leaf node?

19.5 What is a subtree?

19.6 Why are binary trees suitable for algorithms that must search large amounts of information?

19.2 Binary Search Tree Operations

CONCEPT: There are many operations that may be performed on a binary search tree, including creating a binary search tree, inserting, finding, and deleting nodes.

In this section you will learn some basic operations that may be performed on a binary search tree. We will study a simple class that implements a binary tree for storing integer values.

Creating a Binary Search Tree

We will demonstrate the fundamental binary tree operations using a simple ADT: the `IntBinaryTree` class. The basis of our binary tree node is the following `class` declaration:

```
class TreeNode
{
   friend class IntBinaryTree;
   int value;
   TreeNode *left;
   TreeNode *right;
   TreeNode(int value1,
          TreeNode *left1 = NULL,
          TreeNode *right1 = NULL
          )
   {
      value = value1;
      left = left1;
      right = right1;
   }
};
```

Notice that each node of the tree has a `value` member, as well as two pointers to keep track of the left and right children of the node. The class will only be used by methods of `IntBinaryTree`, which is declared a friend of `TreeNode` to allow it access to all of the members of `TreeNode`.

The entire `IntBinaryTree` class follows:

Contents of `IntBinaryTree.h`

```
1 #ifndef INTBINARYTREE_H
2 #define INTBINARYTREE_H
3
4 class IntBinaryTree
5 {
```

```
6  private:
7     // The TreeNode class is used to build the tree.
8     class TreeNode
9     {
10       friend class IntBinaryTree;
11       int value;
12       TreeNode *left;
13       TreeNode *right;
14       TreeNode(int value1, TreeNode *left1 = NULL,
15                            TreeNode *right1 = NULL)
16       {
17          value = value1;
18          left = left1;
19          right = right1;
20       }
21    };
22
23    TreeNode *root;     // Pointer to the root of the tree
24
25    // Various helper member functions.
26    void insert(TreeNode *&, int);
27    void destroySubtree(TreeNode *);
28    void remove(TreeNode *&, int);
29    void makeDeletion(TreeNode *&);
30    void displayInOrder(TreeNode *);
31    void displayPreOrder(TreeNode *);
32    void displayPostOrder(TreeNode *);
33
34 public:
35    // These member functions are the public interface.
36    IntBinaryTree()      // Constructor
37       { root = NULL; }
38    ~IntBinaryTree()     // Destructor
39       { destroySubtree(root); }
40    void insert(int num)
41       { insert(root, num); }
42    bool search(int);
43    void remove(int num)
44       { remove(root, num);}
45    void showInOrder(void)
46       { displayInOrder(root); }
47    void showPreOrder()
48       { displayPreOrder(root); }
49    void showPostOrder()
50       { displayPostOrder(root); }
51 };
52 #endif
```

Besides the `TreeNode` class declaration, the class has a `root` member. This is a pointer to the root node of the binary tree, and plays a role similar to that of the `head` pointer in the linked list class of Chapter 17. In many instances, it is useful to think of the pointer to the node that is the root of a binary tree as the binary tree itself. Thus, we may write

```
TreeNode *tree;
```

or

```
TreeNode *root;
```

and think of both as representing a binary tree because the root provides access to the entire tree. On the other hand, it is also useful to think of an object of the `IntBinaryTree` class as a binary tree, and write

```
IntBinaryTree Tree;
```

To avoid confusion, we will use identifiers with an initial capital letter for a binary tree that is represented by an object of the `IntBinaryTree` class, and use identifiers with initial lowercase letters for a binary tree represented by a pointer to its root node.

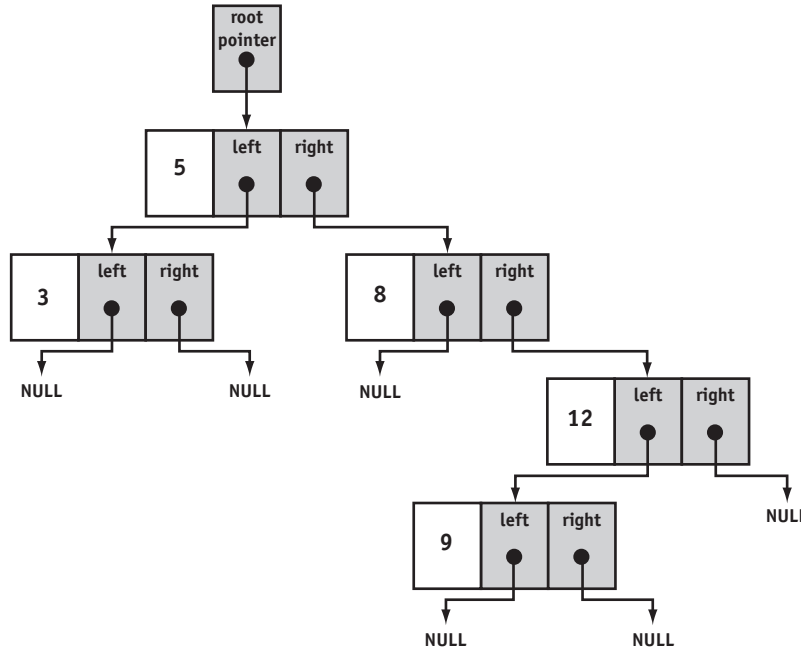
The public member functions of `IntBinaryTree` include a constructor, a destructor, and member functions for inserting a new number into the tree, for searching a tree to determining whether a given number is in the tree, for removing a number from the tree, and for displaying the numbers stored in the tree according to different orders. All of these member functions are discussed in the sections that follow.

Program 19-1 demonstrates the creation of an `IntBinaryTree` object, and the use of the public `insert` member function to build a binary search tree. The implementation code for the member functions are in the `IntBinaryTree.cpp` file; the contents of that file will be discussed later. The tree that results from the execution of Program 19-1 is shown in Figure 19-5.

Program 19-1

```
1  // This program builds a binary tree with 5 nodes.
2  #include <iostream>
3  #include "IntBinaryTree.h"
4  using namespace std;
5
6  int main()
7  {
8     IntBinaryTree tree;
9
10    cout << "Inserting numbers. ";
11    tree.insert(5);
12    tree.insert(8);
13    tree.insert(3);
14    tree.insert(12);
15    tree.insert(9);
16    cout << "Done.\n";
17    return 0;
18 }
```

Figure 19-5



Implementation of the Binary Search Tree Operations

Many binary tree operations have very natural recursive implementations. This is because a binary tree is an inherently recursive data structure: every nonempty binary tree consists of a root node together with the left and right subtrees, which are of course, binary trees. Many binary tree operations can be implemented by performing some processing at the root node and then recursively performing the operation on the left and right subtrees. For example, if the root node is represented by a pointer

```
TreeNode *tree;
```

Then the value in the root node will be `tree->value`, and the left and right subtrees will be given by `tree->left` and `tree->right`. A recursive operation might first process `tree->value`, and then the recursively operate on `tree->left` and `tree->right`.



VideoNote
Inserting an Element into a Binary Tree

Inserting an Element

The work of inserting a number into a binary search tree is performed by the private member function

```
insert(TreeNode *&tree, int num)
```

which is passed a pointer `tree` to the root node of a binary search tree and a number `num` to be inserted into the tree. It uses a recursive strategy: if the binary tree is empty (this is the base case for the recursion), it creates a new `TreeNode` object whose value member is the given number, and makes it the root of the tree:

```
if (!tree)
{
   tree = new TreeNode(num);
   return;
}
```

If, however, the binary search tree is not empty, the `insert` function compares `num` to the `tree->value`, the value in the root node. Depending on the outcome of this comparison, the new value is recursively inserted into the left or right subtree:

```
if (num < tree->value)
   insert(tree->left, num);
else
   insert(tree->right, num);
```

The entire function is given here:

```
void IntBinaryTree::insert(TreeNode * &tree, int num)
{
   // If the tree is empty, make a new node and make it
   // the root of the tree.
   if (!tree)
      {
        tree = new TreeNode(num);
        return;
      }

   // If num is already in tree: return.
   if (tree->value == num)
     return;

   // The tree is not empty: insert the new node into the
   // left or right subtree.
   if (num < tree->value)
      insert(tree->left, num);
   else
      insert(tree->right, num);
}
```

Note that the function is passed a reference to a pointer because the pointer passed may need to be modified by the function. This is also the reason the `remove` and `makeDeletion` functions are passed their parameters by reference.



NOTE: The shape of the tree shown in Figure 19-5 is determined by the order in which the values are inserted. The root node holds the value 5 because that was the first value inserted. By stepping through the function, you can see how the other nodes came to appear in their depicted positions.

Traversing the Tree

There are three common methods for traversing a nonempty binary tree and processing the value of each node: *inorder*, *preorder*, and *postorder*. Each of these methods is best implemented as a recursive function. The algorithms are described as follows.

- *Inorder traversal*
  1. The node's left subtree is traversed.
  2. The node's data is processed.
  3. The node's right subtree is traversed.
- *Preorder traversal*
  1. The node's data is processed.
  2. The node's left subtree is traversed.
  3. The node's right subtree is traversed.
- *Postorder traversal*
  1. The node's left subtree is traversed.
  2. The node's right subtree is traversed.
  3. The node's data is processed.

The `IntBinaryTree` class can display all the values in the tree using all three of these algorithms. The algorithms are initiated by the following inline public member functions:

```
void showInOrder(void)
   { displayInOrder(root); }
void showPreOrder()
   { displayPreOrder(root); }
void showPostOrder()
   { displayPostOrder(root); }
```

Each of the public member functions calls a recursive private member function and passes the root pointer as an argument. The recursive functions are very simple and straightforward:

```
void IntBinaryTree::displayInOrder(TreeNode *tree)
{
   if (tree)
   {
      displayInOrder(tree->left);
      cout << tree->value << "  ";
      displayInOrder(tree->right);
   }
}

void IntBinaryTree::displayPreOrder(TreeNode *tree)
{
   if (tree)
   {
      cout << tree->value << "  ";
      displayPreOrder(tree->left);
      displayPreOrder(tree->right);
   }
}

void IntBinaryTree::displayPostOrder(TreeNode *tree)
{
   if (tree)
   {
      displayPostOrder(tree->left);
      displayPostOrder(tree->right);
      cout << tree->value << "  ";
   }
}
```

Program 19-2, which is a modification of Program 19-1, demonstrates each of these traversal methods.

Program 19-2

```
1  // This program builds a binary tree with 5 nodes.
2  // The nodes are displayed with inorder, preorder,
3  // and postorder algorithms.
4  #include <iostream>
5  #include "IntBinaryTree.h"
6  using namespace std;
7
8  int main()
9  {
10    IntBinaryTree tree;
11    cout << "Inserting the numbers 5 8 3 12 9.\n\n";
12    tree.insert(5);
13    tree.insert(8);
14    tree.insert(3);
15    tree.insert(12);
16    tree.insert(9);
17
18    cout << "Inorder traversal:  ";
19    tree.showInOrder();
20
21    cout << "\n\nPreorder traversal:  ";
22    tree.showPreOrder();
23
24    cout << "\n\nPostorder traversal:  ";
25    tree.showPostOrder();
26    return 0;
27 }
```

Program Output

```
Inserting the numbers 5 8 3 12 9.

Inorder traversal:  3  5  8  9  12

Preorder traversal:  5  3  8  12  9

Postorder traversal:  3  9  12  8  5
```

Searching the Binary Search Tree

The `IntBinarySearchTree` class has a public member function `search`, which returns `true` if a given value is found in the tree and returns `false` otherwise. The function simply starts out searching the entire tree. The function compares `num`, the value being searched for, to the value in the root of the tree it is currently searching. If the value matches, the function returns `true`. If the value does not match, the function replaces the tree with either its left subtree or its right subtree and continues the search. The search will terminate when the function finds the value or when the tree being searched becomes empty.

```
bool IntBinaryTree::search(int num)
{
   TreeNode *tree = root;

   while (tree)
   {
      if (tree->value == num)
         return true;
      else if (num < tree->value)
         tree = tree->left;
      else
         tree = tree->right;
   }
   return false;
}
```

Program 19-3 demonstrates this function.

Program 19-3

```
// This program builds a binary tree with 5 nodes.
// The search function determines if the
// value 3 is in the tree.
#include <iostream>
#include "IntBinarytree.h"
using namespace std;

int main()
{
   IntBinaryTree tree;
   cout << "Inserting the numbers 5 8 3 12 9.\n\n";
   tree.insert(5);
   tree.insert(8);
   tree.insert(3);
   tree.insert(12);
   tree.insert(9);

   if (tree.search(3))
      cout << "3 is found in the tree.\n";
   else
      cout << "3 was not found in the tree.\n";
   return 0;
}
```

Program Output

```
Inserting the numbers 5 8 3 12 9.

3 is found in the tree.
```



VideoNote
Removing an Element from a Binary Tree

Removing an Element

To remove an element, we first locate the node containing the element, and then delete the node. The procedure for deleting a node X depends on the number of its children. If X has no children, we first find its parent, set the parent's child pointer that links to X to NULL,

and then free the memory allocated to X. If X is the root of the tree, the procedure we have just described will not work. In that case, we simply delete X and set the pointer to the root of the tree to NULL.

A procedure for deleting a nonleaf node must ensure that the subtrees that the node links to remain as parts of the tree. The procedure varies according to whether the node being deleted has one or two children. Figure 19-6 shows a tree in which we are about to delete a node with one subtree.

Figure 19-6

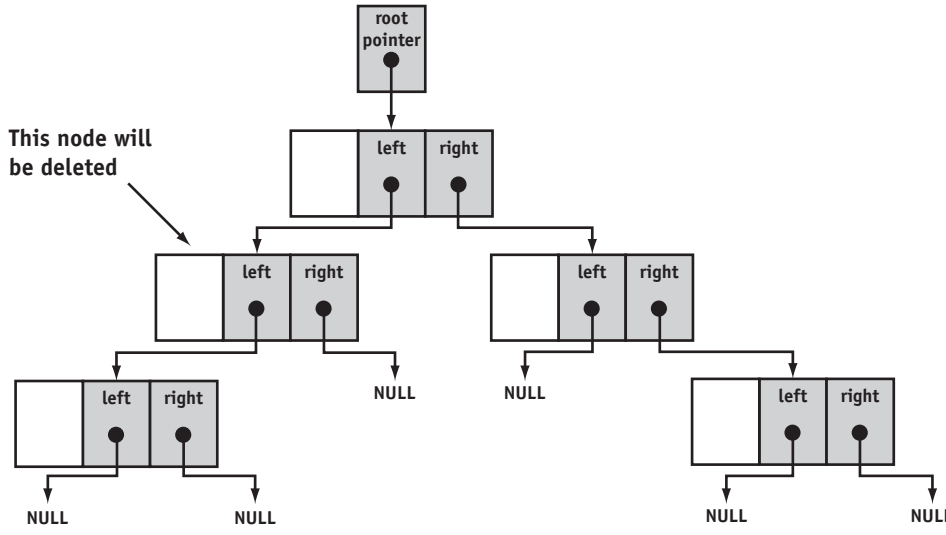
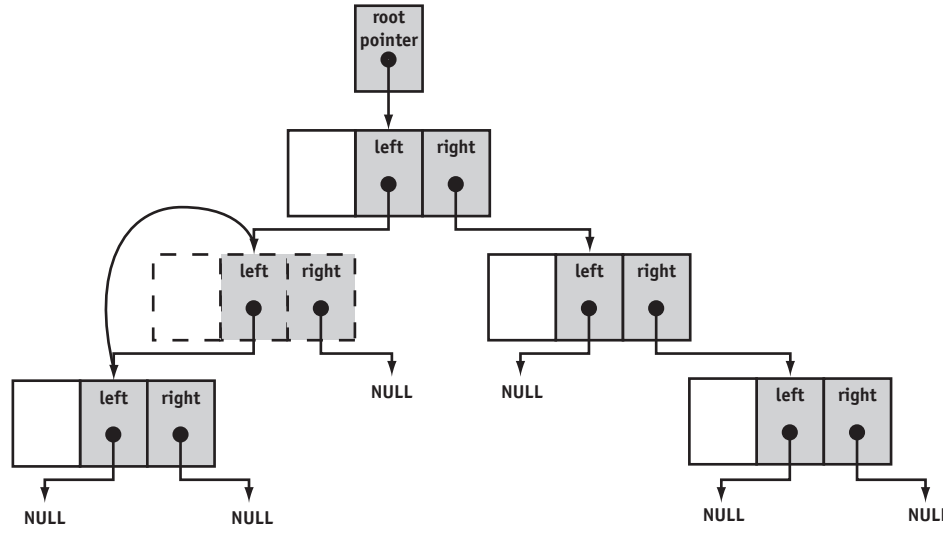


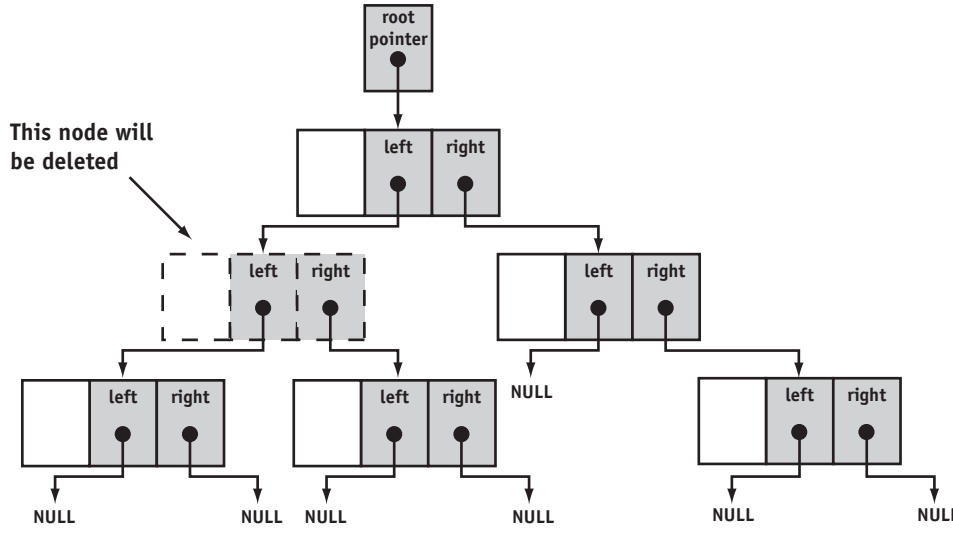
Figure 19-7 shows how we will link the node's subtree with its parent.

Figure 19-7



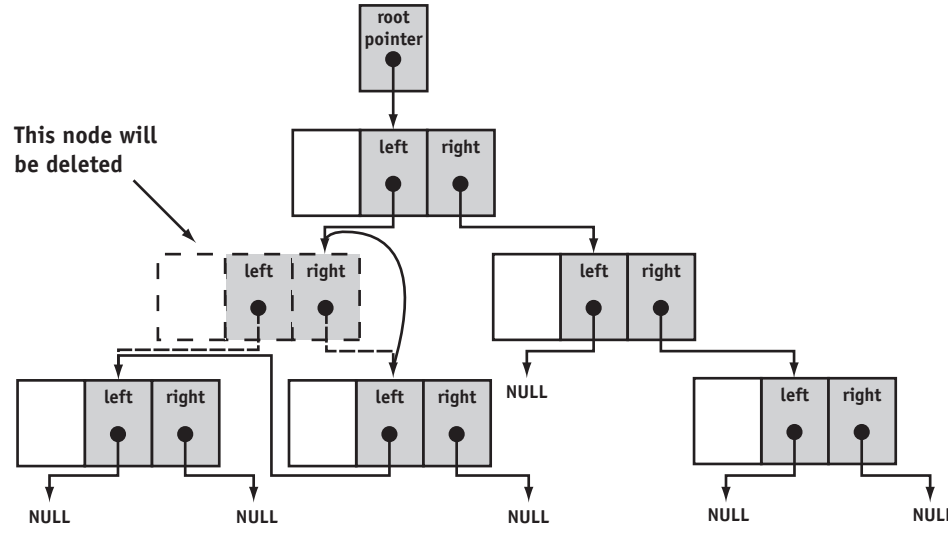
The problem is not as easily solved, however, when the node we are about to delete has two subtrees. For example, look at Figure 19-8.

Figure 19-8



Obviously, we cannot attach both of the node's subtrees to its parent, so there must be an alternative solution. One way of addressing this problem is to attach the node's right subtree to the parent, then find a position in the right subtree to attach the left subtree. The result is shown in Figure 19-9. Note that in attaching the left subtree to the right subtree, we must take care to preserve the binary tree's search property.

Figure 19-9



The deletion of a value from an `IntBinaryTree` object is accomplished by calling the public member function `remove`, which in turn calls the private member function of the same name. This latter function is passed (the root of) a binary search tree `tree`, and a value `num` to be removed from the tree:

```
remove(TreeNode *&tree, int num)
```

The function uses a recursive strategy. If the tree is empty, it returns immediately. Otherwise, if `num` is less than the value stored in the root node, the function recursively removes `num` from the left subtree; but if `num` is greater, the function recursively removes `num` from the right subtree. The case where `num` is found in the root node is handed off to a function

```
makeDeletion(TreeNode *&tree)
```

Here is the code for `remove`:

```
void IntBinaryTree::remove(TreeNode *&tree, int num)
{
   if (tree == NULL) return;
   if (num < tree->value)
      remove(tree->left, num);
   else if (num > tree->value)
      remove(tree->right,num);
   else
      // We have found the node to delete.
      makeDeletion(tree);
}
```

The `makeDeletion` function is designed to remove the root node of the binary search tree passed to it as an argument, leaving a binary search tree consisting of the remaining nodes. Let us take a look at the logic behind `makeDeletion`. There are a number of cases to consider:

1. The root of the tree passed to `makeDeletion` has no children. In this case, we delete the root node and replace the tree with NULL.
2. The root of the tree has only one child. In this case we delete the root node and replace the tree with the child of the deleted root:

   ```
   TreeNode *nodeToDelete = tree;
   if (tree->right == NULL)
      tree = tree->left;
   else if (tree->left == NULL)
      tree = tree->right;
   ```

   Note that this code works for the first case as well.
3. The tree passed to `makeDelete` has two children. The deletion of the root node would leave two subtrees, and we need to do something with both of them. The strategy we adopt is to combine the two subtrees into one binary search tree, and then replace the original tree with the tree built from the combined subtrees. As shown in Figure 19-9, we can do this by attaching the left subtree of the original tree as the left subtree of the least node in the right subtree of the original tree. Here is the code for the entire function.

```
void IntBinaryTree::makeDeletion(TreeNode *&tree)
{
   // Used to hold node that will be deleted.
   TreeNode *nodeToDelete = tree;

   // Used to locate the  point where the
   // left subtree is attached.
   TreeNode *attachPoint;

   if (tree->right == NULL)
   {
      // Replace tree with its left subtree.
      tree = tree->left;
   }
   else if (tree->left == NULL)
   {
      // Replace tree with its right subtree.
      tree = tree->right;
   }
   else
      //The node has two children
      {
         // Move to right subtree.
         attachPoint = tree->right;

         // Locate the smallest node in the right subtree
         // by moving as far to the left as possible.
         while (attachPoint->left != NULL)
         attachPoint = attachPoint->left;

         // Attach the left subtree of the original tree
         // as the left subtree of the smallest node
         // in the right subtree.
         attachPoint->left = tree->left;

         // Replace the original tree with its right subtree.
         tree = tree->right;
      }

      // Delete root of original tree
      delete nodeToDelete;
}
```

Program 19-4 demonstrates these functions.

Program 19-4

```
1 // This program builds a binary tree with 5 nodes.
2 // The deleteNode function is used to remove 2 of them.
3 #include <iostream>
4 #include "IntBinaryTree.h"
5 using namespace std;
6
```

(program continues)

Program 19-4 *(continued)*

```
7  int main()
8  {
9     IntBinaryTree tree;
10
11    cout << "Inserting the numbers 5 8 3 12 9.";
12    tree.insert(5);
13    tree.insert(8);
14    tree.insert(3);
15    tree.insert(12);
16    tree.insert(9);
17
18    cout << "\nHere are the values in the tree:\n";
19    tree.showInOrder();
20
21    cout << "\nDeleting 8...\n";
22    tree.remove(8);
23
24    cout << "Deleting 12...\n";
25    tree.remove(12);
26
27    cout << "Now, here are the nodes:\n";
28    tree.showInOrder();
29    return 0;
30 }
```

Program Output

```
Inserting the numbers 5 8 3 12 9.
Here are the values in the tree:
3  5  8  9  12
Deleting 8...
Deleting 12...
Now, here are the nodes:
3  5  9
```

For your reference, the entire contents of `IntBinaryTree` file are shown here:

Contents of `IntBinaryTree.cpp`

```
1  #include <iostream>
2  #include "IntBinaryTree.h"
3  using namespace std;
4
5  //************************************************
6  // This version of insert inserts a number into    *
7  // a given subtree of the main binary search tree. *
8  //************************************************
9  void IntBinaryTree::insert(TreeNode * &tree, int num)
10 {
11    // If the tree is empty, make a new node and make it
12    // the root of the tree.
13    if (!tree)
14       {
15         tree = new TreeNode(num);
```

```
16         return;
17      }
18
19   // If num is already in tree: return.
20   if (tree->value == num)
21      return;
22
23   // The tree is not empty: insert the new node into the
24   // left or right subtree.
25   if (num < tree->value)
26      insert(tree->left, num);
27   else
28      insert(tree->right, num);
29 }
30
31 //************************************************
32 // destroySubTree is called by the destructor. It   *
33 // deletes all nodes in the tree.                   *
34 //************************************************
35 void IntBinaryTree::destroySubtree(TreeNode *tree)
36 {
37      if (!tree) return;
38      destroySubtree(tree->left);
39      destroySubtree(tree->right);
40      // Delete the node at the root.
41      delete tree;
42 }
43
44 //************************************************
45 // searchNode determines if a value is present in   *
46 // the tree. If so, the function returns true.      *
47 // Otherwise, it returns false.                     *
48 //************************************************
49 bool IntBinaryTree::search(int num)
50 {
51   TreeNode *tree = root;
52
53   while (tree)
54   {
55      if (tree->value == num)
56         return true;
57      else if (num < tree->value)
58         tree = tree->left;
59      else
60         tree = tree->right;
61   }
62   return false;
63 }
64
65 //****************************************
66 // remove deletes the node in the given tree *
67 // that has a value member the same as num.  *
68 //****************************************
```

```
69 void IntBinaryTree::remove(TreeNode *&tree, int num)
70 {
71    if (tree == NULL) return;
72    if (num < tree->value)
73       remove(tree->left, num);
74    else if (num > tree->value)
75       remove(tree->right,num);
76    else
77       // We have found the node to delete.
78       makeDeletion(tree);
79 }
80
81 //*********************************************************
82 // makeDeletion takes a reference to a tree whose root      *
83 // is to be deleted. If the tree has a single child,        *
84 // the tree is replaced by the single child after the       *
85 // removal of its root node. If the tree has two children   *
86 // the left subtree of the deleted node is attached at      *
87 // an appropriate point in the right subtree, and then      *
88 // the right subtree replaces the original tree.            *
89 //*********************************************************
90 void IntBinaryTree::makeDeletion(TreeNode *&tree)
91 {
92    // Used to hold node that will be deleted.
93    TreeNode *nodeToDelete = tree;
94
95    // Used to locate the  point where the
96    // left subtree is attached.
97    TreeNode *attachPoint;
98
99    if (tree->right == NULL)
100   {
101      // Replace tree with its left subtree.
102      tree = tree->left;
103   }
104   else if (tree->left == NULL)
105   {
106      // Replace tree with its right subtree.
107      tree = tree->right;
108   }
109   else
110      //The node has two children
111      {
112        // Move to right subtree.
113        attachPoint = tree->right;
114
115        // Locate the smallest node in the right subtree
116        // by moving as far to the left as possible.
117        while (attachPoint->left != NULL)
118           attachPoint = attachPoint->left;
119
120        // Attach the left subtree of the original tree
121        // as the left subtree of the smallest node
122        // in the right subtree.
123        attachPoint->left = tree->left;
124
```

```
125         // Replace the original tree with its right subtree.
126         tree = tree->right;
127       }
128
129    // Delete root of original tree
130    delete nodeToDelete;
131 }
132
133 //********************************************************
134 // This function displays the values  stored in a tree    *
135 // in inorder.                                            *
136 //********************************************************
137 void IntBinaryTree::displayInOrder(TreeNode *tree)
138 {
139    if (tree)
140    {
141       displayInOrder(tree->left);
142       cout << tree->value << "  ";
143       displayInOrder(tree->right);
144    }
145 }
146
147 //********************************************************
148 // This function displays the values stored in a tree     *
149 // in inorder.                                            *
150 //********************************************************
151 void IntBinaryTree::displayPreOrder(TreeNode *tree)
152 {
153    if (tree)
154    {
155       cout << tree->value << "  ";
156       displayPreOrder(tree->left);
157       displayPreOrder(tree->right);
158    }
159 }
160
161 //********************************************************
162 // This function displays the values  stored  in a tree   *
163 // in postorder.                                          *
164 //********************************************************
165 void IntBinaryTree::displayPostOrder(TreeNode *tree)
166 {
167    if (tree)
168    {
169       displayPostOrder(tree->left);
170       displayPostOrder(tree->right);
171       cout << tree->value << "  ";
172    }
173 }
```



Checkpoint

19.7 Describe the sequence of events in an inorder traversal.

19.8 Describe the sequence of events in a preorder traversal.

19.9 Describe the sequence of events in a postorder traversal.

19.10 Describe the steps taken in deleting a leaf node.

19.11 Describe the steps taken in deleting a node with one child.

19.12 Describe the steps taken in deleting a node with two children.

19.3 Template Considerations for Binary Search Trees

CONCEPT: Binary search trees may be implemented as templates, but any data types used with them must support the <, >, and == operators.

The actual implementation of a binary tree template has been left as a Programming Challenge for students who have covered Chapters 16 and 19. When designing your template, remember that any data types stored in the binary tree must support the <, >, and == operators. If you use the tree to store class objects, these operators must be overridden.

19.4 Tying It All Together: *Genealogy Trees*

Say we want to write a program that will trace peoples' ancestries and build genealogy trees. To keep track of each person's biological parents, we might use a class such as the following:

```
class Person
{
   string name;
   Person *father;
   Person *mother;
};
```

This simple class is very similar to the "node" classes we have been using to build binary trees. In maintaining genealogies, however, we are interested in recording not only a person's ancestors, but their descendants as well. We also want to keep track of gender information to enable people using the program to distinguish between maternal and paternal relatives. The `Person` class, as shown above, is not adequate for these needs. We therefore modify it as shown here:

```
enum Gender {male, female};
class Person
{
   string name;
   Gender gender;
   vector<Person *> parents;
   vector<Person *> children;
};
```

We now have a "node" that can have any number of children and any number of parents. Because each person can have at most two parents, the size of the `parents` vector will never exceed two.

We can make the `Person` class more useful by adding a constructor and several member functions. The method

```
Person *addChild(string name, Gender g);
```

creates a `Person` object with the specified name and gender, adds a pointer `p` to the created object to the `children` of "this" `Person` object, and returns `p` to the caller. Another method,

```
Person *addChild(Person *p);
```

adds a pointer to an already created `Person` object to the `children` vector of `"this"` `Person` object. The following code shows the use of these member functions to record the fact that a father `f` and a mother `m` have a child named `"Charlie"`:

```
Person f("Frank", male);
Person m("Mary", female);
Person *pChild = m.addChild("Charlie", male);
f.addChild(pChild);
```

There is also a method

```
void addParent(Person *p);
```

that is used to record the fact that one person is the parent of another. The class also has an overloaded stream operator that outputs the data in a `Person` object using an XML-like format. Finally, the class has several methods that can be used to access information about various members of the class objects. These additional functions can be seen in Lines 29–34 of the program listing.

Program 19-5

```
1  // This program uses a generalization of binary trees to build
2  // genealogy trees.
3  #include <vector>
4  #include <string>
5  #include <iostream>
6  using namespace std;
7  enum Gender{male, female};
8
9  // Person class represents a person participating in a genealogy.
10 class Person
11 {
12    string name;
13    Gender gender;
14    vector<Person *> parents;
15    vector<Person *> children;
16    void addParent(Person *p){ parents.push_back(p); }
17 public:
18     Person (string name, Gender g)
```

(program continues)

Program 19-5 *(continued)*

```
19     {
20         this->name = name;
21         gender = g;
22     }
23    Person *addChild(string name, Gender g);
24    Person *addChild(Person *p);
25
26    friend ostream &operator << (ostream &out, Person p);
27
28    // Member functions for getting various Person info
29    string getName(){ return name; };
30    Gender getGender(){ return gender; };
31    int getNumChildren(){ return children.size(); }
32    int getNumParents(){ return parents.size(); }
33    Person *getChild(int k);
34    Person *getParent(int k);
35 };
36
37 //*************************************************************
38 // Create a child with specified name and gender, and         *
39 // set one of the parents to be this person.                  *
40 // Add the new child to the list of children for this person *
41 //*************************************************************
42 Person *Person::addChild(string name, Gender g)
43 {
44     Person *child = new Person(name, g);
45     child->addParent(this);     // I am a parent of this child
46     children.push_back(child);  // This is one of my children
47     return child;
48 }
49
50 //*************************************************************
51 // Add a child to the list of children for this person       *
52 //*************************************************************
53 Person *Person::addChild(Person* child)
54 {
55    child->addParent(this);    // I am a parent of this child
56    children.push_back(child); // This is one of my children
57    return child;
58 }
59
60 //*********************************************************
61 // Return a pointer to the specified parent               *
62 //*********************************************************
63 Person *Person::getParent(int k)
64 {
65    if (k < 0 || k >= parents.size())
66    {
67       cout << "Error indexing parents vector." << endl;
68       exit(1);
69    }
70    return parents[k];
71 }
```

(program continues)

Program 19-5 *(continued)*

```
72
73 //*********************************************************
74 // Return a pointer to a specified child                  *
75 //*********************************************************
76 Person *Person::getChild(int k)
77 {
78    if (k < 0 || k >= children.size())
79    {
80       cout << "Error indexing children's vector." << endl;
81       exit(1);
82    }
83     return children[k];
84 }
85
86 //*****************************************************
87 // Overloaded stream output operator                  *
88 //*****************************************************
89 ostream & operator<<(ostream & out, Person p)
90 {
91     out << "<person name = " << p.name << ">" << '\n';
92     if (p.parents.size() > 0)
93         out << "   <parents>" << ' ';
94     for (int k = 0; k < p.parents.size(); k++)
95     {
96         out << " " << p.parents[k]->name << ' ';
97     }
98     if (p.parents.size() > 0)
99        out << " </parents>" << "\n";
100    if (p.children.size() > 0)
101       out << "   <children>" << ' ';
102    for (int k = 0; k < p.children.size(); k++)
103    {
104        out << " " << p.children[k]->name << ' ';
105    }
106    if (p.children.size() > 0)
107         out << " </children>" << "\n";
108    out << "</person>" << "\n";
109    return out;
110 }
111
112
113 int main(int argc, char** argv)
114 {
115     // Here are the people
116     Person adam("Adam", male);
117     Person eve("Eve", female);
118     Person joan("Joan", female);
119
```

(program continues)

Program 19-5 *(continued)*

```
120     // Adam and Eve are parents of Abel
121     Person *pAbel = eve.addChild(new Person("Abel", male));
122     adam.addChild(pAbel);
123
124     // Abel and Joan are parents of Missy
125     Person *pMissy = joan.addChild("Missy", female);
126     pAbel->addChild(pMissy);
127
128     // Output all the people
129     cout << "Here are all the people:\n\n";
130     cout << adam << eve";
131     cout << *pAbel << joan;
132     cout << *pMissy << "\n";
133
134     // Print parents of Missy
135     cout << "Missy's parents are: " << endl;
136     for (unsigned int k = 0; k < pMissy->getNumParents(); k++)
137     {
138         Person * p = pMissy->getParent(k);
139         switch(p->getGender())
140         {
141            case female : cout << "\tMother: "; break;
142            case male: cout << "\tFather: "; break;
143         }
144         cout << p->getName() << endl;
145     }
146     return 0;
147 }
```

Program Output

```
Here are all the people:

<person name = Adam>
   <children>  Abel  </children>
</person>
<person name = Eve>
   <children>  Abel  </children>
</person>
<person name = Abel>
   <parents>  Eve  Adam  </parents>
   <children>  Missy  </children>
</person>
<person name = Joan>
   <children>  Missy  </children>
</person>
<person name = Missy>
   <parents>  Joan  Abel  </parents>
</person>

Missy's parents are:
        Mother: Joan
        Father: Abel
```

Review Questions and Exercises

Fill-in-the-Blank and Short Answer

1. The first node in a binary tree is called the _____.
2. A binary tree node's left and right pointers point to the node's _____.
3. A node with no children is called a(n) _____.
4. A(n)_____ is an entire branch of the tree, from one particular node down.
5. The three common types of traversal with a binary tree are _____, _____, and _____.
6. In what ways is a binary tree similar to a linked list?
7. A *ternary* tree is like a binary tree, except each node in a ternary tree may have three children: a left child, a middle child, and a right child. Write an analogue of the `TreeNode` declaration that can be used to represent the nodes of a ternary tree.
8. Imagine a tree in which each node can have up to a hundred children. Write an analog of the `TreeNode` declaration that can be used to represent the nodes of such a tree. A declaration such as

```
TreeNode
{
   int value;
   TreeNode *child1;
   TreeNode *child2;
   TreeNode *child3;
   .
   .
   .
};
```

that simply lists all the pointers to the hundred children is not acceptable.

Algorithm Workbench

9. Propose a definition of a *preorder traversal* for ternary trees, and give pseudocode for accomplishing such a traversal.
10. Propose a definition of a *postorder traversal* for ternary trees, and give pseudocode for accomplishing such a traversal.
11. What problems do you encounter when you try to define the concept of an *inorder traversal* for ternary trees?
12. Assume that data is stored in a binary tree, but that unlike in the case of binary search tree, no attempt is made to maintain any sort of order in the data stored. Give an algorithm for a function `search` that searches a binary tree for a particular value `num` and returns `true` or `false` according to whether the value `num` is found in the tree.
13. Give an algorithm for a function

```
int largest(TreeNode *tree)
```

that takes a pointer to a root of a binary search tree as parameter and returns the largest value stored in the tree.

14. Give an algorithm for a function

```
int smallest(TreeNode *tree)
```

that takes a pointer to a root of a binary search tree as parameter and returns the smallest value stored in the tree.

15. Give an algorithm for a function

```
void increment (TreeNode *tree)
```

that increments the value in every node of a binary tree by one.

16. Suppose the following values are inserted into a binary search tree, in the order given:

```
12, 7, 9, 10, 22, 24, 30, 18, 3, 14, 20
```

Draw a diagram of the resulting binary tree.

17. How would the values in the tree you sketched for queston 16 be displayed in an inorder traversal?
18. How would the values in the tree you sketched for queston 16 be displayed in a preorder traversal?
19. How would the values in the tree you sketched for queston 16 be displayed in a postorder traversal?

Soft Skills

20. All three binary tree traversal methods studied in this chapter traverse the left subtree before the right subtree. This is an artifact of western culture, where people are accustomed to reading material printed on a page from left to right. In a world of increasing globalization, products and services that will be offered in foreign markets must be designed so that they can be easily altered to target different markets. Discuss with your classmates some of the ways these internationalization considerations are affecting the design of computer software and hardware today. Discuss this with a friend who has had a course in International Business, or take such a course yourself, to become better aware of some of the problems businesses face when they enter international markets.

Programming Challenges



1. Simple Binary Search Tree Class

Write a class for implementing a simple binary search tree capable of storing numbers. The class should have member functions

```
void insert(double x)
bool search(double x)
void inorder(vector <double> & v )
```

The insert function should not use recursion directly, or indirectly by calling a recursive function. The search function should work by calling a private recursive member function

```
bool search(double x, BtreeNode *t)
```

The inorder function is passed an initially empty vector v: it fills v with the inorder list of numbers stored in the binary search tree. Demonstrate the operation of the class using a suitable driver program.



VideoNote
Solving the Tree Size Problem

2. Tree Size

Modify the binary search tree created in the previous programming challenge to add a member function

```
int size()
```

that returns the number of items (nodes) stored in the tree. Demonstrate the correctness of the new member function with a suitable driver program.



3. Leaf Counter

Modify the binary search tree you created in the preceding programming challenges to add a member function

```
int leafCount()
```

that counts and returns the number of leaf nodes in the tree. Demonstrate that the function works correctly in a suitable driver program.

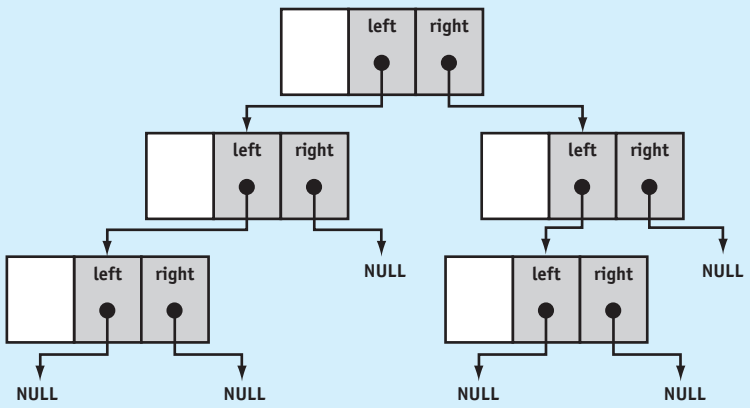
4. Tree Height

Modify the binary search tree created in the preceding programming challenges by adding a member function that computes and returns the height of the tree.

```
int height()
```

The height of the tree is the number of levels it contains. For example, the tree shown in Figure 19-10 has three levels. Demonstrate the function with a suitable driver program.

Figure 19-10



5. Tree Width

Modify the binary search tree created in the preceding programming challenges by adding a member function that computes the width of the tree.

```
int width()
```

The width of a tree is the largest number of nodes at the same level. Demonstrate correctness in a suitable driver program.

6. Tree Copy Constructor

Design and implement a copy constructor for the binary search tree created in the preceding programming challenges. Use a driver program to demonstrate correctness.

7. Tree Assignment Operator

Design and implement an overloaded assignment operator for the binary search tree created in the preceding programming challenges.

8. Employee Tree

Design an `EmployeeInfo` class that holds the following employee information:

Employee ID Number: an integer
Employee Name: a string

Implement a binary tree whose nodes hold an instance of the `EmployeeInfo` class. The nodes should be sorted on the Employee ID number.

Test the binary tree by inserting nodes with the following information.

Employee ID Number	Name
1021	John Williams
1057	Bill Witherspoon
2487	Jennifer Twain
3769	Sophia Lancaster
1017	Debbie Reece
1275	George McMullen
1899	Ashley Smith
4218	Josh Plemmons

Your program should allow the user to enter an ID number, then search the tree for the number. If the number is found, it should display the employee's name. If the node is not found, it should display a message indicating so.

9. Cousins

Building on Program 19-5, write a function that takes a pointer to a `Person` object and produces a list of that person's cousins.

This page intentionally left blank

APPENDIX A

The ASCII Character Set

Nonprintable ASCII Characters			
Dec	Hex	Oct	Name of Character
0	0	0	NULL
1	1	1	SOTT
2	2	2	STX
3	3	3	ETY
4	4	4	EOT
5	5	5	ENQ
6	6	6	ACK
7	7	7	BELL
8	8	10	BKSPC
9	9	11	HZTAB
10	a	12	NEWLN
11	b	13	VTAB
12	c	14	FF
13	d	15	CR
14	e	16	SO
15	f	17	SI
16	10	20	DLE
17	11	21	DC1
18	12	22	DC2
19	13	23	DC3
20	14	24	DC4
21	15	25	NAK
22	16	26	SYN
23	17	27	ETB
24	18	30	CAN
25	19	31	EM
26	1a	32	SUB
27	1b	33	ESC
28	1c	34	FS
29	1d	35	GS
30	1e	36	RS
31	1f	37	US
127	7f	177	DEL

Printable ASCII Characters			
Dec	Hex	Oct	Character
32	20	40	(Space)
33	21	41	!
34	22	42	“
35	23	43	#
36	24	44	$
37	25	45	%
38	26	46	&
39	27	47	‘
40	28	50	(
41	29	51	)
42	2a	52	*
43	2b	53	+
44	2c	54	’
45	2d	55	-
46	2e	56	.
47	2f	57	/
48	30	60	0
49	31	61	1
50	32	62	2
51	33	63	3
52	34	64	4
53	35	65	5
54	36	66	6
55	37	67	7
56	38	70	8
57	39	71	9
58	3a	72	:
59	3b	73	;
60	3c	74	<
61	3d	75	=
62	3e	76	>
63	3f	77	?
64	40	100	@

Printable ASCII Characters

Dec	Hex	Oct	Character
65	41	101	A
66	42	102	B
67	43	103	C
68	44	104	D
69	45	105	E
70	46	106	F
71	47	107	G
72	48	110	H
73	49	111	I
74	4a	112	J
75	4b	113	K
76	4c	114	L
77	4d	115	M
78	4e	116	N
79	4f	117	O
80	50	120	P
81	51	121	Q
82	52	122	R
83	53	123	S
84	54	124	T
85	55	125	U
86	56	126	V
87	57	127	W
88	58	130	X
89	59	131	Y
90	5a	132	Z
91	5b	133	[
92	5c	134	\
93	5d	135	]
94	5e	136	^
95	5f	137	_
96	60	140	`
97	61	141	a
98	62	142	b
99	63	143	c
100	64	144	d
101	65	145	e
102	66	146	f
103	67	147	g
104	68	150	h
105	69	151	i
106	6a	152	j
107	6b	153	k
108	6c	154	l
109	6d	155	m
110	6e	156	n
111	6f	157	o
112	70	160	p
113	71	161	q
114	72	162	r
115	73	163	s
116	74	164	t
117	75	165	u
118	76	166	v
119	77	167	w
120	78	170	x

Printable ASCII Characters

Dec	Hex	Oct	Character
121	79	171	y
122	7a	172	z
123	7b	173	{
124	7c	174	|
125	7d	175	}
126	7e	176	~

Extended ASCII Characters

Dec	Hex	Oct	Character
128	80	200	Ç
129	81	201	ü
130	82	202	é
131	83	203	â
132	84	204	ä
133	85	205	à
134	86	206	å
135	87	207	ç
136	88	210	ê
137	89	211	ë
138	8a	212	è
139	8b	213	ï
140	8c	214	î
141	8d	215	ì
142	8e	216	Ä
143	8f	217	Å
144	90	220	É
145	91	221	æ
146	92	222	Æ
147	93	223	ô
148	94	224	ö
149	95	225	ò
150	96	226	û
151	97	227	ù
152	98	230	ÿ
153	99	231	Ö
154	9a	232	Ü
155	9b	233	¢
156	9c	234	£
157	9d	235	ù
158	9e	236	û
159	9f	237	ƒ
160	a0	240	á
161	a1	241	í
162	a2	242	ó
163	a3	243	ú
164	a4	244	ñ
165	a5	245	Ñ
166	a6	246	ª
167	a7	247	º
168	a8	250	¿
169	a9	251	©
170	aa	252	Ñ
171	ab	253	´

Extended ASCII Characters			
Dec	Hex	Oct	Character
172	ac	254	¨
173	ad	255	¡
174	ae	256	
175	af	257	»
176	b0	260	∞
177	b1	261	±
178	b2	262	≤
179	b3	263	≥
180	b4	264	¥
181	b5	265	*μ*
182	b6	266	∂
183	b7	267	∑
184	b8	270	∏
185	b9	271	π
186	ba	272	∫
187	bb	273	a
188	bc	274	o
189	bd	275	Ω
190	be	276	æ
191	bf	277	ø
192	c0	300	¿
193	c1	301	¡
194	c2	302	¬
195	c3	303	√
196	c4	304	*ƒ*
197	c5	305	≈
198	c6	306	Δ
199	c7	307	
200	c8	310	»
201	c9	311	…
202	ca	312	_
203	cb	313	À
204	cc	314	Ã
205	cd	315	Õ
206	ce	316	Œ
207	cf	317	œ
208	d0	320	–
209	d1	321	—
210	d2	322	“
211	d3	323	”
212	d4	324	‘
213	d5	325	’

Extended ASCII Characters			
Dec	Hex	Oct	Character
214	d6	326	÷
215	d7	327	◊
216	d8	330	ÿ
217	d9	331	Ÿ
218	da	332	/
219	db	333	
220	dc	334	‹
221	dd	335	›
222	de	336	fi
223	df	337	fl
224	e0	340	‡
225	e1	341	·
226	e2	342	‚
227	e3	343	„
228	e4	344	‰
229	e5	345	Â
230	e6	346	Ê
231	e7	347	Á
232	e8	350	Ë
233	e9	351	È
234	ea	352	Í
235	eb	353	Î
236	ec	354	Ï
237	ed	355	Ì
238	ee	356	Ó
239	ef	357	Ô
240	f0	360	
241	f1	361	Ò
242	f2	362	Ú
243	f3	363	Û
244	f4	364	Ù
245	f5	365	ı
246	f6	366	ˆ
247	f7	367	˜
248	f8	370	¯
249	f9	371	˘
250	fa	372	˙
251	fb	373	˚
252	fc	374	¸
253	fd	375	˝
254	fe	376	˛
255	ff	377	

This page intentionally left blank

APPENDIX

B Operator Precedence and Associativity

The operators are shown in order of precedence, from highest to lowest.

Operator	Associativity
`::`	unary: left to right binary: right to left
`()  []  ->  .`	left to right
`++  --  +  -  !  ~  (type)  *  &  sizeof`	right to left
`*  /  %`	left to right
`+  -`	left to right
`<<  >>`	left to right
`<  <=  >  >=`	left to right
`==  !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=`	right to left
`,`	left to right

This page intentionally left blank

INDEX

Symbols and Numerics

symbol, 10, 28, 30, 81, 99
$ (delimiter), 853
%= (combined assignment operator), 102
% (modulus operator), 61, 62, 83, 1088
&& (AND operator), 187–189
& (address operator), 620, 628
& (reference variable), 351
*= (combined assignment operator), 102, 710
* (indirection operator), 621
* (multiplication operator), 61, 83
* (pointer variable), 621
++ (increment operator), 243–248
+= (combined assignment operator), 102, 807
+ (addition operator), 61, 83, 712, 716
+ (concatenation operator), 120, 807
( ) (parentheses), 30, 84
:: (scope resolution operator), 396
; (semicolon), 30, 36, 100
<< (stream insertion operator), 32, 133, 832, 863
<= (less than or equal to operator), 158, 159, 815–817
< (less than operator), 158, 712, 713
<> (angle brackets), 427, 1083
== (equal to operator), 158, 168, 713
= (assignment operator), 14, 39, 59
{ } (braces), 29, 30, 66
-- (decrement operator), 243–248
/ (forward slash), 35
/ (division operator), 61–62
// (double slash), 28, 30
/= (combined assignment operator), 102
/* */ (multi-line comments), 64
\ (backslash), 34, 35
\\ (backslash escape sequence), 35
\' (single quote escape sequence), 35
\" (double quote escape sequence), 35
- (negation operator), 61
- (subtraction operator), 61, 83, 713
-> (structure pointer operator), 653, 659–660
-= (combined assignment operator), 102
"" (quotation marks), 30, 39, 50
>= (greater than or equal to operator), 158, 159, 815–817
> (greater than operator), 158
>> (stream extraction operator), 76, 134–136, 832
! (NOT operator), 187, 191–193
!= (not equal to operator), 158, 159, 815–817
[ ] operator, 721–727
?: (conditional operator), 206–209
? (question mark), 488
~ (tilde character), 407
| (bitwise OR operator), 832, Appendix H
|| (OR operator), 187, 189–191
2D arrays. *See* two-dimensional arrays

A

\a (alarm escape sequence), 35
abs function, 126
abstract base classes, 946–949
abstract data types (ADTs), 387–388
abstract functions, 946
access specification
  base class, 742, 745–748
  member, 747
  private, 745, 748
  protected, 748
  public, 745, 748
access specifiers, 392
accessors, 395
accumulators, 269–270
  summing values with, 505
Ackermann's function, 912
actual arguments, 317
actual parameters, 317
add member function, 710
  NumberList class, 1030–1031, 1033, 1036–1037
  recursive, 1049–1050
addition operator (+), 61, 83, 712, 716
addNumber function, 413
address operator (&), 620, 628
addresses
  array, 518–519, 637
  defined, 4
  pointer, 628, 632–633
  print format, 620
  string literal, 773
  variable, 620
ADTs. *See* abstract data types
Advanced Software Enterprises case study, 817–818

aggregation
defined, 734
has-a relation, 734, 739
through pointers, 736–737
`algorithm` header file, 989, 1009
algorithms
analysis of, 601–609
asymptotic complexity, 607–609
average case complexity, 607
complexity, 603–605
computational problems, 602–603
defined, 7
efficiency of, 601
encryption, 957
enumeration, 922–926
exhaustive, 922–926
programmer-defined, 537
QuickSort, 915–918
run time, 609
search, 577–584
sorting, 587–595
STL, 1008–1009
worst case complexity, 605–607
ALU (arithmetic and logic unit), 3
American Standard Code for Information Interchange. *See* ASCII
ampersand (&)
address operator, 620, 628
reference variables, 351
analysis of algorithms
basic steps, 602–603
big O notation, 607–609
experimental approach, 601–602
introduction to, 601–609
space criterion, 601
time criterion, 601
AND operator (&&), 187–189
defined, 187
precedence, 194
short circuit evaluation, 190
truth table, 188
angle brackets (<>), 427, 1083
animation, 1062
`append` function, 809
application frameworks, 957, 958
application software, 5
arguments
actual, 317
command line, Appendix F
default, 347–350
defined, 85, 316
floating-point, 85
formal, 317
passing, 319
passing array addresses as, 637
passing arrays as, 517–526
passing by reference, 351, 354–356
passing by value, 321–324, 354–356, 444
passing C-strings as, 776, 803–805
passing objects as, 413–420
passing structures as, 444–446
passing to base class constructors, 751–753
arithmetic and logic unit (ALU), 3
arithmetic assignment operators. *See* combined assignment operators
arithmetic operators, 61–63
associativity, 84
list of, 61
overloading, 712–715
precedence, 83
arithmetic, pointer, 629–630
array contents
accessing, 487–489
copying, 504
displaying, 489–493
inputting, 489–492
processing, 502–513
writing to files, 493
arrays, 485–561
accessing elements, 487–489
addresses, 518–519, 637
bounds checking, lack of, 493–494, 627
circular, 1088
of class objects, 549–555
comparing, 504–505
copying from one to another, 504
C-string defined as, 774
defined, 120, 485
defining, 486
elements, 486, 487–489
finding average value in, 505–506
finding highest value in, 506–508
finding lowest value in, 506–508
as function arguments, 517–526
functions, 523–525
implicit sizing, 501–502
initialization, 496–502
initialization list, 497, 502
initialization, partial, 500–501
linked lists versus, 1023–1024
memory requirements, 486–487
name, 486
name, as pointer constant, 629
off-by-one errors, 495
one-dimensional, 526
parallel, 513–516
partially-filled, 508
pointer relationship, 625–629
reading data from files into, 491–492
reasons for use, 509–511
searching for an element, 577–587
size declarators, 486, 487, 488
sorting data in, 587–595
string processing, 511–513
of structures, 555–558
subscripts, 487
summing values in, 505
three or more dimensional, 534–537
two-dimensional, 526–533
`typedef` statement with, 516–517
vectors versus 537–549
writing data to files, 493
ascending order, 587, 591–592
ASCII (American Standard Code for Information Interchange), 48, 51
character set table, 1141–1143
character values, 202
numbers, 787
`assign` function, 809
assignment
arithmetic, 102
combined, 102–104
copy constructors and, 705
default arguments, 348
derived class pointers, 936–937
initialization versus, 695
memberwise, 694–695, 699
multiple, 101–102
type cast, 936
variable, 59–60
assignment operator (=), 694
defined, 14, 39
operands, 59
overloading, 704–710, 814
return value, 709–710

assignment statements, 38, 81
associative containers, 538, 1003–1004. *See also* containers
  defined, 538, 1003
  map, 1003
  multimap, 1003
  multiset, 1004
  set, 1004
  STL, 1003
associativity
  of arithmetic operators, 84
  defined, 84
  of logical operators, 194
  operators, 1145
  of relational operators, 158
asymptotic complexity, 607–609
`at` member function, 548, 809, 1006
`atof` library function, 790
`atoi` library function, 790
`atol` library function, 790
attributes
  class, 456
  defined, 389, 456
  defining, 456
  object, 389
average-case complexity function, 607
`avgVector` function, 546

B

`\b` (backspace escape sequence), 35
`back` member function, 1006, 1055
backslash (\), 34, 35
`bad` member function, 847
`bad_alloc` exception, 645, 983
base case, 904, 913
base classes, 458
  abstract, 946–949
  access specification, 742, 745–748
  constructors, 750
  constructors, passing arguments to, 751–753
  destructor, 750
  functions, overriding, 755–757
  inheritance and, 740
  member functions, 747, 757
  for multiple derived classes, 742
  pointers, 936, 937, 939–940
  protected members, 745–748
BASIC, 9
basic steps, 602–603
`begin` member function, 809, 1004, 1006
behaviors
  class, 457
  polymorphic, 943
bidirectional iterator, 1004
big O notation, 607–609
binary digits (bits), 4
binary files, 862–866
  defined, 683
binary numbers, 7, Appendix H
binary operators, 61
  infix, 927
  postfix, 927
  prefix, 927
binary search, 580–583. *See also* search algorithms
  in array of objects, 584–587
  efficiency, 583–584
  index variables, 581
  pseudocode, 580–581
  as recursive function, 913–914
  worst case complexity, 605
binary search trees
  creating, 1115–1118
  defined, 1113
  deleting nodes, 1122, 1123
  illustrated, 1114
  information storage, 1113
  inorder traversal, 1119, 1120
  inserting elements into, 1118–1119
  left subtree, 1124
  nodes, 1114
  nonempty, 1118, 1119
  operation implementation, 1118
  operations, 1115–1131
  postorder traversal, 1119, 1120
  preorder traversal, 1119, 1120
  removing elements from, 1122–1130
  right subtree, 1124
  root nodes, 1114, 1118, 1119
  searching, 1114, 1121–1122
  templates, 1131
  traversing, 1119–1121
binary trees
  applications of, 1112–1114
  child nodes, 1111
  defined, 1111
  empty, 1111
  implementation, 1112
  leaf nodes, 1111
  parent, 1111
  pointer representation, 1112
  root, 1111
  subtrees, 1112
  templates, 1131
`binary_search` algorithm (STL), 1008, 1011
`binarySearch` function, 581
bit flags, 847
bit patterns, 44
bits (binary digits), 4
bitwise operations, Appendix H
  bitwise OR (|), 832
blank spaces, 106
block scope, 198
body
  function, 307
  loop, 250
`bool` data type, 57
Boole, George, 57
Boolean expressions, 57, 158
Boolean values, 332–334, 427
Boolean variables, 57, 165
  as flags, 169
  and the `!` operator, 192–193
bounds checking, lack of, 493–494, 627
braces (`{}`), 29, 30, 66
  in `if` statements, 164, 165, 167
  in `if/else if` statements, 177
  in `if/else` statements, 173
  in loops, 252–253
`break` statements
  for early loop termination, 279–280
  in nested loops, 281
  in `switch` statements, 211–212
bubble sort, 587–591. *See also* sorting algorithms
`bubbleSort` function, 599
buffer overruns, 123
buffers
  file, 137
  flushing, 843
  input, 77
  keyboard, 77
  output, 843
`byteNum` function, 876
bytes
  in binary numeric representation, 863
  defined, 4
  number of, 865

C

C# programming language, 9
C++
  case sensitivity, 29
  defined, 9
  portability, 9
  prestandard, 37
  program parts, 27–31
  programming language, 2, 7, 9
  standard, 37
C programming language, 9
calling functions. *See* function calls
`capacity` member function, 548, 809, 1006
case conversion, 797–799
case sensitivity, 29, 43
`case` statements, 210–211, 212
case studies
  Advanced Software Enterprises, 817–818
  Central Mountain Credit Union, 287–291
  Crazy Al's Computer Emporium, 229
  Creating an Abstract Array Data Type, 561, 609
  Demetris Leadership Center, 609
  General Crates, Inc., 142
  Green Fields Landscaping (part 1), 140–142
  Green Fields Landscaping (part 2), 225–229
  High Adventure Travel Agency, 373
  Home Software Company, 449–455
  Lightening Lanes, 291
  Little Lotto, 368–373
  National Commerce Bank, 559–561
  Online Friendship Connections, 883–888
  Reliable Software Systems, Inc., 1056–1060
  Secure Encryption Systems, Inc., 957–960
  Set Intersection, 561
  United Cause Relief Agency, 661–669
casts. *See* typecast expressions
`cctype` header file, 431
CD (compact disc) drives, 5
cells, 464
  placing cursor in, 465
Central Mountain Credit Union case study, 287–291
central processing unit (CPU), 3–4, 7
`char` data type, 48–52
  pointers to, 772, 775–776
  size, 48
character data types, 43, 48–52
character testing functions, 793–797
  defined, 793
  `isalnum`, 793
  `isalpha`, 192–193, 793, 796
  `isdigit`, 793, 797
  `islower`, 793
  `isprint`, 793
  `ispunct`, 794
  `isspace`, 794
  `isupper`, 794
characters
  alphabetic, testing for, 193
  ASCII values, 202
  case conversion, 797–799
  comparing, 202–204
  constants, 49–52
  fill, 843
  inputting, 117
  memory storage, 48
  null, 50, 121, 771
  printable, 853
  special, 30
  valid range, 433
  value assignment, 115
  whitespace, 115, 849
  working with, 115–120
`CharRange` input validation class, 432–435
`cin.get`, 117–119
`cin.ignore`, 119
`cin` object, 75–81
  >> (stream extraction operator), 76, 117, 118–119
  C-strings and, 121
  entering multiple values, 78–79
  field widths and, 124
  `getline` function, 124–125
  `ignore` function, 119
  input, 77
  `iostream` file and, 78
  mixing `cin >>` and `cin.get`, 118–119
  prompt, 76
  reading single characters with, 117
  `width` function, 123
cipher text, 957
`Circle` class, 392–395
circular arrays, 1088
circular linked lists, 1054
class templates, 993–1002
  declaration, 993
  defined, 993
  derived from class templates, 1000–1001
  inheritance and, 999–1001
  member function definition, 994
classes. *See also* member functions; member variables
  abstract, 429, 946–949
  access, 458
  access specifiers, 392
  aggregation, 734–739
  attributes, 389, 456
  base, 458, 740, 742
  behaviors, 457
  composition, 734–739, 952–957
  constructors, 402–407, 440
  declaration format, 391
  declarations, 393, 424, 450–451, 742
  defined, 391
  defining by inheritance, 741
  derived, 458, 740–741
  destructors, 407–410
  exception, 977, 981–982
  finding, 458–461
  friends of, 689–693
  identifying, 456
  implementation, 391, 452–455
  implementation file, 424, 430
  inheritance, 458, 952–957
  inline functions, 395, 401
  input validation, 432–435
  instances, 393
  instantiation, 393
  interface, 452
  introduction to, 391–392
  I/O, 430
  inline functions, 395, 401
  member variables, 389
  member functions, 389, 395–401
  name, 391
  object-oriented exception handling with, 977–978
  ownership, 458
  pointer declaration to, 652
  private member functions, 410–413

private members, 392
public members, 392
relationships between, 421, 457–458
responsibilities, 461–463
specification file, 424, 425, 430
subclass, 744
superclass, 744
`clear` member function
file bit flags, 847
`IntQueue` class, 1089
`string` class, 809
`vector` class, 545, 548, 1006
`close` function, 137
`cmath` header file, 86, 126
COBOL, 9
code reuse, 306
columns, two-dimensional array, 533
combined assignment operators, 102–104
overloaded, 710, 814
command line arguments, Appendix F
comments, 64–65
/* as beginning indication, 64
// as beginning indication, 28, 30, 64
defined, 64
function, 309
multi-line, 64–65
single-line, 64
compact disc (CD) drives, 5
`compare` function, 809
comparisons
addresses, 663
array, 504–505
bubble sort, 588, 589
characters, 202–204
C-strings, 206, 778–779
enumerated variables, 220
floating-point numbers, 169–171
pointers, 632–634
strings, 204–205
structure members, 438
compilation errors, 20
compilers, 10, 65–66
convert constructor invocation, 732
dynamic binding and, 946
Microsoft Visual C++ tutorial, Appendix M
MinGW C++ tutorial, Appendix L
prestandard, 109
static binding and, 945–946
string literals and, 772, 779
compiling, 10, 11, 20
multi-file programs, 127–128
complexity, 603–609
asymptotic, 607–608
average case, 607
classes, 608
measurement, 603–605
time, 609
worst case, 605–607
composition
aggregation and, 737
class, 734–739
class relationship, 457
defined, 737
inheritance versus, 952–957
object, 420–424
compound operators. *See* combined assignment operators
computational problems, 602–603
computers
hardware, 2–5
software, 5–6
systems, 2–6
usefulness of, 1–2
concatenation
defined, 120
operator (+), 120, 807
condition
`if` statement, 163–165
`while` statement, 249–251
conditional execution, 163
conditional expressions, 206–209
conditional operator (`?:`), 206–209
consoles
defined, 31
input, 75
operating system functions, 464
output, 31
`const` key word, 98, 99, 641, 680
`const` parameters, 703
`const` pointers, 641–643
constant factor, 607
constant member functions, 677, 680–681
constant pointers
to constants, 642–643
defined, 641
initializing, 642
constant reference parameters, 416–417
constant time, 609
constant variables. *See* named constants
constants
address, passing, 640
character, 49–52
constant pointers to, 642–643
defined, 37
floating-point, 55–56
global, 340–343
hexadecimal, 47
integer, 40, 47
named, 97–101
octal, 47
pointers to, 639–640
string, 40, 49–52
uses, 40
`Constructor` function, 1074
constructors, 402–407
base class, 750–753
convert, 715, 731–734, 995
copy, 695–704
default, 405, 406–407
defined, 402
defining, 402–404
derived class, 750, 753
file stream, 835
as inline function, 403
overloading, 405–406
containers. *See also* Standard Template Library (STL)
associative, 538, 1003–1004
defined, 538, 1002
`deque`, 1003, 1096–1097
`list`, 1003, 1054–1056
`queue`, 1097–1098
sequential, 538, 1002–1003
`stack`, 1082–1084
`vector`, 1003, 1005–1007
`continue` statement, 281–284
control unit, 3
control structures
branching, 162–164,172, 176, 210
looping, 249, 259, 263–264
control variables, 511
conversion
algebraic expression to C++, 84–85
character case, 431–432, 797–799
data type, 89–94

conversion (*continued*)
explicit type, 90–94
floating-point to integer, 56–57, 90, 91
implicit type, 89–90
integer to floating-point, 90
number to string, 40
type conversion operators, 728–730
string/numeric functions, 790–792
convert constructors, 715, 731–734, 995
C-strings and, 734
defined, 731
examples, 731
invocation, 731
uses, 733–734
coordinates, cell, 464
copy constructors
assignment and, 705
calling, 703–704
default, 696–697
default, deficiencies, 697–700
defined, 695, 696
invocation of, 703–704
object initialization and, 704
parameter, 702
passing by value and, 702
programmer-defined, 700–703
`copy` function, 809
`cos` function, 126
`count` algorithm (STL), 1008, 1011
`countChars` function, 803–805
count-controlled loops, 263
counters, 256–258
defined, 256
`for` loop, 264–265
`cout` object
C-strings and, 121
defined, 31
`endl` stream manipulator and, 33–34
expression value display, 82
in formatting data, 105–114
`iostream` object and, 36
output display with, 32
as stream object, 32
`.cpp` file, 424–429
CPU (central processing unit), 3–4, 7
Crazy Al's Computer Emporium case study, 229
`createItem` function, 417
Creating an Abstract Array Data Type case study, 561, 609
`cstdlib` header file, 128, 364
`c_str` function, 809
C-strings
arguments, passing, 776
arguments, passing with pointers, 803–805
assigning values to, 122
comparing, 206, 778–779
convert constructors and, 734
defined, 120, 771
defined as array, 774
displaying, 121
forms, 772
functions, writing, 800–805
inputting, 121
library functions, 776–787
memory allocation, 122–124
pointer to `char`, 772, 775–776
reading line of input, 124–125
reasons for using, 121
size declarator, 120
storage, 512
string literals holding, 773
using, 120–125
C-style type cast expressions, 93
`ctime` header file, 129
current directory, 832
cursors, positioning on screen, 464–467

D

dangling pointers, 646
data abstraction, 388
data hiding, 390
data types
abstract, 387–388
`bool`, 57
`char`, 48–52
character, 43, 48–52
conversion, 728
defined, 43
demoted values, 89–90
`double`, 54, 55
enumerated, 219–222
`float`, 54, 55
floating-point, 43, 54–57
`fstream`, 131
`ifstream`, 131, 274
`int`, 45
integer, 43–48
`long`, 45
`long double`, 55
numeric, 43–47
`ofstream`, 131
programmer-defined, 537
promoted values, 89–90
ranking, 89
return values, 329, 331
selecting, 44
self-referential, 1025
`short`, 45
size, 44
size determination, 45, 58–59
`string` class used as, 52–54
structure, 435
`unsigned int`, 45
`unsigned long`, 45
`unsigned short`, 44, 45
`vector`, 537
database management systems (DBMSs), 829
debugging, 17, 21
hand tracing, 138–140
nested `if/else` statements, 185
using good test data, 284–286
validating output results, 223–225
`dec` manipulator, 841
decimal notation, 55–56, 111
decision structure
defined, 163
nested, 184
declaration
class, 391, 393, 424, 450–451, 742
class template, 993
definition versus, 685
enumerated data type, 219
friend function, 689
linked list nodes, 1026–1027
named constants, 97–98
pointer, 621–622
static member, 685
structure, 435–436
union, 447
virtual function, 943
decrement, 243
decrement operator (--)
with arrays, 502
in `cout` statements, 246
defined, 244
in mathematical expressions, 247–248
postfix mode, 244, 245–247
prefix mode, 244, 245–247
in relational expressions, 248

decryption, 957, 958
default arguments
assignment, 348
defined, 347
example use, 348–349
overriding, 348
parameter mixture of, 350
summary, 350
uses, 348
values, 350
default constructors, 405, 406–407
default copy constructors, 696–700
default directory, 832
#define directive, 98–100
definition statements, 393
delete operator, 645, 646–648, 649, 657
Demetris Leadership Center case study, 609
demoted values, 89–90
deque container, 1003, 1096–1097
defined, 1003
member functions, 1096
dequeue, 1085, 1086
dequeue function, 1092
dereferencing pointers, 623
derived classes, 458
constructors, 750, 753
destructor, 750
inheritance and, 740–741
objects, 743
overriding base class member functions, 757
pointers, 936, 937
descending order, 587, 591–592
deserializing objects, 884
design
object-oriented, 455–463
program, 17–21
top-down, 18, 21
desk-checking, 20–21
destructors
base class, 750
characteristics, 408
defined, 407
derived class, 750
inheritance and, 740
tilde character (~), 407
uses, 408
direct recursion, 907
directives
#define, 98–100
defined, 10
#endif, 425
#include, 28, 35–36, 425, 427
#ifndef, 425
preprocessor, 28, 36
directories, 832
disk drives, 5
display function, 884
displayList function, 1032
displayMenu function, 336
displayMessage function, 307, 308
displayStars function, 349
displayValue function, 316–317, 318, 319
divide and conquer, 305
divide function, 975, 976
division, 61, 62
integer, 62
division operator (/), 61
precedence, 83
division-by-zero error, 973
documentation, 2, 64, 97–98, 309. *See also* comments
dot operator (.), 394, 436, 438
double data type, 54, 55
double precision, 54
doubleArray function, 521
doubleNum function, 351, 353
doubleValue function, 635, 637
doubly linked lists, 1054
do-while loops. *See also* loops
defined, 258
flowchart, 259
format, 259
iteration, 259
with menus, 260–262
as post test loop, 259
termination, 259
use decision, 277
uses, 259
while loop versus, 259
drivers, 367–368
dummy functions, 366
dummy names, 321
DVD (digital versatile disc) drives, 5
dynamic binding, 946
dynamic memory allocation, 643–648. *See also* memory
in class objects, 652, 653–655
defined, 643, 644
requests, 644, 645
throwing an exception, 645
dynamic queues, 1092–1095. *See also* queues
linked list implementation, 1092
dynamic stacks, 1072, 1073, 1079–1082. *See also* stacks
linked list implementation, 1079–1082
DynIntStack class, 1079–1082

E

E notation, 54
EBCDIC, 48
elements, array, 486
accessing, 487–489
address, obtaining, 628
assignment statements, 488
class objects, 549–559
initialization, 497
number of, 487
passed to functions, 517
processing, 502–513
in relational expressions, 503
sorting, 596–599
structures, 555–558
subscripts, 487
two-dimensional array, 532
unknown contents, 488
elements, vector, 541
adding, 541–542
number of, 543
removing, 544–546
empty member function
list container, 1055
stack container, 1083
string class, 809
vector class, 546–548, 1006
empty queues, 1086, 1088
encapsulation, 389, 955
encryption, 957, 958
end member function, 809, 1004, 1006, 1055
end of file mark, 273
#endif directive, 425
endl stream manipulator, 33–34, 39, 841, 843
enqueue, 1085
enqueue function, 1092
enumerated data types, 219–222
declaration, 219
defined, 219
relational operator comparison, 220
symbolic names, 219, 220
symbols, 220
values, 219

enumeration algorithms, 922–926
defined, 922, 923
recursive techniques in, 923–924
environment variables, 819
`eof` function, 275, 836, 837, 847
equal to operator (==), 158, 168
overloading, 713
`erase` function, 810, 1006, 1055
errors
buffer overrun, 123
compiler reported, 10, 20
correcting, 20
division-by-zero, 973
file open, 222
`if` statement, 166–168
logical, 20
off-by-one, 495
round-off, 291
run-time, 20
syntax, 20
testing for, 973
escape sequences, 34–35
exception handlers, 974
exceptions, 973–985
`bad_alloc`, 645, 983
catch block, 975
classes, 977
classes, information extraction from, 981–982
default action, 645
defined, 645, 973, 974
handling, 975–976
multiple, 979–980
object-oriented handling with classes, 977–978
overflow, 1074, 1092
rethrowing, 984–985
stack, 1078–1079
static queue, 1092
throw point, 974
throwing, 974
try block, 975
try/catch construct, 975–976
uncaught, 976
underflow, 1074, 1092
executable code, 10
executable files, 10
linking programs to create, 20
exhaustive algorithms, 922–926
recursive techniques in, 923–924
`exit` function, 363–364
`EXIT_FAILURE`, 364
`EXIT_SUCCESS`, 364
`exp` function, 126
explicit type conversion, 90–94
expressions
algebraic, 84, 85
Boolean, 57, 158
conditional, 206–209
defined, 81
infix, 927, 1103
initialization, 263, 266, 267–268
mathematical, 81–89
multiplication, 84
postfix, 927, 1103
prefix, 927
relational, 158, 159
test, 263
type cast, 90–94
update, 263, 266, 267–268

F

`factorial` function, 908–910
factorials, 908
`fail` member function, 222, 836, 847
`false`, 159, 274, 275, 332–333
fetch/decode/execute cycle, 4
Fibonacci numbers, 911
Fibonacci sequence, 911–912
field widths, 843
`cin` object and, 124
defined, 106
floating-point numbers, 108
specifying, 107
strings, 108
fields
defined, 866
information organization in, 867
key, 584
file access
random, 871–878
sequential, 871
file buffers, 137
file names, 133
conventions, 830
extensions, 830
string objects for, 831
variable, using, 831
file positioning flags, 872
file stream objects. *See also* `ifstream` class; `ofstream` class
classes, 830
constructors, 835
defined, 131, 830
defining, 131–132
extraction operator (>>) and, 832
insertion operator (<<) and, 832
passing to functions, 845–846
read position, 135
testing for file open errors, 222
files. *See also* `ifstream` class; `ofstream` class
binary, 862–866
class implementation, 424
class specification, 424
closing, 137
`.cpp`, 424–429
end of, testing for, 836–837
executable, 10, 20
`fstream`, 831, 832
`.h`, 424–429
header, 28, 37
`ifstream`, 831
including in another file, 28, 35–36
input, 131, 831
introduction, 130–138
`ios::app`, 834, 835
`ios::badbit`, 847
`ios::beg`, 872
`ios::binary`, 835, 864, 867
`ios::cur`, 872
`ios::end`, 872
`ios::eofbit`, 847
`ios::failbit`, 847
`ios::goodbit`, 847
`ios::hardfail`, 847
`ios::in`, 832, 833, 835, 878
`ios::out`, 833, 834, 835, 864, 878
`ios::trunc`, 835
`iostream`, 28, 29, 36
moving around in, 859
multi-file projects, 424–430, Appendix J
naming conventions, 424
object, 10
`ofstream`, 831
open errors, testing for, 222
open limits, 137
open modes, 832
opening, 132–133, 834
opening, for input and output, 878–882
operation errors, testing for, 835–836
output, 131, 831
passing, to functions, 356–357

path, 132
program location, 831–832
random-access, 871–878
reading, 134–137, 832–835
reading data into arrays, 491–492
reading, member functions, 849–862
reading with loops, 273–276
rewinding, 858–860
source, 10
text, 863
use steps, 130–131
using, 133–137, 830–831
writing, 133–134, 832–835
writing array contents to, 493
writing, member functions, 849–862
fill character, 843
`find` algorithm (STL), 1008, 1013
`find` function, 810
finding classes, 456, 458–461
first-in, first-out (FIFO) order, 1084–1085
`fixed` manipulator, 111, 841
fixed-point notation, 111
flags
bit, 847
defined, 169
file open modes, 832
file positioning, 872
format, 843, 844
manipulator, 843
mode, 872
flash memory, 5
`float` data type, 54, 55
floating-point
arguments, 85
constants, 55–56
data types, 43, 54–57
division, 710
output, 111
representions, 54
values, assigning to integer variables, 56–57
floating-point numbers
comparing, 169–171
defined, 54
double-precision, 16
field width, 108
identifying, 44
precision, 843
floppy disk drives, 5
flowcharts, 19, Appendix O
decision structure, 163
`do-while` loop, 259
`for` loop, 264
`if/else if` statement, 176
`if/else` statement, 172
nested `if` statements, 183
`while` loop, 249
`flush` manipulator, 841, 843
flushing buffers, 843
`fmod` function, 126
`for` loops, 263–269. *See also* loops
body, 250
counter, 264–265
defined, 263
event sequence, 264
expressions, omitting, 268
flowchart, 264
format, 263
header, 263
initialization expression, 263, 267–268
defining variable in, 266
as pretest loop, 265
test expression, 263
update expression, 263, 266
use decision, 277
user-controlled, 267
`for_each` algorithm (STL), 1008, 1014–1015
formal arguments, 317
formal parameters, 317
format flags, 843, 844
formatting
defined, 105, 863
member functions, 843–844
output, 105–114, 838–844
FORTRAN, 9
forward iterator, 1004
friend classes, 693
friend functions, 689–693
declaration, 689
defined, 689
list, 689
`front` member function
`deque` container, 1096
`list` container, 1055
`queue` container, 1097
`vector` class, 1006
`fstream` class
data type, 878–882
objects, 831, 832
`fstream` header file, 131
function calls
defined, 306
overloaded functions, 360
recursive, 902
statements, 307, 309
static member functions, 688
syntax, 321
value-returning function, 328–331
function prototypes
defined, 314, 315
dummy names, 321
`main` and, 316
parameter data type, 320
placement, 315
value-returning functions, 328
function templates
defined, 985
defining, 993
example, 986
as function specification, 988
with multiple types, 991
operators in, 990–991
overloading with, 992–993
placement, 988
prefix, 986
type parameters, 985–986, 987, 990, 992
uses, 985
functional notation, 93
functions
abstract, 946
arguments, 316, 319, 320–321, 337, 347–350, 354–356, 517–526
array, 523–525
average-case complexity, 607
base class overriding, 755–757
body, 307
calling, 306–313
character conversion, 797
character testing, 793–797
comments, 309
constructor, 402–407
C-string, writing, 800–805
default arguments, 347–350
defined, 29, 305
defining, 306–307
drivers, 367–368
dummy, 366
execution, 307

functions (*continued*)
friend, 689–693
global variables, 339–340, 343–344
header, 307, 308
implementation section, 395
inline, 395, 400–401
library, 85, 126–130, Appendix G
local variables, 337–339, 343–347
member, 120, 389, 395–401
in menu-driven programs, 334–337
name, 306
operator, 705, 706
overloading, 359–363
parameter list, 306
parameters, 314, 316, 635–639
passing data by reference, 350–358
passing data by value, 321–324, 354–358, 444
passing file stream objects to, 845–846
passing files to, 356–357
passing objects to, 413–420
passing structures to, 444–446
passing two-dimensional arrays to, 530–532
polymorphic, 941
recursive, 901–907, 1046–1049
return type, 307
return values, 326–331
returning Boolean values, 332–334
returning objects from, 417–419
returning pointers from, 648–650
returning structures from, 446–447
sending data into, 316–321
signature, 360
static local variables, 344–347
static member, 685–688
stubs, 366–367
termination, 325
uses, 305–306
value-returning, 326–331
virtual, 941–946
void, 307
worst-case complexity, 605–607

G

gcd recursive function, 910–911
General Crates, Inc. case study, 142
generalization, 740
get (getter) functions, 395
`get` member function, 854–855
`cin` object, 118
`peek` member function versus, 855
versions, 855
`getArea` function, 391, 395, 397, 400
`getBasePay` function, 341, 343
`getChar` function, 434
`getChoice` function, 336
`getFeet` function, 712
`getHighest` function, 523
`getInches` function, 712
`getInput` function, 979
`getLength` function, 397
`getline` library function
arguments, 125
`cin` object, 124–125
defined, 116
file stream objects and, 832
`getline` member function, 774, 851–854
arguments, 851, 852
delimiter use, 854
file stream objects and, 832
function call, 851
`getLowest` function, 523
`getName` function, 649, 755, 942
`getNum` function, 352
`getNumber` function, 637
`getOvertimePay` function, 341
`getPrice` method, 584
`getQuality` function, 682
`getRadius` function, 329, 331
`getRandomNumbers` function, 649
`getSales` function, 639
`getValue` function, 678
`getWidth` function, 397
global constants, 340–343
global variables, 339–340. *See also* variables
defined, 337
initializing, 340
problems, 341
with same name, 343–344
scope, 339
`good` member function, 847
greater than operator (>), 158
greater than or equal to operator (>=), 158, 159, 815–817
Green Fields Landscaping case study
part 1, 140–142
part 2, 225–229
grouping, with parentheses, 84

H

`.h` file, 424–429
hand tracing, 138–140
hard copy, 5
hardware
components, 2
CPU, 3–4
defined, 2
input devices, 5
main memory, 4
output devices, 5
secondary storage, 4–5
has-a relationship
class composition, 953
defined, 734
head, linked list, 1024, 1045
header files, Appendix G
`algorithm`, 989, 1009
`cctype`, 431
`cmath`, 86, 126
`cstdlib`, 128, 364
`ctime`, 129
defined, 28
for file input/output, 131
`fstream`, 131
`.h`, 424–429
`iomanip`, 107, 110
`iostream`, 78
older style, 37
`string`, 52, 806
table, Appendix G
`vector`, 538
heap, 645
`hex` manipulator, 841
hexadecimal constants, 47
hierarchies, inheritance, 935–936
hierarchy charts, 18, 19, 369
High Adventure Travel Agency case study, 373
high-level languages, 8
Hoare, C. A. R., 915
Home Software Company case study, 449–455

I

identifiers
defined, 41
legal, 42–43
rules, 42–43

IDEs (integrated development environments), 11, 132
Microsoft Visual C++ IDE tutorial, Appendix M
wxDev-C++ IDE tutorial, Appendix L
`if` statements, 162–172
braces, 164, 165, 167
conditions, 165
defined, 163
errors, 166–168
example, 164, 165
flags and, 169
floating-point number comparison, 169–171
nested, 183–187
programming style and, 166
relational expressions and, 168–169
semicolons and, 166–167
`if/else` `if` statement, 175–181
braces, 177
characteristics, 177–178
construction, 176
defined, 175
flowchart, 176
nested, 185
paths, 178
trailing `else`, 179–181
`if/else` statements, 172–175
braces, 173
defined, 172
flowchart, 172
paths of execution, 173
programming style, 173
`#ifndef` directive, 425
`ifstream` class. *See also* file stream objects
data type, 131, 274
file open mode, 834
`get` member function, 854
`getline` member function, 851
objects, 831
`peek` member function, 853–857, 928
`read` member function, 865
`seekg` member function, 859, 878
`tellg` member function, 876–878
`ignore` function, 119
image maps, 1060–1061
implementation file, class, 424, 430
implicit array sizing, 501–502
implicit pointers, 870
implicit type conversion, 89–90
`#include` directive, 28, 35–36
in multi-file programs, 424–429, Appendix J
preprocessor and, 36
include file directory, 427
include guard, 425
increment, 243
increment operator (++)
with arrays, 502
in `cout` statements, 246
defined, 244
in mathematical expressions, 247–248
postfix mode, 25–27, 244
prefix mode, 244, 245–247
in relational expressions, 248
indentation, 167
index variables, 581
indirect recursion, 907
indirection operator (*)
defined, 14
in pointer declaration, 621–622, 637
infinite loops, 252
infix expressions, 927, 1103
inheritance, 740–745
base class, 740
class access, 745–748
class relationship, 458
class templates and, 999–1001
composition versus, 952–957
constructors, destructors, and, 750–754
defined, 952
defining classes by, 741
derived class, 740–741
hierarchies, 935–936
is-a relationship, 740–744
multiple, 744, Appendix K
overriding base class functions, 756–757
`private` base class, 747–748
`protected` base class, 747–748
`public` base class, 747–748
type compatibility in, 936–939
virtual, Appendix K
initialization
array, 488, 496–498, 529–530
array of objects, 552–553
array of structures, 557–558
array, partial, 500–501
assignment versus, 695
constant pointer, 642
with constructors, 402–407
expression, 263–264
global variable, 340
with initialization list, 438–440, 497–498, 529–530
linked list node, 1026–1027
local variable, 339
member variable, 440
named constant, 97
object, 402–407
pointer, 630–631
static local variable, 346
structure, 438–440
two-dimensional array, 529
variable, 59–60
vector, 538–539
initialization expressions
defined, 263
multiple statements in, 267–268
omitting, 268
variable definition in, 266
initialization lists
array, 497, 502
array of objects, 552
defined, 438
drawbacks, 440
items, 439
member, 734–736
specifying, 502
structure, 438–440
two-dimensional array, 530
inline expansion, 401
inline member functions, 395, 400–401
inner loops, 277, 279
inorder traversal, 1119, 1120
input
buffer, 77
`cin` object, 77
in class object, 430
console, 75
defined, 16
devices, 5
file, 131, 831
iterator, 1004
opening files for, 878–882
reading line of, 124–125
test data, 20

input validation, 183
class, 432–435
defined, 196, 253
examples, 196
logic, 254
objects, 431–435
user, 196–197
`while` loops for, 253–256
`inputFile` object, 132
input-output stream library, 36
inputting
array contents, 489–492
characters, 117
C-strings, 121
strings, 115–117
`insert` function, 810, 1055
instance members, 678, 685
instances
computational problem, 602
defined, 393, 678
member variable, 420
instantiation, 393
instructions, 3, 4, 6
`int` data type, 45, 729
`IntArray` class, 724
`IntBinarySearchTree` class, 1121
`IntBinaryTree` class, 1115–1117, 1120
`IntClass` class, 731–732
integers
constants, 40, 47
data types, 43–48
division, 62, 710
variables, 56–57
integrated development environments (IDEs), 11, 132
wxDev-C++ IDE tutorial, Appendix L
Microsoft Visual C++ IDE tutorial, Appendix M
`IntQueue` class, 1089–1092
`IntRange` class, 981
`IntStack` class, 1073–1079
I/O manipulators. *See* manipulators
`iomanip` header file, 107, 110
`ios::app`, 834, 835
`ios::ate`, 835
`ios::badbit`, 847
`ios::beg`, 872
`ios::binary`, 835, 864, 867
`ios::cur`, 872
`ios::end`, 872
`ios::eofbit`, 847
`ios::failbit`, 847
`ios::goodbit`, 847
`ios::hardfail`, 847
`ios::in`, 832, 833, 835, 878
`ios::nocreate`, 878
`ios::out`, 833, 834, 835, 864, 878
`ios::trunc`, 835
`ios` class, 832
`iostream` header file, 28, 36
is-a relationship, 740–744
class inheritance, 952–953
defined, 740
examples, 740
`isalnum` library function, 793
`isalpha` library function, 192–193, 793, 796
`isdigit` library function, 793, 797
`isEmpty` function, 1074, 1078
`isEven` function, 332, 333
`isFull` function, 1079
`islower` library function, 793
`isprint` library function, 793
`ispunct` library function, 794
`isspace` library function, 794
`istream` class, 865
`istringstream` class, 788
`isupper` library function, 794
`isValid` function, 332
iteration
defined, 251
loop, 251, 259, 279
overhead, 926
recursion versus, 926
iterators. *See also* Standard Template Library (STL)
defined, 1002
defining, 1004–1005
types, 1004
use of, 1004–1005
`itoa` library function, 790, 791

J

Java, 9
JavaScript, 9

K

key fields, 584
key words. *See also specific key words*
defined, 13
list of, 42
use of, 13–14
keyboard buffers, 77

L

last-in-first-out (LIFO) structures, 1071
leaf nodes, 1111
leaks, memory
defined, 646, 657
in loops, 646
stopping, 657
left child node, 1111
`left` manipulator, 112–113, 841
left to right associativity, 84
left-justified output, 113
`length` function, 120, 400, 512, 810
less than operator (<), 158
overloaded, 712, 713
less than or equal to operator (<=), 158, 159, 815–817
libraries. *See also* Standard Template Library (STL)
input-output stream, 36
run-time, 10, 126
library functions, 124–125, Appendix G
`abs`, 126
`atof`, 790
`atoi`, 790
`atol`, 790
`cos`,126
for C-strings, 776–787
defined, 85
`exp`, 126
`fmod`, 126
`isalnum`, 793
`isalpha`, 192–193, 793, 796
`isdigit`, 793, 797
`islower`, 793
`isprint`, 793
`ispunct`, 794
`isspace`, 794
`isupper`, 794
`itoa`, 790, 791
`log10`, 126
`log`, 126
mathematical, 126–130
`pow`, 85, 128
`rand`, 128
`round`, 126
`sin`, 126
`sqrt`, 126, 127
`srand`, 128
`strcat`, 776–777, 783
`strcmp`, 779–782, 783, 792

strcpy, 122, 778, 783
strlen, 776, 783
strncpy, 783
strstr, 783
tan, 126
time, 129
tolower, 797–798
toupper, 431–432, 433, 797–798
lifetime, local variable, 339
Lightening Lanes case study, 291
linear search. *See also* search algorithms
advantage, 580
in array of objects, 584–587
defined, 577
inefficiency, 580
pseudocode, 578
linear time, 609
lines, 14
linked lists, 1023–1060
adding elements to, 1031–1032
advantages, 1023–1024
building, 1027
C++ representation, 1024–1026
circular, 1054
defined, 1023
deleting nodes, 1030, 1032
destroying, 1032–1034
displaying, 1032
doubly, 1054
empty, 1031, 1032
head, 1024, 1045
initializing nodes, 1026–1027
inserting nodes into sorted list, 1035–1038
nodes, 1023, 1024
operations, 1029–1041
queues, 1092
recursive list functions, 1046–1048
recursive member functions, 1048–1049
recursive operations, 1045–1053
removing nodes from, 1038–1041
singly, 1054
in sorted order, 1034–1035
stacks, 1079–1082
structure, 1024
tail, 1046
template, 1041–1045
traversing, 1027–1029
variations, 1054
LinkedList class, 1041–1045
linkers, 10
linking, 10, 11, 20
Linux, 337
list STL container
defined, 1003
member functions, 1055
operations, 1056
ListNode structure, 1025, 1030
literals, *See* string literals
Little Lotto case study, 368–373
load function, 885
local scope, 198
local variables, 337–339. *See also* variables
defined, 337
initializing, 339
lifetime, 339
with same name, 343–344
shadowing global variable, 343
static, 344–346
log10 function, 126
log function, 126
logarithmic time, 609
logical errors, 20
logical operators, 187–195
&& (AND), 187–189
! (NOT), 187, 191–193
|| (OR), 187, 189–191
associativity, 194
defined, 187
list of, 187
numeric range checking with, 194–195
parentheses and, 194
precedence, 194
long data type, 45
long double data type, 55
long integer constants, 47
looping control structures, 249
loops
body, 250
braces and, 252–253
breaking out of, 279–281
condition, 249–250
conditional, 263
continue statement, 281–284
control variables, 251
count-controlled, 263
counters, 256–258
defined, 249
do-while, 258–262
early termination, 279
exit, 251, 280
for, 263–269
header, 249, 263
infinite, 252
inner, 277, 279
introduction, 249–253
iteration, 251, 259
nested, 277–279
outer, 277, 279
post test, 259
pretest, 251, 265
programming style and, 253
for reading data, 273–276
running totals, 269–271
sentinels, 271–273
use decision, 276–277
user control, 257–258
while, 249–256
low-level languages, 8
lvalue, 59

M

Mac OS operating system, 31, 337
machine language, 7
main function, 307, 308
defined, 29
execution, 307
function calls and, 309
function prototypes and, 316
opening/closing braces, 66
main memory, 4
makeDeletion function, 1125
Managed C++, Appendix N
manipulators
dec, 841
endl, 33–34, 39, 841, 843
fixed, 111, 841
flush, 841, 843
hex, 841
left, 112–113, 841
noshowpoint, 842
noshowpos, 842
oct, 842
output, 106–114, 114
right, 112–113, 842
scientific, 842, 843
setfill, 842, 843
setprecision, 108–111, 842
setw, 105, 106, 107, 838, 842
showpoint, 111–112, 842
showpos, 842
mantissa, 54

`map` STL container, 1003
mathematical expressions, 81–89
  associativity, 84
  converting to programming statements, 84–85
  decrement operator (--) in, 247–248
  evaluation, 81–83, 89–90
  grouping with parentheses, 84
  increment operator (++) in, 247–248
  operator precedence, 82–83
  parentheses around, 82
  value display, 82
mathematical library functions, 126–130
`max_element` algorithm (STL), 1009, 1012
member access specification, 747
member functions. *See also specific member functions and specific classes*
  base class, 747, 757
  bit flag reporting, 847
  calling, 394
  class template, 994
  constant, 677, 680–681
  constructors, 402–407
  defined, 120, 389
  defining, 395–401
  destructors, 407–410
  double colon (::), 396
  get (getter), 395
  implementation, 395
  inline, 400–401
  mutator, 395
  name, 396
  naming conventions, 397–400
  operator overloading, 715–716
  output formatting with, 843–844
  overloaded, 711
  overridden, 756, 757
  private, 410–413
  protected, 747
  public, 392, 450
  reading/writing fields, 849–862
  recursive, 1048–1049
  set (setter), 395
  static, 685–688
  virtual, 941–946
member variables
  defined, 389
  initializing, 440
  instance of another class, 420
  key field, 584
  private, 392, 450
  public, 392
  safeguarding, 400
members
  accessing, 394–395
  comparing, 438
  inherited, 747
  instance, 678, 685
  pointers as, 659–660
  `private`, 392
  `protected`, 745–748
  `public`, 392
  selecting, 659–661
  static, 681–688
  structure, 436–438
memberwise assignment, 694–695, 699
memory
  array requirements, 486–487
  C-strings, 122–124
  dynamic allocation, 643–652
  flash, 5
  freeing, 645
  heap, 645
  leaks, 646, 657
  main, 4
  pointer, 700
  random-access (RAM), 4, 15, 38
  secondary storage, 4–5
  stack, 401
  union, 447
  variable location, 619–620
menu-driven programs, 181–183
  defined, 181
  functions in, 334–337
menus
  defined, 181
  `do-while` loops with, 260–262
  `switch` statement and, 215–217
`merge` member function, 1055
`message` function, 901–903
methods, 389
Microsoft Visual C++, 132, 429, Appendix M
milliseconds, 465
`min_element` algorithm (STL), 1009, 1012
`mkChange` recursive function, 923–924
`mod` operation, 130
mode flags, 872
modular programming, 305
modulus operator (%), 61, 62, 83, 1088
MS-DOS, 5, 6
multidimensional arrays, 534–536. *See also* arrays
multi-file programs, 424–430, Appendix J
multi-line comments, 64–65. *See also* comments
`multimap` STL container, 1003
multiple assignment, 101–102
multiple exceptions, 979–980
multiple inheritance, 744, Appendix K
multiplication expressions, 84
multiplication operator (*), 61, 83
`multiset` STL container, 1003, 1004
multitasking operating systems, 6
multiuser operating systems, 6
mutators, 395
`MyString` class, 811–817

N

`\n` (newline escape sequence), 34, 35
n log *n* time, 609
named constants. *See also* constants
  declaring, 97–98
  `#define` directive, 98–100
  defined, 97
  initialization, 97
`nameSlice` function, 801–802
namespaces, 29, Appendix E
National Commerce Bank case study, 559–561
negation operator (`-`), 61
negative offsets, 873
nested `if` statements, 183–187. *See also* `if` statements
  defined, 183
  flowchart, 183
nested loops, 277–279. *See also* loops
  `break` statement in, 281
  defined, 277
  inner loop, 277, 279
  outer loop, 277, 279
  use examples, 277
nested structures. *See also* structures
  attribute relatedness, 441
  defined, 440

member access, 441
use decision, 441
.NET, Appendix N
new operator
bad_alloc exception, 983
in memory allocation, 644, 645, 647, 649
nodes (binary tree). *See also* binary search trees and binary trees
child, 1111
defined, 1111
deleting, 1122, 1123
empty collection, 1111
leaf, 1111
left child, 1111
left subtree, 1124
linking with parent, 1123
parent, 1111
processing, 1119
right child, 1111
right subtree, 1124
root, 1114, 1118, 1119
search tree, 1114
value storage in, 1112
nodes (linked list). *See also* linked lists
data type, 1024–1025
declaration, 1026–1027
defined, 1023
deleting, 1030, 1032
dynamic allocation, 1023
illustrated, 1024
initializing, 1026–1027
inserting into sorted lists, 1035–1038
processing, 1027–1028
successor pointer, 1024
noshowpoint manipulator, 842
noshowpos manipulator, 842
not equal to operator (!=), 158, 159, 815–817
NOT operator (!), 191–192
Boolean variables and, 192–193
defined, 187, 191
precedence, 194
truth table, 191
nouns. *See also* finding classes
identifying, 459
list, refining, 459
meaning, 459–460
plural versus singular, 461
representation, 460–461
null characters
defined, 50, 121, 771
storing in character variable, 771
null pointers, 631
null statements, 166
null terminator
defined, 50, 121
loops and, 802–803
placement, 51
space replacement with, 801
string object termination, 512
NumberArray class, 700–702, 704, 706
NumberList class, 1030, 1031, 1032, 1034
numbers
binary, 7
conversion to strings, 787–793
Fibonacci, 911
floating-point, 16, 44, 54
negative, 94
pseudorandom, 128
random, 128–130
real, 16
stored as strings, 39–40
numeric arrays, 505–508. *See also* arrays
numeric data, 15–16
numeric data types, 43–47
numeric ranges, checking, 194–195

O

object code, 10
object composition, 420–424
object files, 10
object serialization
defined, 883
member functions for, 884
scheme determination, 885–888
object-based programming, 463
object-oriented analysis and design, 455–463
attribute definition, 456
behavior definition, 457
class relationship definition, 457–458
class/object identification, 456, 458–461
defined, 455
process steps, 456
object-oriented exception handling, 977–978
object-oriented programming (OOP)
basis, 389
case study, 449–455
data hiding, 390
encapsulation, 389
in general-purpose object design, 431
methods, 389
objects, 389
procedural programming versus, 389
tutorial, Appendix C
objects
array of, 549–555
attributes, 389
cin, 75–81
cout, 31–35
creating, 393
defined, 389, 393
derived class, 743
deserializing, 884
dynamic allocation of, 653–655
file stream, 131–132, 830–831
general purpose, 431
identifying, 456
input validation, 431–435
input/output in, 430
instantiation, 393
introduction to, 393–395
lifetime, 737–739
member functions, 389, 390
calling, 394, 653
member variables, 389, 390
members, accessing, 394–395
passing, to functions, 413–417
pointers to, 655–657
returning, from functions, 417–419
reusability, 463
selecting members of, 659–661
serializing, 883, 884
stream, 32
string, 53, 204–206
subset, 953
vector, 538
oct manipulator, 842
octal constants, 47
off-by-one errors, 495
ofstream class. *See also* file stream objects
data type, 131
file open mode, 834
put member function, 857
seekp member function, 872
tellp member function, 876–878
write member function, 864
one-dimensional arrays, 526. *See also* arrays

Online Friendship Connections case study, 883–888
OOP. *See* object-oriented programming
`open` function, 132, 222, 831
operands
defined, 59
lvalue, 59
rvalue, 59
operating systems. *See also* software
defined, 5
functions, 464
multitasking, 6
multiuser, 6
single tasking, 5
single user, 6
`operator=` function, 705, 706, 711
`operator[]` function, 723, 724
operator functions
calling, 706
defined, 705
implementation, 706
parameters to, 706
operator overloading, 704–728
approaches to, 711–712
arithmetic operators, 712–715
assignment operator, 704–710
general issues, 710–711
member-function, 715–716
relational operators, 712–715
stand-alone, 715–716
operators
addition, 61, 83, 712, 716
address, 620, 628
arithmetic, 61–63, 83–84, 712–715
assignment, 14, 39, 168, 694, 704–710
associativity, 84, 1145
binary, 61, 927
combined assignment, 102–104
conditional, 207–209
decrement, 243–249
defined, 13
`delete`, 645, 646–648, 649, 657
division, 61, 62, 83, 710
dot, 394, 436, 438
in function templates, 990–991
increment, 243–249
indirection, 14, 621–622, 637
logical, 187–195
modulus, 61, 62, 83
multiplication, 61, 83
negation, 61
`new`, 644, 645, 647, 649
overloaded, 704–727, 815–817
postfix, 717
precedence, 82–83, 161, 194, 1145
prefix, 716–717
relational, 157–162, 202–206
scope resolution, 396
`sizeof`, 45, 58, 865, 867
stream extraction, 76, 117–119, 134–136, 275, 717–721, 837
stream insertion, 32, 133, 717–721, 832, 863
`string` class, 120, 807
structure pointer, 653, 659–660
subtraction, 61, 83
ternary, 61
type conversion, 728–730
unary, 61
optical devices, 5
OR operator (`||`), 189–191
defined, 187, 189
precedence, 194
short circuit evaluation, 190
truth table, 190
`ostream` class, 864
`ostringstream` class, 788, 840
outer loops, 277, 279
`OutOfRange` class, 981
output
buffers, 843
in class object, 430
console, 31
defined, 16–17
devices, 5
display, 32
file, 131, 831
floating-point, 111
iterator, 1004
left-justified, 113
manipulators, 106–114
opening files for, 878–882
right-justified, 106
validating, 223–225
output formatting, 105–114, 838–844
defined, 838
with member functions, 843–844
with `sstream` objects, 839–843
`outputFile` object, 132
overflow, 94–95
defined, 94
exceptions, 1074, 1075, 1092
stack, 1079
overloaded functions
calling, 360
defined, 359
return values, 361
uses, 360–361
overloaded operators. *See also* operators
++ (postfix) operator, 717
++ (prefix) operator, 716–717
+= operator, 710, 814
+ operator, 712, 716
<< operator, 717–721
< operator, 712, 713, 815–817
== operator, 815–817
= operator, 704–710, 814
!= operator, 815–817
>= operator, 815–817
>> operator, 717–721
> operator, 815
`[]` operator, 721–727
behavior, 710
as member function, 711
types of, 711
overloading
constructors, 405–406
defined, 757
with function templates, 992–993
overriding versus, 756–757
overridden functions
base class versus derived class versions, 756
gaining access to, 757
overriding
base class functions, 755–757
default arguments, 348
overloading versus, 756–757
ownership, class relationship, 458

P

padding, 106
parallel arrays, 513–516. *See also* arrays
defined, 513
subscript, 514
parameter list, 306, 317
parameters
actual, 317
`const`, 703
constant reference, 416–417

copy constructor, 702
data type, 320
default arguments, 348
default constructor, 407
defined, 314, 316, 321
formal, 317
initializing local variables with, 339
multiple, 321
number of, 314
to operator functions, 706
order of, 321
pointers as, 635–639
pointers to class objects as, 655–657
reference, 706
reference variables as, 350–358
scope, 321
storage location, 323
type, 985–986
types of, 314
parent node, 1111
parentheses `()`, 30
arguments in, 318
conditional expressions and, 160, 209
data passed into functions, 309
grouping with, 84
`if` statements and, 164, 165
logical operators and, 194
loops and, 249, 259, 263
pointers and, 626
relational expressions and, 160
parsing, 863
partial array initialization, 500–501
partially-filled arrays, 508
`partition` function, 916–917
Pascal, 9
passing
file stream objects to functions, 845–846
files to functions, 356–357
objects to functions, 413–417
structures to functions, 444–446
two-dimensional arrays to functions, 530–532
passing by reference
arguments, 351
defined, 635
files, 845–846
objects, 416–417
pointers versus, 635
structures, 444
when to use, 354–356
passing by value, 321–324
copy constructors and, 702
defined, 321
when to use, 354–356
`peek` member function, 855–857
pivot, 915
plain text, 957
pointer constants, 629
pointers
addresses, 628, 632–633
aggregation through, 736–737
arithmetic, 629–630
array addresses as arguments, 637
array relationship, 625–629
asterisk (*), 621, 637
base class, 936, 937, 939–940
to `char`, 772, 775–776
to class objects, 655–657
comparing, 632–634
constant, 641–643
to constants, 639–640
dangling, 646
declaration, 621–622
decrementing, 629, 630
defined, 619, 621
dereferencing, 623
dereferencing to structures, 660
derived class, 936, 937
as function parameters, 635–639
identification, 621
implicit, 870
incrementing, 629, 630
initializing, 630–631
as members, 659–660
memory allocation, 700
notation, 626, 627, 637
null, 631
parentheses and, 626
passing by reference versus, 635
passing C-string arguments with, 803–805
returning from functions, 648–650
to structures, 652–655
in structures, 870
successor, 1024
`this`, 677–680
variables, 621–624
pointer-to-a-pointer, 662
polymorphism
behavior, 943
defined, 941
dynamic binding, 946
static binding, 945–946
`pop` member function
`dynIntStack` class, 1079
`IntStack` class, 1074, 1077, 1081
`queue` container, 1097
`stack` container, 1083
pop operation, 1072, 1073
`pop_back` function
`list` container, 1055
`vector` class, 544, 545, 548, 1007
`pop_front` member function
`deque` container, 1096
`list` container, 1055
positive offsets, 873
post test loops, 259
postfix (++) operator, 717
postfix expressions, 927, 1103
postfix mode
defined, 244
prefix mode versus, 245–247
postorder traversal, 1119, 1120
pound sign (#), 10, 28, 30
`pow` function, 85, 128, 169, 316, 326
precedence
arithmetic operators, 82–83
logical operators, 194
relational operators, 160
table, 1145
precision, 108, 843
`precision` member function, 843, 844
prefix (++) operator, 716–717
prefix expressions, 927
prefix mode
defined, 244
postfix mode versus, 245–247
prefix notation, 93
preorder traversal, 1119, 1120
preprocessor, 10, 11
preprocessor directives, 10, 28, 36
prestandard C++, 37
prestandard compilers, 109
prestandard type cast expressions, 93
pretest loops, 251, 265
priming read, 254
`print` member function, 739
printable characters, 853
printing queues, 1085

`private` access specification, 392, 745, 748
private member functions, 410–413
  defined, 410
  example, 410–413
private member variables, 392
`private` members, 392
problem domain, 458–459
procedural programming, 389
processing, 16
program, 6–8
programmer-defined data types, 537
programmer-defined identifiers, 13, 14
programming
  as art and science, 2
  modular, 305
  object-based, 463
  object-oriented, 389–391
  procedural, 389
  process, 17–21
programming languages, 6–9
  BASIC, 9
  C#, 9
  C++, 9, 29, 37
  C, 9
  COBOL, 9
  elements, 12–14
  FORTRAN, 9
  high-level, 8
  Java, 9
  JavaScript, 9
  low-level, 8
  Pascal, 9
  Python, 9
  Ruby, 9
  Visual Basic, 9
programming style
  consistency, 66
  defined, 65
  `if` statement, 166
  `if/else` statement, 173
  multi-line statements, 67
  visual arrangement, 66
  `while` loop and, 253
programs
  defined, 6
  designing and creating, 17–21
  elements, 2, 12–16
  file location, 831–832
  function definition, 18
  hand tracing, 138–140
  input, 16
  language elements, 12–14
  lines and statements, 14–15
  linking, 20
  menu-driven, 181–183, 334–337
  model creation, 18–19
  multi-file, 424–430, Appendix J
  older style, 37
  output, 16–17
  parts of, 27–31
  processing, 16
  results validation, 21, 223–225
  running with test data, 20, 284–286
  steps for writing, 17
  three-step process, 16–17
  top-down design, 18, 21
  variable definitions, 15–16
  variables, 15
  visualization, 18
promoted values, 89–90
prompts, 76
`protected` access specification, 748
protected members, 745–748
prototypes. *See* function prototypes
pseudocode, 19
pseudorandom numbers, 128
`public` access specification, 392, 742, 745, 748
public member functions, 392
public member variables, 392
punctuation, 13, 14
pure virtual functions, 946–949
`push` member function
  `IntStack` class, 1074, 1075, 1077, 1081
  `queue` container, 1097
  `stack` container, 1083
push operation, 1072
`push_back` member function
  `deque` container, 1096
  `list` container, 1055
  `vector` class, 541–542, 548, 1007
`push_front` member function, 1055
`put` member function, 857–858
Pythagorean theorem, 127
Python, 9

Q

`qSort` function, 1099
quadratic time, 609
question mark (`?`), 488
`queue` container, 1097–1098
queues
  applications of, 1085
  based on circular arrays, 1088
  defined, 1084–1085
  dequeue, 1085, 1086
  dynamic, 1085, 1092–1095
  empty, 1086, 1088
  enqueue, 1085
  expanding/shrinking, 1092
  FIFO order, 1084–1085
  full, 1088
  introduction to, 1084–1092
  linked list implementation, 1092
  operations, 1085–1088
  printing, 1085
  static, 1085, 1092
  testing, 1088
QuickSort algorithm
  defined, 915
  functioning of, 915
  implementation, 916
  partitioning, 915
  pivot, 915
  recursive calls to, 1099
quotation marks (`""`), 30, 39, 50

R

`\r` (return escape sequence), 35
RAM (random-access memory), 4, 15, 38
`rand` function, 128
random access
  container, 1003
  files, 871–878
  iterator, 1004
  modes, 872
random numbers, 128–130
random-access memory (RAM), 4, 15, 38
`random_shuffle` algorithm (STL), 1009, 1011
ranking, data types, 89
read
  position, 135
  priming, 254
`read` member function, 865
reading
  data with loops, 273–276
  files, 134–137, 832–835
  files, member functions, 849–862
  line of input, 124–125
  multiple values, 79–80
  single characters, 117
real numbers, 16
records
  creating with structures, 866–871

fields, 866
information organization in, 867
recursion, 901–933
depth of, 902
direct, 907
eliminating, 1098–1102
indirect, 907
introduction to, 901–908
iteration versus, 926
overhead, 926, 1098
uses, 905
recursive algorithms
binary search, 913–914
enumeration, 922–926
exhaustive, 922–926
factorial, 908–910
gcd, 910–911
linked list operations, 1045–1053
QuickSort, 915–918
Towers of Hanoi, 919–922
recursive functions. *See also* recursive algorithms
base case, 904, 913
defined, 901
number of recursive call and, 902
recursively defined problems, solving, 911–912
reference, passing arguments by, 351
reference parameters, 706
reference variables
ampersand (&), 351
changes to, 521
defined, 350, 351
definition, 351
as parameters, 350–358
pointers and, 635
use instances, 355
`reinterpret_cast`, 864
relational expressions
array elements in, 503
as Boolean expressions, 158
decrement operator (--) in, 248
defined, 158
example, 159
`if` statements and, 168–169
increment operator (++) in, 248
in parentheses, 160
statements including, 161
relational operators, 157–162
as binary operators, 158
in comparing characters, 202–204
in comparing enumerated variables, 220
in comparing pointers, 632–634
in comparing strings, 204–205
defined, 157
function templates and, 990
list of, 158
overloading, 712–715
precedence, 160, 161
`string` class, 807
relationships
access, 458
class, 457–458
composition, 457
has-a, 734, 739
inheritance, 458
is-a, 740–744
ownership, 458
value of, 158–159
Reliable Software Systems, Inc. case study, 1056–1060
remainder, 62
`remove` member function, 1030, 1038–1040
recursive, 1051–1053
sorted/unsorted lists, 1039
`replace` function, 810
reserved words. *See* key words
`resize` member function
`string` class, 810
`vector` class, 548, 1007
responsibilities, class, 461–463
rethrowing exceptions, 984–985
`return` statements, 325–326
defined, 325
format, 327
value-returning functions, 327
return types, 307
`char *`, 648
structure variable, 446
`void`, 307, 709
return values
assignment operator (=), 709–710
Boolean, 332–334
data types, 329, 331
functions, 326–331
multiple, 326, 327, 447
object as, 417–419
structure as, 446–447
using, 329
reusability, object, 463
`reverse` member function, 548, 1007, 1055
rewinding files, 858–860
right child node, 1111
`right` manipulator, 112–113, 842
right to left associativity, 84
right-justified output, 106
root, binary tree, 1111
root nodes, binary search tree, 1114, 1118, 1119
`round` function, 126
round-off errors, 291
rows, two-dimensional array, 532–533
Ruby, 9
running totals
accumulator, 269–270
defined, 269
example use, 271
logic, 269
run-time errors, 20
run-time libraries, 10
run-time type identification, Appendix I
rvalue, 59

S

`scientific` manipulator, 842, 843
scientific notation, 54, 843
scope
block, 198
coming into, 199
defined, 60, 198
of global variables, 339
leaving, 199
local, 198
of parameters, 321
resolution operator, 396, 757
variable, 60–61, 197–199
scope resolution operator (`::`), 396, 757
screens
clearing, 337
control, 464–469
input form, controlling, 467–469
positioning cursor on, 464–467
search algorithms. *See also* algorithms
binary search, 580–584
defined, 577
introduction to, 577–584
linear search, 577–580

`SearchableVector` class, 1000, 1001
searching
  arrays of objects, 584–587
  binary search trees, 1114, 1121–1122
  vectors, 599–601
`searchList` function, 578
secondary storage, 4–5
Secure Encryption Systems, Inc. case study, 957–960
`seekg` member function, 859, 872–876
`seekp` member function, 872–876
selection sort, 591–595. *See also* sorting algorithms
  worst case complexity, 607
`selectSort` function, 663, 664–665
self-referential data types, 1025
semicolons (;), 30, 36, 100
  `if` statements and, 166–167
  infinite loops and, 252
sentinels, 271–273
sequence structure, 162–163
sequential containers, 538, 1002–1003
  STL, 1003
sequential file access, 871
sequential search. *See* linear search
set (setter) functions, 395
Set Intersection case study, 561
`set` STL container, 1003, 1004
`SetConsoleCursorPosition` function, 464
`SetConsoleTextAttribute` function, 464
`setf` member function, 843, 844
`setfill` manipulator, 842, 843
`setLength` function, 397, 427, 712
`setName` function, 745
`setprecision` manipulator
  defined, 108, 842
  `iomanip` header file and, 110
  output, 109
  value display and, 109
`setRadius` function, 391, 395
`setValue` function, 679, 680
`setw` manipulator, 105, 106, 107, 838, 842
`setWidth` function, 397, 427
short circuit evaluation, 188, 190
`short` data type, 45
`show` member function, 663, 665
`showArea` function, 348
`showArray` function, 530, 532
`showFees` function, 336, 366, 367
`showLocal` function, 345
`showpoint` manipulator, 111–112, 842
`showpos` manipulator, 842
`showRec` function, 876
`showSorted` function, 663, 665–666
`showState` function, 847–848
`showStatic` function, 346
`showSum` function, 320–321
`showValues` function, 414, 417, 517–518, 520
significant digits, 108
`sin` function, 126
single precision data type, 54
single tasking operating systems, 5
single user operating systems, 6
single-line comments, 64
singly linked lists, 1054
size, data type, 44, 45, 58–59
size declarators. *See also* arrays
  compiler requirement, 490
  defined, 120, 486
  example, 487
  subscripts versus, 488
  two-dimensional arrays, 527
  vector, 538
`size` member function
  `list` container, 1055
  `stack` container, 1083
  `string` class, 810
  `vector` class, 543, 548, 601, 1007
`sizeof` operator, 45, 58, 865, 867
soft copy, 5
software
  abstraction and, 388
  application, 5
  defined, 5
  engineering, 21
  operating system, 5, 6
`sort` algorithm (STL), 1009, 1011
sorting
  ranges, 1098
  strings, 782–783
  sublists, 915–916
sorting algorithms. *See also* algorithms
  array of objects, 596–599
  bubble sort, 587–591
  defined, 587
  introduction to, 587–595
  QuickSort, 915–918
  selection sort, 591–595
  for vectors, 599–601
source code
  compiling, 20
  defined, 10
  writing, 20
source files
  defined, 10
  translation process, 11
spaces
  blank, 106
  file names and, 133
spacing, 167
special characters, 30
specialization, 740
specification file, class, 424, 425, 430
spreadsheets, 829
`sqrt` function, 126, 127, 388
`square` function, 329, 331, 359–360, 986
`srand` function, 128
`sstream` objects, 839–843
`stack` container (STL), 1083
stack, memory, 401
stacks
  applications of, 1072
  becoming empty, 1078
  `Constructor` function, 1074
  dynamic, 1072, 1079–1082
  exception handling, 1078–1079
  filling up, 1078
  `isEmpty` function, 1074, 1078
  isEmpty operation, 1073
  `isFull` function, 1079
  as LIFO structures, 1071
  linked list, 1079–1082
  operations, 1072–1073
  overflow, 1079
  `pop` function, 1074, 1077, 1081
  pop operation, 1072, 1073
  `push` function, 1074, 1075, 1077, 1081
  push operation, 1072
  static, 1072
  templates, 1079
  underflow, 1079
  unwinding, 983–984
stale data, avoiding, 400
stand-alone operator overloading, 715–716

standard C++, 37
Standard Template Library (STL)
algorithms, 1008–1009
containers, 538, 1002–1004, 1054–1056, 1082–1084
defined, 537, 1002
introduction to, 1002–1015
iterators, 1002, 1004–1005
statements
assignment, 38–39, 59–60, 81
`break`, 211–212
`case`, 210–211, 212
`continue`, 281–284
converting algebraic expressions to, 84–85
defined, 14–15
definition, 393
function call, 307, 309
`if`, 162–172
`if/else`, 172–175
`if/else if`, 175–181
multi-line, 67
multiple, 267–268
null, 166
`return`, 325–326
`switch`, 210–218
`typedef`, 516–517
variable definition, 16
`while` loop, 250
static binding, 945–946
`static` key word, 685
static local variables, 344–347
static members. *See also* members
declarations, 685
defined, 681
defining, 685
functions, 685–688
variables, 682–685
static queues. *See also* queues
defined, 1085
exceptions, 1092
static stacks. *See also* stacks
defined, 1072
isEmpty operation, 1073
`static_cast` type cast expression, 92, 93
`std` namespace, 29
STL. *See* Standard Template Library
storage. *See* memory
`store` function, 885
`strcat` library function, 776–777, 783
`strcmp` library function, 779–782
case sensitivity, 780–781
defined, 779, 783
example use, 780
NOT (!) operator with, 782
prototype, 779
return value, 780, 782, 792
summary, 783
`strcpy` library function, 122, 778, 783
stream extraction operator (>>)
`cin` object, 76–79
defined, 76
in end of file test, 837
file stream objects and, 832
overloading, 717–721
in reading from files, 134–137
returning value, 275
stream insertion operator (<<)
`cout` object , 32
defined, 32
file stream objects and, 832
in number formatting, 863
overloading, 717–721
in writing to files, 133–134
stream manipulators. *See* manipulators
stream objects
defined, 32
writing to, 839
`string` class, 52–53, 805–810
advantages, 805
`append` function, 809
`assign` function, 809
`at` function, 809
`begin` function, 809
`capacity` function, 809
`clear` function, 809
`compare` function, 809
constructors, 806
`copy` function, 809
creating, 810–817
`c_str` function, 809
`data` function, 809
defined, 52
`empty` function, 809
`end` function, 809
`erase` function, 810
`find` function, 810
header file, 52, 806
`insert` function, 810
`length` function, 512, 810
member functions, 805–806, 809–810
operators, 807
`replace` function, 810
`resize` function, 810
`size` function, 810
`substr` function, 810
`swap` function, 810
using, 52–53
string constants, *See* string literals
`string` header file, 52
string literals, 29, 40, 49–52, 772–773
addresses, 773
compilers and, 772, 779
holding C-strings, 773
numbers represented as, 39–40
`stringCopy` function, 801
strings. *See also* `string` class
comparing, 204–206
converting to numbers, 787–793
defined, 29
field width, 108
inputting, 115–117
numbers stored as, 39–40
objects, 53
operators, 120
processing, 511–513
sorting, 782–783
storage, 512
`strlen` library function, 776, 783
`strncpy` library function, 783
`strstr` library function, 783–786
structure pointer operator (->), 659–660
defined, 653
in node processing, 1027
structure variables
comparing, 438
displaying, 438
names, 436
passing to functions, 444–446
returning from functions, 446–447
structures
arrays of, 555–559
declaring, 435–436
defined, 435
dereferencing pointers to, 660
initializing, 438–440
member access, 436–438
members, 440
nested, 440–442
passing to functions, 444–446

structures (*continued*)
pointers in, 870
pointers to, 653–655
record creation with, 866–871
returning, from functions, 446–447
stubs, 366–367
subclasses, 744
sublists, 915–916
subscripts. *See also* arrays
accessing array elements with, 487
defined, 487
out of bounds, 493
parallel arrays, 514
size declarators versus, 488
starting number, 487
`substr` function, 810
subtraction operator (`-`), 61, 83, 713
successor pointers, 1024
`sum` function, 327, 328
`sumArray` function, 523
superclasses, 744
`swap`, 917, 989
`list` container, 1055
`string` class, 810
`vector` class, 548, 1007
`switch` statements, 210–218
`break` statements, 211–212
`case` statements, 210–211, 212
defined, 210
fall-through capability, 213, 214
format, 210
in menu-driven systems, 215–217
syntax
defined, 13
errors, 10, 65

T

`\t` (horizontal tab escape sequence), 35
tail, linked lists, 1046
`tan` function, 126
`tellg` member function, 876–878
`tellp` member function, 876–878
templates. *See also* Standard Template Library (STL)
binary search tree, 1131
class, 993–1002
function, 985–993
linked list, 1041–1045
stack, 1079
type parameters, 985–986, 987, 990, 992, 994
ternary operators, 61
test data, 284–286
test expressions, 263
testing
alphabetic characters, 193
Boolean variables, 192–193
character, 793–797
for end of file, 836–837
error, 973
for file open errors, 222
for file operation errors, 835–836
with good test data, 284
queues, 1088
`testNum` function, 796–797
text
cipher, 957
editors, 10
files, 863
plain, 957
`this` pointer, 677–680
defined, 677
use example, 680
three-dimensional arrays, 534–536. *See also* arrays
throw point, 974
`throw` statement, 981
throwing exceptions. *See also* exceptions
defined, 974
rethrowing, 984–985
throw point, 974
tilde character (~), 407
`time` function, 129
time sharing, 6
`tolower` library function, 797–798
`top` member function, 1083
top-down design, 18, 21
totals, running, 269–271
`toupper` library function, 431–432, 433, 797–798
Towers of Hanoi recursion example, 919–922
trailing `else`, 179–181
`transform` virtual function, 958
traversing
binary search trees, 1119–1121
lists, 1027–1029
`true`, 159, 274, 332–333
truncation, 56, 90
try/catch construct, 975–976
two-dimensional arrays. *See also* arrays
braces and, 530
column information, 532
defined, 526
defining, 527
element access, 532
information processing, 527
initialization, 529
initialization list, 530
passing to functions, 530–532
processing, 527
size declarators, 527
summing all elements, 532
summing columns, 533
summing rows, 532–533
`typedef` statement with, 532
type cast expressions, 90–93, Appendix I
assignment, 936
with base class pointers, 939–940
C-style, 93
defined, 90
example, 90–91
format, 90
prestandard, 93
`reinterpret_cast`, 864
`static_cast`, 92, 93
type coercion, 89
type compatibility, 936–939
type conversion
explicit, 90–94
implicit, 89–90
type parameters. *See also* class templates; function templates
declaration, use of, 992
defined, 985–986
definition of, 987
list of, 994
multiple types, 991
`typedef` statement
with arrays, 516–517
defined, 516
with 2D arrays, 532
uses, 517

U

UML (Unified Modeling Language), 391, Appendix D
unary operators, 61
underflow, 94–95
defined, 94
exceptions, 1074, 1092
stack, 1079
Unified Modeling Language (UML), 391, Appendix D
unions, 447–449

unique member function, 1055
United Cause Relief Agency case study, 661–669
UNIX, 6
unsetf member function, 844
unsigned int data type, 45
unsigned long data type, 45
unsigned short data type, 44, 45
unwinding the stack, 983–984
update expressions
  defined, 263
  forms, 266
  multiple statements in, 267–268
  omitting, 268
USB flash drives, 5
user-controlled for loops, 267
uses-a, 457
using namespace std; statement, 37

V

validation
  input, 183, 196–197
  output results, 223–225
value, passing arguments by, 321–323
value-returning functions, 326–331
  calling, 328–331
  defined, 326
  defining, 327–328
  prototype, 328
  return statement, 327
  return values, 329
variable definitions
  defined, 15–16, 38
  definition location, 38, 197–201
  in for loop initialization expression, 266
  requirement, 38
variables
  accumulator, 269
  address, 620
  assignments, 59–60
  Boolean, 192–193
  concept, 37–38
  constant, 97–101
  control, 511
  counter, 256–258, 264–265, 266
  data types, 43–48
  defined, 15, 37
  enumerated, 220
  environment, 819
  flag, 169
  floating-point, 44, 54–55
  global, 337, 339–340, 343–344
  index, 581
  initialization, 59–60
  integer, 38, 43–47, 56–57
  local, 337–339, 343–347
  loop control, 251
  member, 389
  memory location, 619–620
  names, 41–42, 43
  nested block definitions, 200
  parameter, 317, 318
  pointer, 621–624
  reference, 350–358
  with same name, 199–200
  scope, 60–61, 197–199
  static member, 682–685
  structure, 436, 438
  union, 448
vector data type, 537
  header file, 538
  member functions, 548
vector header file, 538
vector member functions
  at, 548, 809, 1006
  back, 1006
  begin, 1006
  capacity, 548, 1006
  clear, 545, 548, 1006
  empty, 548, 1006
  end, 1004, 1006
  erase, 1006
  front, 1006
  insert, 1007
  pop_back, 544, 545, 548, 1007
  push_back, 541–542, 548, 1007
  resize, 548, 1007
  reverse, 548, 1007
  size, 543, 548, 1007
  summary table, 548
  swap, 548, 1007
vector STL container, 1003, 1005–1007
vectors, 537–549
  advantages, 538
  clearing, 545–546
  defined, 538
  defining, 538–539
  elements, 541
  elements, number of, 543
  elements, removing, 544–545
  empty, detecting, 546–547
  example definitions, 539
  function summary table, 548
  initializing, 539
  linked lists versus, 1023–1024
  objects, 538
  searching, 599–601
  size declarator, 538
  size determination, 543–544
  sorting, 599–601
  starting size, 538
  values, storing and retrieving, 539–541
virtual functions
  declaration, 943
  defined, 941
  polymorphism and, 941–946
  pure, 946–949
  tagging, 943
  uses, 943
virtual inheritance, Appendix K
virtual key word, 943
Visual Basic, 9
Visual C++, 132, 429, Appendix M
void functions, 307
void return type, 709

W

while loops. *See also* loops
  body, 250
  braces and, 252–253
  break statement, 280
  condition, 249–250
  continue statement, 282
  control variable, 511
  defined, 249
  do-while loop versus, 259
  flowchart, 249
  format, 249
  header, 249
  for input validation, 253–256
  iteration, 251
  as pretest loop, 251
  programming style and, 253
  use decision, 276
whitespace characters, 115, 849
width function, 123, 843, 844
Windows operating system, 31
withdraw member function, 451
word processors, 829
words, 4
worst-case complexity function, 605–607
write member function, 864
writing
  files, 133–134, 832–835, 849–862
  programs, 17