

STARTING OUT WITH

C++

From Control Structures
through Objects

NINTH EDITION

STARTING OUT WITH

C++

From Control Structures through Objects

NINTH EDITION

Tony Gaddis

Haywood Community College



Pearson

330 Hudson Street, New York, NY 10013

Senior Vice President Courseware Portfolio Management: Marcia J. Horton
Director, Portfolio Management: Engineering, Computer Science & Global Editions: Julian Partridge
Portfolio Manager: Matt Goldstein
Portfolio Management Assistant: Kristy Alaura
Field Marketing Manager: Demetrius Hall

Product Marketing Manager: Yvonne Vannatta
Managing Producer, ECS and Math: Scott Disanno
Content Producer: Sandra L. Rodriguez
Composition: iEnergizer Aptara®, Ltd.
Cover Designer: Joyce Wells
Cover Photo: Samokhin/123RF

Credits and acknowledgments borrowed from other sources and reproduced, with permission, appear on the Credits page in the endmatter of this textbook.

Copyright © 2018, 2015, 2012, 2009 Pearson Education, Inc. Hoboken, NJ 07030. All rights reserved.
Manufactured in the United States of America. This publication is protected by copyright and permissions 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 otherwise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions department, please visit www.pearsoned.com/permissions/.

Many of the designations by manufacturers and seller 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.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages with, or arising out of, the furnishing, performance, or use of these programs.

Pearson Education Ltd., London
Pearson Education Singapore, Pte. Ltd
Pearson Education Canada, Inc.
Pearson Education Japan
Pearson Education Australia PTY, Ltd

Pearson Education North Asia, Ltd., Hong Kong
Pearson Education de Mexico, S.A. de C.V.
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., Hoboken

Library of Congress Cataloging-in-Publication Data

Names: Gaddis, Tony, author.
Title: Starting out with C++. From control structures through objects / Tony Gaddis, Haywood Community College.
Other titles: From control structures through objects | Starting out with C plus plus. From control structures through objects
Description: Ninth edition. | Boston : Pearson Education, Inc., [2017]
Identifiers: LCCN 2016056456 | ISBN 9780134498379 (alk. paper) | ISBN 0134498372 (alk. paper)
Subjects: LCSH: C++ (Computer program language)
Classification: LCC QA76.73.C153 G334 2017 | DDC 005.13/3—dc23 LC record available at <https://lccn.loc.gov/2016056456>

Contents at a Glance

Preface	xvii
CHAPTER 1	Introduction to Computers and Programming 1
CHAPTER 2	Introduction to C++ 27
CHAPTER 3	Expressions and Interactivity 85
CHAPTER 4	Making Decisions 151
CHAPTER 5	Loops and Files 231
CHAPTER 6	Functions 305
CHAPTER 7	Arrays and Vectors 381
CHAPTER 8	Searching and Sorting Arrays 463
CHAPTER 9	Pointers 503
CHAPTER 10	Characters, C-Strings, and More about the <code>string</code> Class 557
CHAPTER 11	Structured Data 613
CHAPTER 12	Advanced File Operations 665
CHAPTER 13	Introduction to Classes 719
CHAPTER 14	More about Classes 817
CHAPTER 15	Inheritance, Polymorphism, and Virtual Functions 907
CHAPTER 16	Exceptions and Templates 989
CHAPTER 17	The Standard Template Library 1029
CHAPTER 18	Linked Lists 1123
CHAPTER 19	Stacks and Queues 1165
CHAPTER 20	Recursion 1223
CHAPTER 21	Binary Trees 1257
Appendix A: The ASCII Character Set 1287	
Appendix B: Operator Precedence and Associativity 1289	

Quick References	1291
Index	1293
Credit	1311
Online	The following appendices are available at www.pearsonhighered.com/gaddis .
Appendix C: Introduction to Flowcharting	
Appendix D: Using UML in Class Design	
Appendix E: Namespaces	
Appendix F: Passing Command Line Arguments	
Appendix G: Binary Numbers and Bitwise Operations	
Appendix H: STL Algorithms	
Appendix I: Multi-Source File Programs	
Appendix J: Stream Member Functions for Formatting	
Appendix K: Unions	
Appendix L: Answers to Checkpoints	
Appendix M: Answers to Odd Numbered Review Questions	
Case Study 1: String Manipulation	
Case Study 2: High Adventure Travel Agency—Part 1	
Case Study 3: Loan Amortization	
Case Study 4: Creating a String Class	
Case Study 5: High Adventure Travel Agency—Part 2	
Case Study 6: High Adventure Travel Agency—Part 3	
Case Study 7: Intersection of Sets	
Case Study 8: Sales Commission	

Contents

Preface xvii

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 8
 - 1.4 What Is a Program Made of? 14
 - 1.5 Input, Processing, and Output 17
 - 1.6 The Programming Process 18
 - 1.7 Procedural and Object-Oriented Programming 22
- Review Questions and Exercises* 24

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 36
 - 2.4 Variables, Literals, and Assignment Statements 38
 - 2.5 Identifiers 42
 - 2.6 Integer Data Types 43
 - 2.7 The `char` Data Type 49
 - 2.8 The C++ `string` Class 53
 - 2.9 Floating-Point Data Types 55
 - 2.10 The `bool` Data Type 58
 - 2.11 Determining the Size of a Data Type 59
 - 2.12 More about Variable Assignments and Initialization 60
 - 2.13 Scope 62
 - 2.14 Arithmetic Operators 63
 - 2.15 Comments 71
 - 2.16 Named Constants 73
 - 2.17 Programming Style 75
- Review Questions and Exercises* 77
Programming Challenges 81

CHAPTER 3 Expressions and Interactivity 85

- 3.1 The `cin` Object 85
- 3.2 Mathematical Expressions 91
- 3.3 When You Mix Apples and Oranges: Type Conversion 100
- 3.4 Overflow and Underflow 102
- 3.5 Type Casting 103
- 3.6 Multiple Assignment and Combined Assignment 106
- 3.7 Formatting Output 110
- 3.8 Working with Characters and `string` Objects 120
- 3.9 More Mathematical Library Functions 126
- 3.10 Focus on Debugging: Hand Tracing a Program 132
- 3.11 Focus on Problem Solving: A Case Study 134
- Review Questions and Exercises* 138
- Programming Challenges* 144

CHAPTER 4 Making Decisions 151

- 4.1 Relational Operators 151
- 4.2 The `if` Statement 156
- 4.3 Expanding the `if` Statement 164
- 4.4 The `if/else` Statement 168
- 4.5 Nested `if` Statements 171
- 4.6 The `if/else if` Statement 178
- 4.7 Flags 183
- 4.8 Logical Operators 184
- 4.9 Checking Numeric Ranges with Logical Operators 191
- 4.10 Menus 192
- 4.11 Focus on Software Engineering: Validating User Input 195
- 4.12 Comparing Characters and Strings 197
- 4.13 The Conditional Operator 201
- 4.14 The `switch` Statement 204
- 4.15 More about Blocks and Variable Scope 213
- Review Questions and Exercises* 216
- Programming Challenges* 222

CHAPTER 5 Loops and Files 231

- 5.1 The Increment and Decrement Operators 231
- 5.2 Introduction to Loops: The `while` Loop 236
- 5.3 Using the `while` Loop for Input Validation 243
- 5.4 Counters 245
- 5.5 The `do-while` Loop 246
- 5.6 The `for` Loop 251
- 5.7 Keeping a Running Total 261
- 5.8 Sentinels 264
- 5.9 Focus on Software Engineering: Deciding Which Loop to Use 265
- 5.10 Nested Loops 266
- 5.11 Using Files for Data Storage 269
- 5.12 Optional Topics: Breaking and Continuing a Loop 288
- Review Questions and Exercises* 292
- Programming Challenges* 297

CHAPTER 6 Functions 305

- 6.1 Focus on Software Engineering: Modular Programming 305
- 6.2 Defining and Calling Functions 306
- 6.3 Function Prototypes 315
- 6.4 Sending Data into a Function 317
- 6.5 Passing Data by Value 322
- 6.6 Focus on Software Engineering: Using Functions in a Menu-Driven Program 324
- 6.7 The `return` Statement 328
- 6.8 Returning a Value from a Function 330
- 6.9 Returning a Boolean Value 338
- 6.10 Local and Global Variables 340
- 6.11 Static Local Variables 348
- 6.12 Default Arguments 351
- 6.13 Using Reference Variables as Parameters 354
- 6.14 Overloading Functions 360
- 6.15 The `exit()` Function 364
- 6.16 Stubs and Drivers 367
- Review Questions and Exercises* 369
- Programming Challenges* 372

CHAPTER 7 Arrays and Vectors 381

- 7.1 Arrays Hold Multiple Values 381
- 7.2 Accessing Array Elements 383
- 7.3 No Bounds Checking in C++ 395
- 7.4 The Range-Based for Loop 398
- 7.5 Processing Array Contents 402
- 7.6 Focus on Software Engineering: Using Parallel Arrays 410
- 7.7 Arrays as Function Arguments 413
- 7.8 Two-Dimensional Arrays 424
- 7.9 Arrays with Three or More Dimensions 431
- 7.10 Focus on Problem Solving and Program Design: A Case Study 433
- 7.11 Introduction to the STL `vector` 435
- Review Questions and Exercises* 449
- Programming Challenges* 454

CHAPTER 8 Searching and Sorting Arrays 463

- 8.1 Focus on Software Engineering: Introduction to Search Algorithms 463
- 8.2 Focus on Problem Solving and Program Design: A Case Study 469
- 8.3 Focus on Software Engineering: Introduction to Sorting Algorithms 476
- 8.4 Focus on Problem Solving and Program Design: A Case Study 486
- 8.5 Sorting and Searching `vectors` 495
- Review Questions and Exercises* 498
- Programming Challenges* 499

CHAPTER 9 Pointers 503

- 9.1 Getting the Address of a Variable 503
- 9.2 Pointer Variables 505
- 9.3 The Relationship between Arrays and Pointers 512

9.4	Pointer Arithmetic	516
9.5	Initializing Pointers	518
9.6	Comparing Pointers	519
9.7	Pointers as Function Parameters	521
9.8	Dynamic Memory Allocation	530
9.9	Returning Pointers from Functions	534
9.10	Using Smart Pointers to Avoid Memory Leaks	541
9.11	Focus on Problem Solving and Program Design: A Case Study	544
	<i>Review Questions and Exercises</i>	550
	<i>Programming Challenges</i>	553

CHAPTER 10 Characters, C-Strings, and More about the string Class 557

10.1	Character Testing	557
10.2	Character Case Conversion	561
10.3	C-Strings	564
10.4	Library Functions for Working with C-Strings	568
10.5	String/Numeric Conversion Functions	579
10.6	Focus on Software Engineering: Writing Your Own C-String-Handling Functions	585
10.7	More about the C++ <i>string</i> Class	591
10.8	Focus on Problem Solving and Program Design: A Case Study	603
	<i>Review Questions and Exercises</i>	604
	<i>Programming Challenges</i>	607

CHAPTER 11 Structured Data 613

11.1	Abstract Data Types	613
11.2	Structures	615
11.3	Accessing Structure Members	618
11.4	Initializing a Structure	622
11.5	Arrays of Structures	625
11.6	Focus on Software Engineering: Nested Structures	627
11.7	Structures as Function Arguments	631
11.8	Returning a Structure from a Function	634
11.9	Pointers to Structures	637
11.10	Focus on Software Engineering: When to Use., When to Use \rightarrow , and When to Use *	640
11.11	Enumerated Data Types	642
	<i>Review Questions and Exercises</i>	653
	<i>Programming Challenges</i>	659

CHAPTER 12 Advanced File Operations 665

12.1	File Operations	665
12.2	File Output Formatting	671
12.3	Passing File Stream Objects to Functions	673
12.4	More Detailed Error Testing	675
12.5	Member Functions for Reading and Writing Files	678
12.6	Focus on Software Engineering: Working with Multiple Files	686
12.7	Binary Files	688
12.8	Creating Records with Structures	693

- 12.9 Random-Access Files 697
- 12.10 Opening a File for Both Input and Output 705
- Review Questions and Exercises* 710
- Programming Challenges* 713

CHAPTER 13 Introduction to Classes 719

- 13.1 Procedural and Object-Oriented Programming 719
- 13.2 Introduction to Classes 726
- 13.3 Defining an Instance of a Class 731
- 13.4 Why Have Private Members? 744
- 13.5 Focus on Software Engineering: Separating Class Specification from Implementation 745
- 13.6 Inline Member Functions 751
- 13.7 Constructors 754
- 13.8 Passing Arguments to Constructors 759
- 13.9 Destructors 767
- 13.10 Overloading Constructors 771
- 13.11 Private Member Functions 775
- 13.12 Arrays of Objects 777
- 13.13 Focus on Problem Solving and Program Design: An OOP Case Study 781
- 13.14 Focus on Object-Oriented Programming: Simulating Dice with Objects 788
- 13.15 Focus on Object-Oriented Design: The Unified Modeling Language (UML) 792
- 13.16 Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities 794
- Review Questions and Exercises* 803
- Programming Challenges* 808

CHAPTER 14 More about Classes 817

- 14.1 Instance and Static Members 817
- 14.2 Friends of Classes 825
- 14.3 Memberwise Assignment 830
- 14.4 Copy Constructors 831
- 14.5 Operator Overloading 837
- 14.6 Object Conversion 864
- 14.7 Aggregation 866
- 14.8 Focus on Object-Oriented Design: Class Collaborations 871
- 14.9 Focus on Object-Oriented Programming: Simulating the Game of Cho-Han 876
- 14.10 Rvalue References and Move Semantics 886
- Review Questions and Exercises* 895
- Programming Challenges* 900

CHAPTER 15 Inheritance, Polymorphism, and Virtual Functions 907

- 15.1 What Is Inheritance? 907
- 15.2 Protected Members and Class Access 916
- 15.3 Constructors and Destructors in Base and Derived Classes 922
- 15.4 Redefining Base Class Functions 936
- 15.5 Class Hierarchies 941
- 15.6 Polymorphism and Virtual Member Functions 947
- 15.7 Abstract Base Classes and Pure Virtual Functions 963

- 15.8 Multiple Inheritance 970
- Review Questions and Exercises* 977
- Programming Challenges* 981

CHAPTER 16 Exceptions and Templates 989

- 16.1 Exceptions 989
- 16.2 Function Templates 1008
- 16.3 Focus on Software Engineering: Where to Start When Defining Templates 1014
- 16.4 Class Templates 1014
- Review Questions and Exercises* 1024
- Programming Challenges* 1026

CHAPTER 17 The Standard Template Library 1029

- 17.1 Introduction to the Standard Template Library 1029
- 17.2 STL Container and Iterator Fundamentals 1029
- 17.3 The `vector` Class 1040
- 17.4 The `map`, `multimap`, and `unordered_map` Classes 1054
- 17.5 The `set`, `multiset`, and `unordered_set` Classes 1079
- 17.6 Algorithms 1086
- 17.7 Introduction to Function Objects and Lambda Expressions 1107
- Review Questions and Exercises* 1114
- Programming Challenges* 1118

CHAPTER 18 Linked Lists 1123

- 18.1 Introduction to the Linked List ADT 1123
- 18.2 Linked List Operations 1125
- 18.3 A Linked List Template 1141
- 18.4 Variations of the Linked List 1153
- 18.5 The STL `list` and `forward_list` Containers 1154
- Review Questions and Exercises* 1158
- Programming Challenges* 1161

CHAPTER 19 Stacks and Queues 1165

- 19.1 Introduction to the Stack ADT 1165
- 19.2 Dynamic Stacks 1182
- 19.3 The STL `stack` Container 1193
- 19.4 Introduction to the Queue ADT 1195
- 19.5 Dynamic Queues 1207
- 19.6 The STL `deque` and `queue` Containers 1214
- Review Questions and Exercises* 1217
- Programming Challenges* 1219

CHAPTER 20 Recursion 1223

- 20.1 Introduction to Recursion 1223
- 20.2 Solving Problems with Recursion 1227
- 20.3 Focus on Problem Solving and Program Design: The Recursive `gcd` Function 1235
- 20.4 Focus on Problem Solving and Program Design: Solving Recursively Defined Problems 1236

- 20.5 Focus on Problem Solving and Program Design: Recursive Linked List Operations 1237
- 20.6 Focus on Problem Solving and Program Design: A Recursive Binary Search Function 1241
- 20.7 The Towers of Hanoi 1243
- 20.8 Focus on Problem Solving and Program Design: The QuickSort Algorithm 1246
- 20.9 Exhaustive Algorithms 1250
- 20.10 Focus on Software Engineering: Recursion versus Iteration 1253
Review Questions and Exercises 1253
Programming Challenges 1255

CHAPTER 21 Binary Trees 1257

- 21.1 Definition and Applications of Binary Trees 1257
- 21.2 Binary Search Tree Operations 1260
- 21.3 Template Considerations for Binary Search Trees 1277
Review Questions and Exercises 1283
Programming Challenges 1284

Appendix A: The ASCII Character Set 1287**Appendix B: Operator Precedence and Associativity 1289****Quick References 1291****Index 1293****Credit 1311****Online** The following appendices are available at www.pearsonhighered.com/gaddis.**Appendix C: Introduction to Flowcharting****Appendix D: Using UML in Class Design****Appendix E: Namespaces****Appendix F: Passing Command Line Arguments****Appendix G: Binary Numbers and Bitwise Operations****Appendix H: STL Algorithms****Appendix I: Multi-Source File Programs****Appendix J: Stream Member Functions for Formatting****Appendix K: Unions****Appendix L: Answers to Checkpoints****Appendix M: Answers to Odd Numbered Review Questions****Case Study 1: String Manipulation****Case Study 2: High Adventure Travel Agency—Part 1****Case Study 3: Loan Amortization****Case Study 4: Creating a String Class****Case Study 5: High Adventure Travel Agency—Part 2****Case Study 6: High Adventure Travel Agency—Part 3****Case Study 7: Intersection of Sets****Case Study 8: Sales Commission**

LOCATION OF VIDEONOTES IN THE TEXT

Chapter 1	Introduction to Flowcharting, p. 20 Designing a Program with Pseudocode, p. 20 Designing the Account Balance Program, p. 25 Predicting the Result of Problem 33, p. 26
Chapter 2	Using cout, p. 32 Variable Definitions, p. 38 Assignment Statements and Simple Math Expressions, p. 63 Solving the Restaurant Bill Problem, p. 81
Chapter 3	Reading Input with cin, p. 85 Formatting Numbers with setprecision, p. 113 Solving the Stadium Seating Problem, p. 144
Chapter 4	The if Statement, p. 156 The if/else statement, p. 168 The if/else if Statement, p. 178 Solving the Time Calculator Problem, p. 223
Chapter 5	The while Loop, p. 236 The for Loop, p. 251 Reading Data from a File, p. 278 Solving the Calories Burned Problem, p. 297
Chapter 6	Functions and Arguments, p. 317 Value-Returning Functions, p. 330 Solving the Markup Problem, p. 372
Chapter 7	Accessing Array Elements with a Loop, p. 386 Passing an Array to a Function, p. 413 Solving the Chips and Salsa Problem, p. 455
Chapter 8	The Binary Search, p. 466 The Selection Sort, p. 482 Solving the Charge Account Validation Modification Problem, p. 500
Chapter 9	Dynamically Allocating an Array, p. 531 Solving the Pointer Rewrite Problem, p. 554
Chapter 10	Writing a C-String-Handling Function, p. 585 More About the string Class, p. 591 Solving the Backward String Problem, p. 607

LOCATION OF VIDEONOTES IN THE TEXT (continued)



Chapter 11	Creating a Structure, p. 615 Passing a Structure to a Function, p. 631 Solving the Weather Statistics Problem, p. 659
Chapter 12	Passing File Stream Objects to Functions, p. 673 Working with Multiple Files, p. 686 Solving the File Encryption Filter Problem, p. 716
Chapter 13	Writing a Class, p. 726 Defining an Instance of a Class, p. 731 Solving the Employee Class Problem, p. 808
Chapter 14	Operator Overloading, p. 837 Class Aggregation, p. 866 Solving the NumDays Problem, p. 901
Chapter 15	Redefining a Base Class Function in a Derived Class, p. 936 Polymorphism, p. 947 Solving the Employee and ProductionWorker Classes Problem, p. 981
Chapter 16	Throwing an Exception, p. 990 Handling an Exception, p. 990 Writing a Function Template, p. 1008 Solving the Exception Project Problem, p. 1028
Chapter 17	The array Container, p. 1032 Iterators, p. 1034 The vector Container, p. 1040 The map Container, p. 1054 The set Container, p. 1079 Function Objects and Lambda Expressions, p. 1107 The Course Information Problem, p. 1119
Chapter 18	Appending a Node to a Linked List, p. 1126 Inserting a Node in a Linked List, p. 1133 Deleting a Node from a Linked List, p. 1137 Solving the Member Insertion by Position Problem, p. 1162
Chapter 19	Storing Objects in an STL stack, p. 1193 Storing Objects in an STL queue, p. 1216 Solving the File Compare Problem, p. 1221
Chapter 20	Reducing a Problem with Recursion, p. 1228 Solving the Recursive Multiplication Problem, p. 1255
Chapter 21	Inserting a Node in a Binary Tree, p. 1262 Deleting a Node from a Binary Tree, p. 1268 Solving the Node Counter Problem, p. 1284

Preface

Welcome to *Starting Out with C++: From Control Structures through Objects, 9th edition*. This book is intended for use in a two-semester C++ programming sequence, or an accelerated one-semester course. Students new to programming, as well as 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.

Changes in the Ninth Edition

This book's pedagogy, organization, and clear writing style remain the same as in the previous edition. Many improvements and updates have been made, which are summarized here:

- The material on the Standard Template Library (STL) has been completely rewritten and expanded into its own chapter. Previously, Chapter 16 covered exceptions, templates, and gave brief coverage to the STL. In this edition, Chapter 16 covers exceptions and templates, and Chapter 17 is a new chapter dedicated to the STL. The new chapter covers the following topics:
 - The `array` and `vector` classes
 - The various types of iterators
 - Emplacement versus insertion
 - The `map`, `multimap`, and `unordered_map` Classes
 - The `set`, `multiset`, and `unordered_set` Classes
 - Sorting and searching algorithms
 - Permutation algorithms
 - Set algorithms
 - Using function pointers with STL algorithms
 - Function objects, or functors
 - Lambda expressions
- Chapter 2 now includes a discussion of alternative forms of variable initialization, including functional notation, and brace notation (also known as uniform initialization).

- Chapter 3 now mentions the `round` function, introduced in C++ 11.
- Chapter 7 now introduces array initialization much earlier.
- In Chapter 8, the bubble sort algorithm has been rewritten and improved.
- A new example of sorting and searching a `vector` of strings has been added to Chapter 8.
- In Chapter 9, the section on smart pointers now gives an overview of `shared_ptr`s and `weak_ptr`s, in addition to the existing coverage of `unique_ptr`s.
- In Chapter 10, a new *In the Spotlight* section on string tokenizing has been added.
- Chapter 10 now covers the string-to-number conversion functions that were introduced in C++ 11.
- The material on unions that previously appeared in Chapter 11 has been moved to Appendix K, available on the book's companion Website.
- Chapter 13 has new sections covering:
 - Member initialization lists.
 - In-place initialization.
 - Constructor delegation.
- Several new topics were added to Chapter 14, including:
 - Rvalue references and move semantics.
 - Checking for self-assignment when overloading the `=` operator.
 - Using member initialization lists in aggregate classes.
- Chapter 15 includes a new section on constructor inheritance.
- Several new programming problems have been added throughout the book.

Organization of the Text

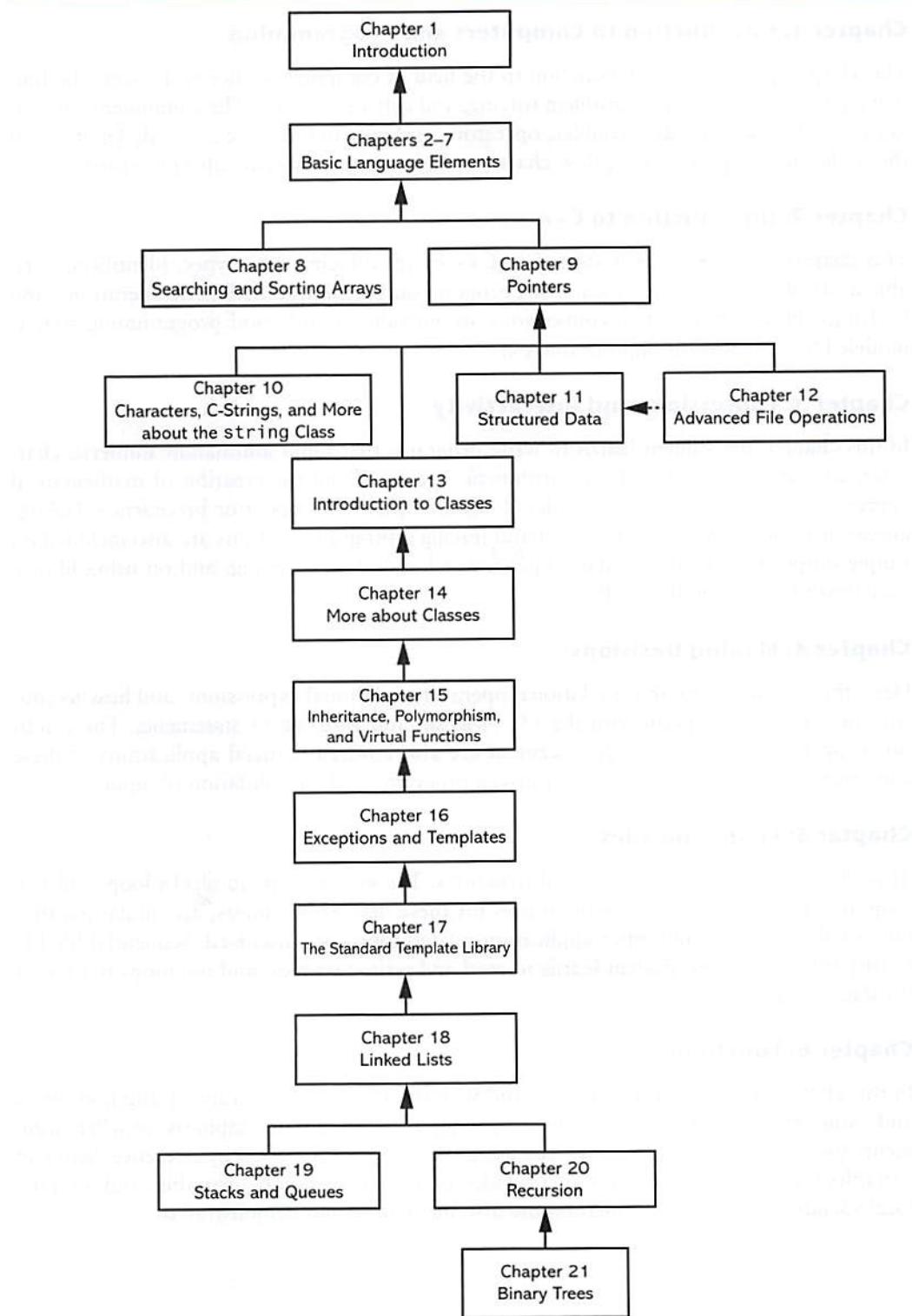
This text teaches C++ in a step-by-step fashion. Each chapter covers a major set of topics and builds knowledge as the student progresses through the book. Although the chapters can be easily taught in their existing sequence, some flexibility is provided. The diagram shown in Figure P-1 suggests possible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

Chapter 1 covers fundamental hardware, software, and programming concepts. You may choose to skip this chapter if the class is already familiar with those topics. Chapters 2 through 7 cover basic C++ syntax, data types, expressions, selection structures, repetition structures, functions, and arrays. Each of these chapters builds on the previous chapter and should be covered in the order presented.

After Chapter 7 has been covered, you may proceed to Chapter 8, or jump to Chapter 9.

After Chapter 9 has been covered, Chapter 10, 11, 12 or 13 may be covered. (If you jump to Chapter 12 at this point, you will need to postpone Sections 12.8, 12.9, and 12.10 until Chapter 11 has been covered.) After Chapter 13, you may cover Chapters 14 through 18 in sequence. Next, you can proceed to either Chapter 19 or Chapter 20. Finally, Chapter 21 may be covered.

This text's approach starts with a firm foundation in structured, procedural programming before delving fully into object-oriented programming and advanced data structures.

Figure P-1 Chapter dependency chart

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 programming, problem solving, and software design. The components of programs, such as key words, variables, operators, and punctuation, are covered. The tools of the trade, such as pseudocode, flow charts, and hierarchy charts, are also presented.

Chapter 2: Introduction to C++

This chapter gets the student started in C++ by introducing data types, identifiers, variable declarations, constants, comments, program output, simple arithmetic operations, and C-strings. Programming style conventions are introduced and good programming style is modeled here, as it is throughout the text.

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 in greater detail, with emphasis on operator precedence. Debugging is introduced, with a section on hand tracing a program. Sections are also included on simple output formatting, on data type conversion and type casting, and on using library functions that work with numbers.

Chapter 4: Making Decisions

Here, the student learns about relational operators, relational expressions, and how to control the flow of a program with the `if`, `if/else`, and `if/else if` statements. The conditional operator and the `switch` statement are also covered. Crucial applications of these constructs are covered, such as menu-driven programs and the validation of input.

Chapter 5: Loops and Files

This chapter covers repetition control structures. The `while` loop, `do-while` loop, and `for` loop are taught, along with common uses for these devices. Counters, accumulators, running totals, sentinels, and other application-related topics are discussed. Sequential file I/O is also introduced. The student learns to read and write text files, and use loops to process the data in a file.

Chapter 6: Functions

In this chapter, the student learns how and why to modularize programs, using both `void` and value returning functions. Argument 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.

Chapter 7: Arrays and Vectors

In this chapter, the student learns to create and work with single and multi-dimensional arrays. Many examples of array processing are provided including examples illustrating how to find the sum, average, highest, and lowest values in an array, and how to sum the rows, columns, and all elements of a two-dimensional array. 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.

Chapter 8: Searching and Sorting Arrays

Here, the student learns the basics of sorting arrays and searching for data stored in them. The chapter covers the Bubble Sort, Selection Sort, Linear Search, and Binary Search algorithms. There is also a section on sorting and searching STL vector objects.

Chapter 9: Pointers

This chapter explains how to use pointers. Pointers are compared to and contrasted with reference variables. Other topics include pointer arithmetic, initialization of pointers, relational comparison of pointers, pointers and arrays, pointers and functions, dynamic memory allocation, and more.

Chapter 10: Characters, C-Strings, and More about the string Class

This chapter discusses various ways to process text at a detailed level. Library functions for testing and manipulating characters are introduced. C-strings are discussed, and the technique of storing C-strings in char arrays is covered. An extensive discussion of the `string` class methods is also given.

Chapter 11: Structured Data

The student is introduced to abstract data types and taught how to create them using structures, unions, and enumerated data types. Discussions and examples include using pointers to structures, passing structures to functions, and returning structures from functions.

Chapter 12: Advanced File Operations

This chapter covers sequential access, random access, text, and binary files. The 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.

Chapter 13: Introduction to Classes

The student now shifts focus to the object-oriented paradigm. This chapter covers the fundamental concepts of classes. Member variables and functions are discussed. The student learns about private and public access specifications, and reasons to use each. The topics of constructors, overloaded constructors, and destructors are also presented. The chapter presents a section modeling classes with UML, and how to find the classes in a particular problem.

Chapter 14: More about Classes

This chapter continues the study of classes. Static members, friends, memberwise assignment, and copy constructors are discussed. The chapter also includes in-depth sections on operator overloading, object conversion, and object aggregation. There is also a section on class collaborations and the use of CRC cards.

Chapter 15: Inheritance, Polymorphism, and Virtual Functions

The study of classes continues in this chapter with the subjects of inheritance, polymorphism, and virtual member functions. The topics covered include base and derived class constructors and destructors, virtual member functions, base class pointers, static and dynamic binding, multiple inheritance, and class hierarchies.

Chapter 16: Exceptions and Templates

The student learns to develop enhanced error trapping techniques using exceptions. Discussion then turns to function and class templates as a method for reusing code.

Chapter 17: The Standard Template Library

This chapter discusses the containers, iterators, and algorithms in the Standard Template Library (STL). The specific containers covered are the `array`, `vector`, `map`, `multimap`, `unordered_map`, `set`, `multiset`, and `unordered_set` classes. The student then learns about sorting, searching, permutation, and set algorithms. The chapter concludes with a discussion of function objects (functors) and lambda functions.

Chapter 18: 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.

Chapter 19: 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.

Chapter 20: Recursion

This chapter discusses recursion and its use in problem solving. A visual trace of recursive calls is provided, and recursive applications are discussed. Many recursive algorithms are presented, including recursive functions for finding factorials, finding a greatest common denominator (GCD), performing a binary search, and sorting (QuickSort). The classic Towers of Hanoi example is also presented. For students who need more challenge, there is a section on exhaustive algorithms.

Chapter 21: 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.

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 chart showing the C++ operators and their precedence

The following appendices are available online at www.pearsonhighered.com/gaddis.

Appendix C: Introduction to Flowcharting

A brief introduction to flowcharting. This tutorial discusses sequence, decision, case, repetition, and module structures.

Appendix D: Using UML in Class Design

This appendix shows the student how to use the Unified Modeling Language to design classes. Notation for showing access specification, data types, parameters, return values, overloaded functions, composition, and inheritance are included.

Appendix E: Namespaces

This appendix explains namespaces and their purpose. Examples showing how to define a namespace and access its members are given.

Appendix F: Passing Command Line Arguments

Teaches the student how to write a C++ program that accepts arguments from the command line. This appendix will be useful to students working in a command line environment, such as Unix, Linux, or the Windows command prompt.

Appendix G: Binary Numbers and Bitwise Operations

A guide to the C++ bitwise operators, as well as a tutorial on the internal storage of integers

Appendix H: STL Algorithms

This appendix gives a summary of each of the function templates provided by the Standard Template Library (STL), and defined in the `<algorithm>` header file.

Appendix I: Multi-Source File Programs

Provides a tutorial on creating programs that consist of multiple source files. Function header files, class specification files, and class implementation files are discussed.

Appendix J: Stream Member Functions for Formatting

Covers stream member functions for formatting such as `setf`

Appendix K: Unions

This appendix introduces unions. It describes the purpose of unions and the difference between a union and a struct, demonstrates how to declare a union and define a union variable, and shows example programs that use unions.

Appendix L: Answers to Checkpoints

Students may test their own progress by comparing their answers to the Checkpoint exercises against this appendix. The answers to all Checkpoints are included.

Appendix M: Answers to Odd Numbered Review Questions

Another tool that students can use to gauge their progress

Features of the Text

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

Example Programs The text has hundreds of 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.



In the Spotlight

Each of these sections provides a programming problem and a detailed, step-by-step analysis showing the student how to solve it.



VideoNotes

A series of online videos, developed specifically for this book, is available for viewing at www.pearsonhighered.com/gaddis. 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 can be downloaded from the book's Website at www.pearsonhighered.com/gaddis. This allows students to 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 are notes that 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. 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. Answers to the odd-numbered review questions and review exercises can be downloaded from the book's Website at www.pearsonhighered.com/gaddis.

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.

**Software Development Project:
Serendipity Booksellers**

Available for download from the book's Website at www.pearsonhighered.com/gaddis. This is an ongoing project that instructors can optionally assign to teams of students. It systematically develops a "real-world" software package: a point-of-sale program for the fictitious Serendipity Booksellers organization. The Serendipity assignment for each chapter adds more functionality to the software, using constructs and techniques covered in that chapter. When complete, the program will act as a cash register, manage an inventory database, and produce a variety of reports.

C++ Quick Reference Guide**C++11**

For easy access, a quick reference guide to the C++ language is printed on the inside back cover of the book.

Throughout the text, new C++11 language features are introduced. Look for the C++11 icon to find these new features.

Supplements

Student Online Resources

Many student resources are available for this book from the publisher. The following items are available on the Computer Science Portal at www.pearsonhighered.com/gaddis:

- The source code for each example program in the book
- Access to the book's VideoNotes
- A full set of appendices, including answers to the Checkpoint questions and answers to the odd-numbered review questions
- A collection of valuable Case Studies
- The complete Serendipity Booksellers Project

Online Practice and Assessment with MyProgrammingLab

MyProgrammingLab helps students fully grasp the logic, semantics, and syntax of programming. Through practice exercises and immediate, personalized feedback, MyProgrammingLab improves the programming competence of beginning students who often struggle with the basic concepts and paradigms of popular high-level programming languages.

A self-study and homework tool, a MyProgrammingLab course consists of hundreds of small practice exercises organized around the structure of this textbook. For students, the system automatically detects errors in the logic and syntax of their code submissions and offers targeted hints that enable students to figure out what went wrong—and why. For instructors, a comprehensive gradebook tracks correct and incorrect answers and stores the code inputted by students for review.

MyProgrammingLab is offered to users of this book in partnership with Turing's Craft, the makers of the CodeLab interactive programming exercise system. For a full demonstration, to see feedback from instructors and students, or to get started using MyProgrammingLab in your course, visit www.myprogramminglab.com.

Instructor Resources

The following supplements are available only to qualified instructors:

- Answers to all Review Questions in the text
- Solutions for all Programming Challenges in the text
- PowerPoint presentation slides for every chapter
- Computerized test bank
- Answers to all Student Lab Manual questions
- Solutions for all Student Lab Manual programs

Visit the Pearson Instructor Resource Center (www.pearsonhighered.com/irc) for information on how to access instructor resources.

Textbook Web site

Student and instructor resources, including links to download Microsoft® Visual Studio Express and other popular IDEs, for all the books in the Gaddis *Starting Out with* series can be accessed at the following URL:

<http://www.pearsonhighered.com/gaddis>

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++: From 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
<p>LATE INTRODUCTION OF OBJECTS</p> <p>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.</p> <p>INTRODUCTION OF DATA STRUCTURES AND RECURSION</p> <p>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.</p>	<p>EARLIER INTRODUCTION OF OBJECTS</p> <p>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.</p> <p>INTRODUCTION OF DATA STRUCTURES AND RECURSION</p> <p>Linked lists, stacks and queues, and binary trees are introduced in the final chapters of the text, after the chapter on recursion.</p>

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 for the 9th Edition

Chia-Chin Chang
Lakeland College

William Duncan
Louisiana State University

Pranshu Gupta
DeSales University

Charles Hardnett
Gwinnett Technical College

Svetlana Marzelli
Atlantic Cape Community College

Jie Meichsner
St. Cloud State University

Ron Del Porto
Penn State Erie, The Behrend College

Lisa Rudnitsky
Baruch College

Reviewers for Previous Editions

Ahmad Abuhejleh
University of Wisconsin–River Falls

Karen M. Arlien
Bismarck State College

David Akins
El Camino College

Mary Astone
Troy University

Steve Allan
Utah State University

Ijaz A. Awan
Savannah State University

Vicki Allan
Utah State University

Robert Baird
Salt Lake Community College

- Don Biggerstaff
Fayetteville Technical Community College
- Michael Bolton
Northeastern Oklahoma State University
- Bill Brown
Pikes Peak Community College
- Robert Burn
Diablo Valley College
- Charles Cadenhead
Richland Community College
- Randall Campbell
Morningside College
- Wayne Caruolo
Red Rocks Community College
- Cathi Chambley-Miller
Aiken Technical College
- C.C. Chao
Jacksonville State University
- Joseph Chao
Bowling Green State University
- Royce Curtis
Western Wisconsin Technical College
- Joseph DeLibero
Arizona State University
- Michael Dixon
Sacramento City College
- Jeanne Douglas
University of Vermont
- Michael Dowell
Augusta State University
- Qiang Duan
Penn State University—Abington
- William E. Duncan
Louisiana State University
- Daniel Edwards
Ohlone College
- Judy Etchison
Southern Methodist University
- Dennis Fairclough
Utah Valley State College
- Xisheng Fang
Ohlone College
- Mark Fienup
University of Northern Iowa
- Richard Flint
North Central College
- Ann Ford Tyson
Florida State University
- Jeanette Gibbons
South Dakota State University
- James Gifford
University of Wisconsin-Stevens Point
- Leon Gleberman
Touro College
- Barbara Guillott
Louisiana State University
- Ranette Halverson, Ph.D.
Midwestern State University
- Ken Hang
Green River Community College
- Carol Hannahs
University of Kentucky
- Dennis Heckman
Portland Community College
- Ric Heishman
George Mason University
- Michael Hennessy
University of Oregon
- Ilga Higbee
Black Hawk College
- Patricia Hines
Brookdale Community College
- Mike Holland
Northern Virginia Community College
- Mary Hovik
Lehigh Carbon Community College

- Richard Hull
Lenoir-Rhyne College
- Kay Johnson
Community College of Rhode Island
- Chris Kardaras
North Central College
- Willard Keeling
Blue Ridge Community College
- A.J. Krygeris
Houston Community College
- Sheila Lancaster
Gadsden State Community College
- Ray Larson
Inver Hills Community College
- Michelle Levine
Broward College
- Jennifer Li
Ohlone College
- Norman H. Liebling
San Jacinto College
- Cindy Lindstrom
Lakeland College
- Zhu-qu Lu
University of Maine, Presque Isle
- Heidar Malki
University of Houston
- Debbie Mathews
J. Sargeant Reynolds Community College
- Rick Matzen
Northeastern State University
- Robert McDonald
East Stroudsburg University
- James McGuffee
Austin Community College
- Dean Mellas
Cerritos College
- Lisa Milkowski
Milwaukee School of Engineering
- Marguerite Nedreberg
Youngstown State University
- Lynne O'Hanlon
Los Angeles Pierce College
- Frank Paiano
Southwestern Community College
- Theresa Park
Texas State Technical College
- Mark Parker
Shoreline Community College
- Tino Posillico
SUNY Farmingdale
- Frederick Pratter
Eastern Oregon University
- Susan L. Quick
Penn State University
- Alberto Ramon
Diablo Valley College
- Bazlur Rasheed
Sault College of Applied Arts and Technology
- Farshad Ravanshad
Bergen Community College
- Susan Reeder
Seattle University
- Sandra Roberts
Snead College
- Lopa Roychoudhuri
Angelo State University
- Dolly Samson
Weber State University
- Ruth Sapir
SUNY Farmingdale
- Jason Schatz
City College of San Francisco
- Dr. Sung Shin
South Dakota State University
- Bari Siddique
University of Texas at Brownsville

- William Slater
Collin County Community College
- Shep Smithline
University of Minnesota
- Richard Snyder
Lehigh Carbon Community College
- Donald Southwell
Delta College
- Caroline St. Claire
North Central College
- Kirk Stephens
Southwestern Community College
- Cherie Stevens
South Florida Community College
- Dale Suggs
Campbell University
- Mark Swanson
Red Wing Technical College
- Ann Sudell Thorn
Del Mar College
- Martha Tillman
College of San Mateo
- Ralph Tomlinson
Iowa State University
- David Topham
Ohlone College
- Robert Tureman
Paul D. Camp Community College
- Arisa K. Ude
Richland College
- Peter van der Goes
Rose State College
- Stewart Venit
California State University, Los Angeles
- Judy Walters
North Central College
- John H. Whipple
Northampton Community College
- Aurelia Williams
Norfolk State University
- Chadd Williams
Pacific University
- Vida Winans
Illinois Institute of Technology

I would also like to thank my family and friends for their support in all of my projects. I am extremely fortunate to have Matt Goldstein as my editor, and Kristy Alaura as editorial assistant. Their guidance and encouragement made it a pleasure to write chapters and meet deadlines. I am also fortunate to have Demetrius Hall as my marketing manager. His hard work is truly inspiring, and he does a great job of getting this book out to the academic community. The production team, led by Sandra Rodriguez, worked tirelessly to make this book a reality. Thanks to you all!

About the Author

Tony Gaddis is the principal author of the *Starting Out with* series of textbooks. He has nearly two decades of experience teaching computer science courses, primarily at Haywood Community College. Tony is a highly acclaimed instructor who 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 textbooks covering Programming Logic and Design, Alice, C++, Java™, Microsoft® Visual Basic®, Microsoft® Visual C#, Python, and App Inventor, all published by Pearson.

MyProgrammingLab™

Through the power of practice and immediate personalized feedback, MyProgrammingLab helps improve your students' performance.

PROGRAMMING PRACTICE

With MyProgrammingLab, your students will gain first-hand programming experience in an interactive online environment.

IMMEDIATE, PERSONALIZED FEEDBACK

MyProgrammingLab automatically detects errors in the logic and syntax of their code submission and offers targeted hints that enables students to figure out what went wrong and why.

GRADUATED COMPLEXITY

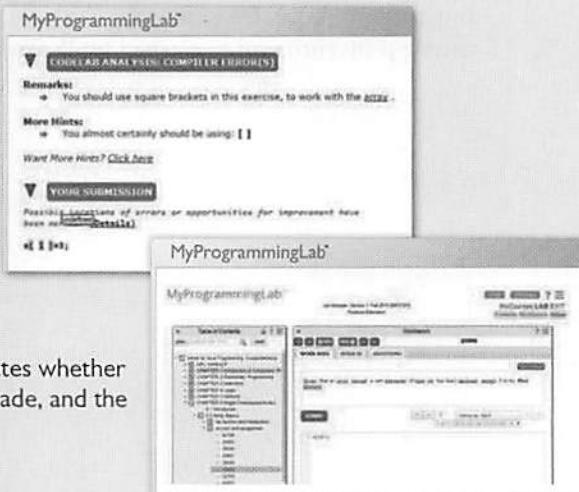
MyProgrammingLab breaks down programming concepts into short, understandable sequences of exercises. Within each sequence the level and sophistication of the exercises increase gradually but steadily.

DYNAMIC ROSTER

Students' submissions are stored in a roster that indicates whether the submission is correct, how many attempts were made, and the actual code submissions from each attempt.

PEARSON eTEXT

The Pearson eText gives students access to their textbook anytime, anywhere.



STEP-BY-STEP VIDEONOTE TUTORIALS

These step-by-step video tutorials enhance the programming concepts presented in select Pearson textbooks.

For more information and titles available with **MyProgrammingLab**, please visit www.myprogramminglab.com.

Copyright © 2018 Pearson Education, Inc. or its affiliate(s). All rights reserved. HELO88173 • 11/15

Introduction to Computers and Programming

TOPICS

- | | |
|---|--|
| 1.1 Why Program? | 1.4 What Is a Program Made of? |
| 1.2 Computer Systems: Hardware and Software | 1.5 Input, Processing, and Output |
| 1.3 Programs and Programming Languages | 1.6 The Programming Process |
| | 1.7 Procedural and Object-Oriented Programming |

1.1

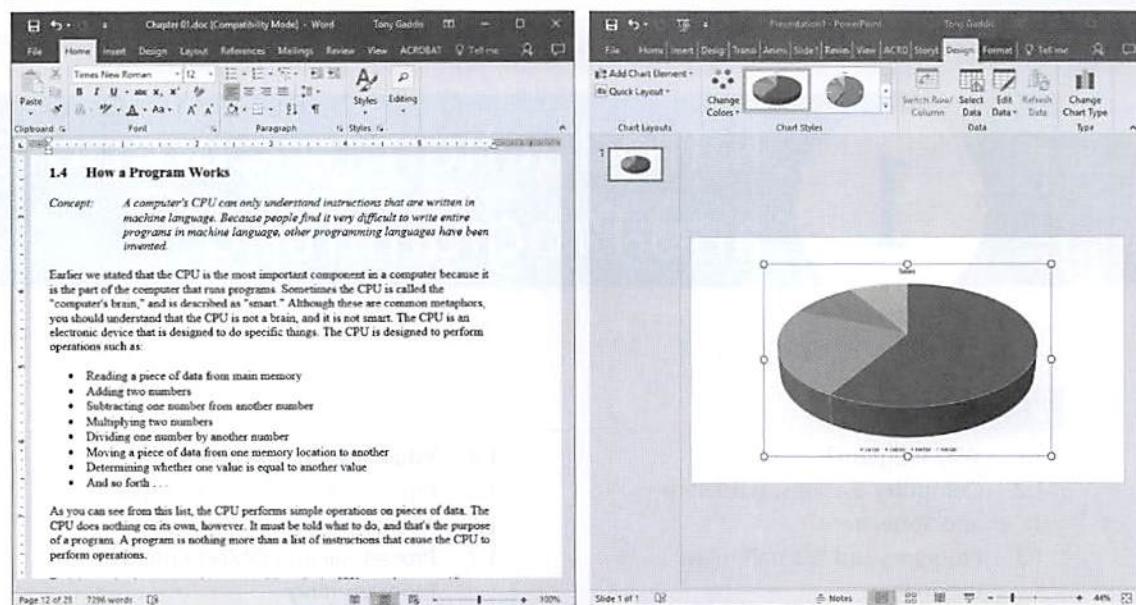
Why Program?

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

Think about some of the different ways that people use computers. In school, students use computers for tasks such as writing papers, searching for articles, sending e-mail, and participating in online classes. At work, people use computers to conduct business transactions, communicate with customers and coworkers, analyze data, make presentations, control machines in manufacturing facilities, and many many other tasks. At home, people use computers for tasks such as paying bills, shopping online, social networking, and playing computer games. And don't forget that smartphones, MP3 players, DVRs, car navigation systems, and many other devices are computers as well. The uses of computers are almost limitless in our everyday lives.

Computers can do such a wide variety of things because they can be programmed. This means computers are not designed to do just one job, but any job that their programs tell them to do. A *program* is a set of instructions that a computer follows to perform a task. For example, Figure 1-1 shows screens using Microsoft Word and PowerPoint, two commonly used programs.

Programs are commonly referred to as *software*. Software is essential to a computer because without software, a computer can do nothing. All of the software we use to make our computers useful is created by individuals known as programmers or software developers. A *programmer*, or *software developer*, is a person with the training and skills necessary to design, create, and test computer programs. Computer programming is an exciting and rewarding career. Today, you will find programmers working in business, medicine, government, law enforcement, agriculture, academia, entertainment, and almost every other field.

Figure 1-1 A word processing program and a presentation program

Computer programming is both an art and a science. It is an art because every aspect of a program should be carefully designed. 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"
- Documentation, help files, tutorials, and so on

There is also a scientific, or engineering, side to programming. Because programs rarely work right the first time they are written, a lot of testing, correction, and redesigning is required. This demands patience and persistence from 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 make 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.

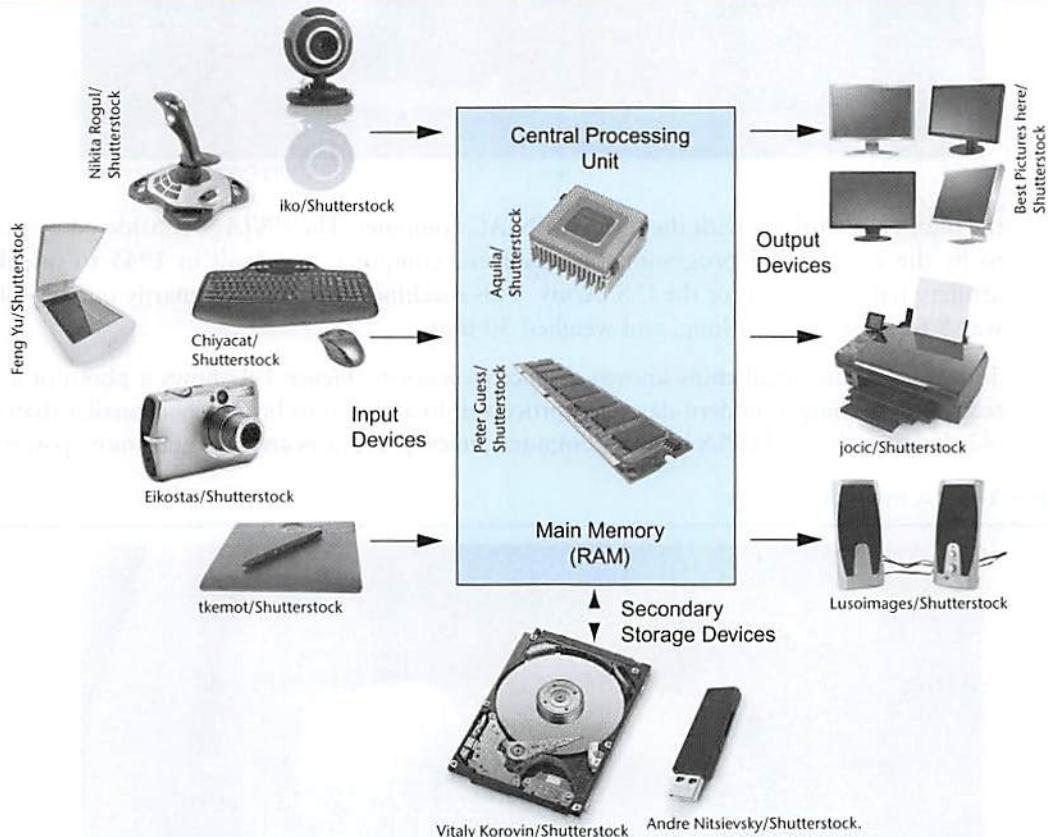
Hardware

Hardware refers to the physical components of which a computer is made. 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:

- The central processing unit (CPU)
- Main memory
- Secondary storage devices
- Input devices
- Output devices

The organization of a computer system is depicted in Figure 1-2.

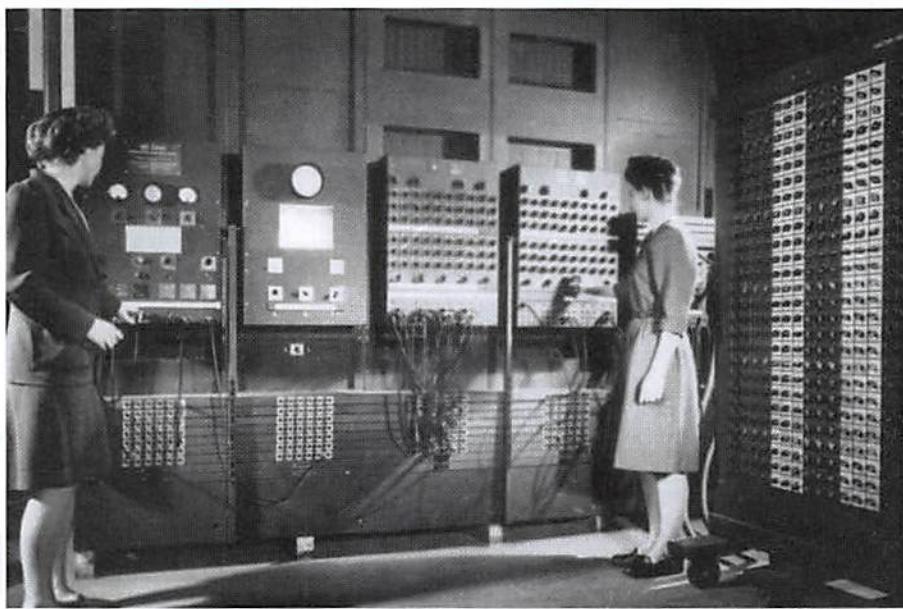
Figure 1-2 Typical devices in a computer system



The CPU

When a computer is performing the tasks that a program tells it to do, we say that the computer is *running* or *executing* the program. The *central processing unit*, or *CPU*, is the part of a computer that actually runs programs. The CPU is the most important component in a computer because without it, the computer could not run software.

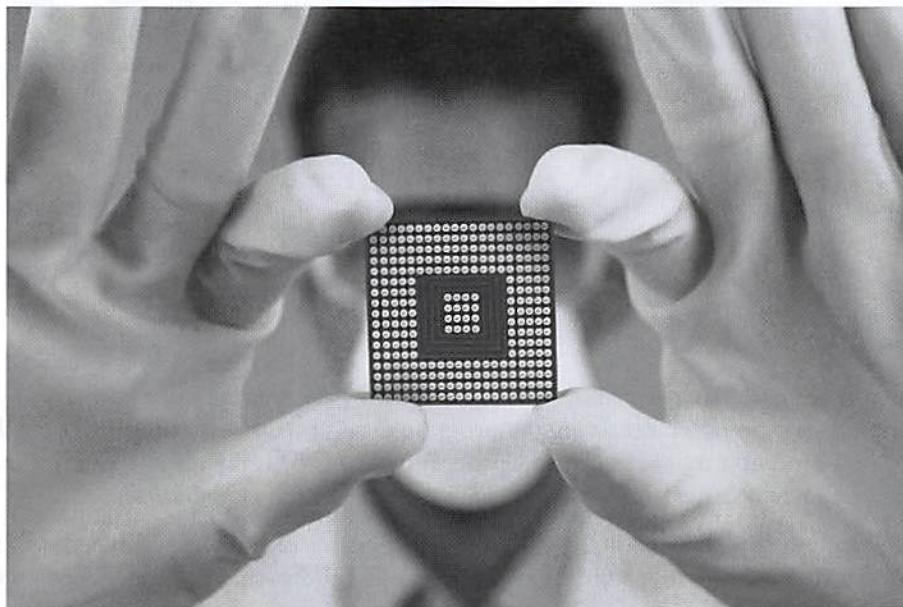
In the earliest computers, CPUs were huge devices made of electrical and mechanical components such as vacuum tubes and switches. Figure 1-3 shows such a device. The two women in

Figure 1-3 The ENIAC computer

U.S. Army Center of Military History

the photo are working with the historic ENIAC computer. The *ENIAC*, considered by many to be the world's first programmable electronic computer, was built in 1945 to calculate artillery ballistic tables for the U.S. Army. This machine, which was primarily one big CPU, was 8 feet tall, 100 feet long, and weighed 30 tons.

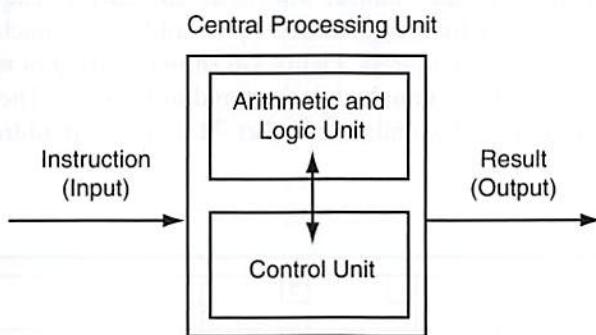
Today, CPUs are small chips known as *microprocessors*. Figure 1-4 shows a photo of a lab technician holding a modern-day microprocessor. In addition to being much smaller than the old electromechanical CPUs in early computers, microprocessors are also much more powerful.

Figure 1-4 A microprocessor

Creativa/Shutterstock

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 1-5.

Figure 1-5 Organization of a CPU



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:

- | | |
|----------------|---|
| <i>Fetch</i> | The CPU's control unit fetches, from main memory, the next instruction in the sequence of program instructions. |
| <i>Decode</i> | The instruction is encoded in the form of a number. The control unit decodes the instruction and generates an electronic signal. |
| <i>Execute</i> | 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

You can think of main memory as the computer's work area. This is where the computer stores a program while the program is running, as well as the data with which the program is working. For example, suppose you are using a word processing program to write an essay for one of your classes. While you do this, both the word processing program and the essay are stored in main memory.

Main memory is commonly known as *random-access memory* or RAM. It is called this because the CPU is able to quickly access data stored at any random location in RAM. RAM is usually a *volatile* type of memory that is used only for temporary storage while a program is running. When the computer is turned off, the contents of RAM are erased. Inside your computer, RAM is stored in small chips.

A computer's memory is divided into tiny storage locations known as bytes. One *byte* is enough memory to store only a letter of the alphabet or a small number. In order to do

anything meaningful, a computer must have lots of bytes. Most computers today have millions, or even billions, of bytes of memory.

Each byte is divided into eight smaller storage locations known as bits. The term *bit* stands for *binary digit*. Computer scientists usually think of bits as tiny switches that can be either on or off. Bits aren't actual "switches," however, at least not in the conventional sense. In most computer systems, bits are tiny electrical components that can hold either a positive or a negative charge. Computer scientists think of a positive charge as a switch in the *on* position, and a negative charge as a switch in the *off* position.

Each byte is assigned a unique number known as an *address*. The addresses are ordered from lowest to highest. A byte is identified by its address in much the same way a post office box is identified by an address. Figure 1-6 shows a group of memory cells with their addresses. In the illustration, sample data is stored in memory. The number 149 is stored in the cell with the address 16, and the number 72 is stored at address 23.

Figure 1-6 Memory

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	149	17	18
20	21	22	23	72	24	25	26	27	28

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. Programs are normally stored in secondary memory and loaded into main memory as needed. Important data such as word processing documents, payroll data, and inventory records is saved to secondary storage as well.

The most common type of secondary storage device is the disk drive. A traditional *disk drive* stores data by magnetically encoding it onto a circular disk. *Solid-state drives*, which store data in solid-state memory, are increasingly becoming popular. A solid-state drive has no moving parts and operates faster than a traditional disk drive. Most computers have some sort of secondary storage device, either a traditional disk drive or a solid-state drive, mounted inside their case. External storage devices can be used to create backup copies of important data or to move data to another computer. For example, *USB (Universal Serial Bus) drives* and *SD (Secure Digital) memory cards* are small devices that appear in the system as disk drives. They 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 used for data storage. Data is not recorded magnetically on an optical disc, but 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 mediums for creating backup copies of data.

Input Devices

Input is any data 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, touchscreen, scanner, digital camera, and microphone. Disk drives, CD/DVD drives, and USB 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 screens, printers, and speakers. Storage devices can also be considered output devices because the CPU sends them data to be saved.

Software

If a computer is to function, software is not optional. Everything a computer does, from the time you turn the power switch on until you shut the system down, is under the control of software. There are two general categories of software: system software and application software. Most computer programs clearly fit into one of these two categories. Let's take a closer look at each.

System Software

The programs that control and manage the basic operations of a computer are generally referred to as *system software*. System software typically includes the following types of programs:

- **Operating Systems**

An *operating system* is the most fundamental set of programs on a computer. The operating system controls the internal operations of the computer's hardware, manages all the devices connected to the computer, allows data to be saved to and retrieved from storage devices, and allows other programs to run on the computer.

- **Utility Programs**

A *utility program* performs a specialized task that enhances the computer's operation or safeguards data. Examples of utility programs are virus scanners, file-compression programs, and data-backup programs.

- **Software Development Tools**

The software tools that programmers use to create, modify, and test software are referred to as *software development tools*. Compilers and integrated development environments, which we will discuss later in this chapter, are examples of programs that fall into this category.

Application Software

Programs that make a computer useful for everyday tasks are known as *application software*. These are the programs that people normally spend most of their time running on their computers. Figure 1-1, at the beginning of this chapter, shows screens from two commonly used applications—Microsoft Word, a word processing program, and Microsoft

PowerPoint, a presentation program. Some other examples of application software are spreadsheet programs, e-mail programs, web browsers, and game programs.



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? What is its purpose?
- 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 fundamental set of programs control the internal operations of the computer's hardware?
- 1.9 What do you call a program that performs a specialized task, such as a virus scanner, a file-compression program, or a data-backup program?
- 1.10 Word processing programs, spreadsheet programs, e-mail programs, web browsers, and game programs belong to what category of software?

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 should 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 you and I 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*. If you were to look at a machine language program, you would see a stream of *binary numbers* (numbers consisting of only 1s and 0s). The binary numbers form machine language instructions, 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 then wanted to run it on computer B, which 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, 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++.

The “Program Output with Example Input” 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 rate. The program displays the earnings, which are \$150.



NOTE: The line numbers that are 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  
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;
```

(program continues)

Program 1-1

(continued)

```

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**

How much do you get paid per hour? **15**

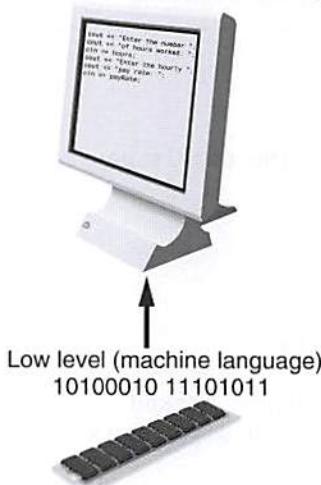
You have earned \$150

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 1-7 illustrates the concept of language levels.

Figure 1-7 Low-level versus high-level languages

High level (easily understood by humans)



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

In addition to the high-level features necessary for writing applications such as payroll systems and inventory programs, C++ 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. Since C++ evolved from C, it carries all of C’s low-level capabilities with it.

Table 1-1 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.
FORTRAN	Formula Translator. A language designed for programming complex mathematical algorithms.
COBOL	Common Business-Oriented Language. A language designed for business applications.
Pascal	A structured, general-purpose language designed primarily for teaching programming.
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.
Java	An object-oriented language that may be used to develop programs that run on many different types of devices.
JavaScript	JavaScript can be used to write small programs that run in webpages. Despite its name, JavaScript is not related to Java.
Python	Python is a general-purpose language created in the early 1990s. It has become popular in both business and academic applications.
Ruby	Ruby is a general-purpose language that was created in the 1990s. It is increasingly becoming a popular language 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 popular not only because of its mixture of low- and high-level features, but also because of its *portability*. This means that a C++ program can be written on one type of computer, then run on many other types of systems. This usually requires the program to be 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 macOS 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 that cause the preprocessor to modify 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*.

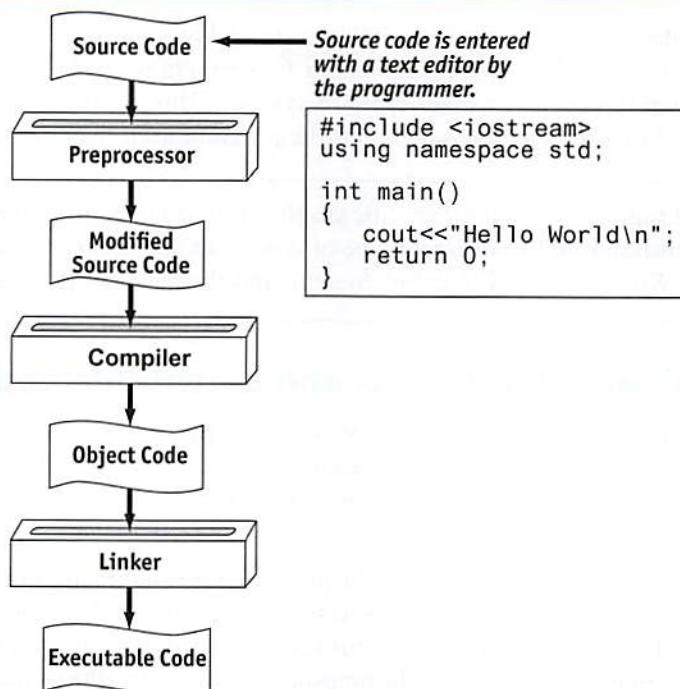
Although an object file contains machine language instructions, it is not a complete program. Here is why: C++ is conveniently equipped with a library of prewritten code for performing common operations or sometimes-difficult tasks. For example, the library contains hardware-specific code for displaying 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 *runtime 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 runtime 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 1-8 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
```

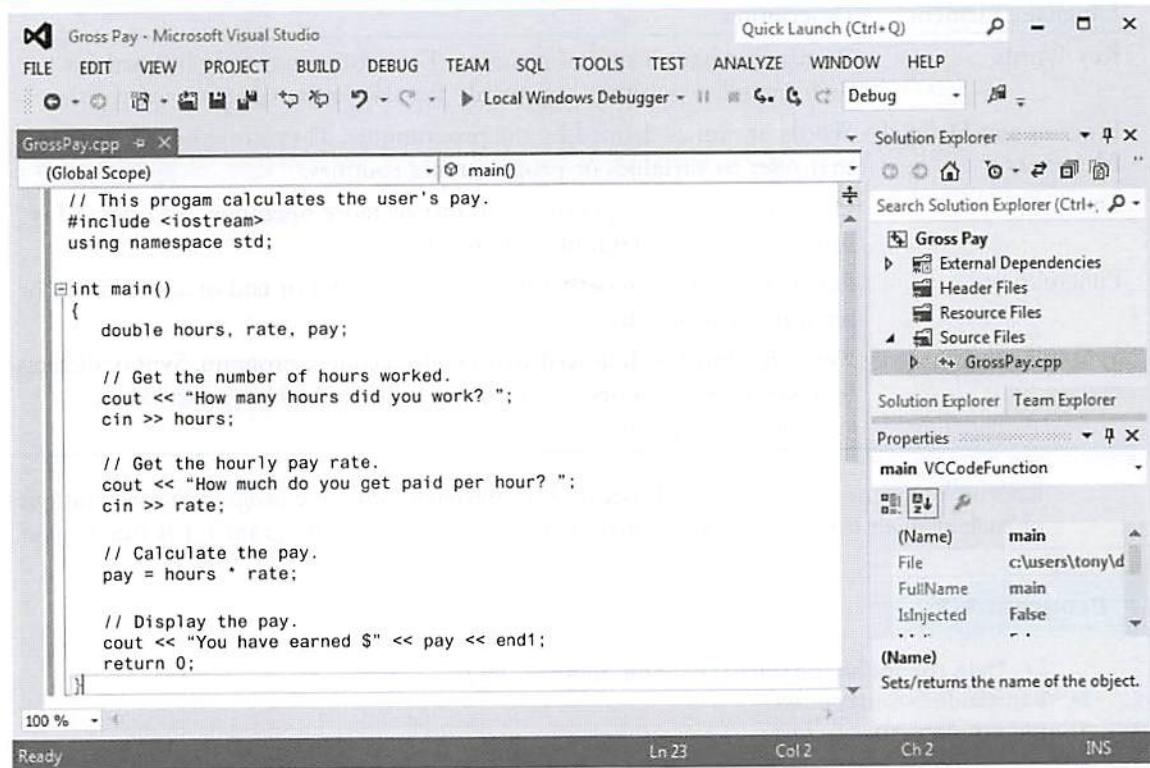
Figure 1-8 Translating a C++ source file to an executable file



Appendix F explains how compiling works in .Net. You can download Appendix F from the Computer Science Portal at www.pearsonhighered.com/gaddis.

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 1-9 shows a screen from the Microsoft Visual Studio IDE.

Figure 1-9 An integrated development environment (IDE)



Checkpoint

- 1.11 What is an algorithm?
- 1.12 Why were computer programming languages invented?
- 1.13 What is the difference between a high-level language and a low-level language?
- 1.14 What does *portability* mean?
- 1.15 Explain the operations carried out by the preprocessor, compiler, and linker.
- 1.16 Explain what is stored in a source file, an object file, and an executable file.
- 1.17 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 you will find in almost every language.

Table 1-2 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 end 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 your 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 code 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 lines 3, 7, 10, 11, 14, 15, 18, 21, and 22 all 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 jargon). 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 has 23 lines. 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? ";
```

This statement 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 often 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 information. The information stored in variables may change while the program is running (hence the name “variable”). Notice 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 has 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 will be 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 1-10 illustrates the way the variable name represents a memory location.

Figure 1-10 A variable name represents a memory location

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29

length

In Figure 1-10, 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. If it helps, you can think of a variable as a box that holds information. In Figure 1-10, 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 and characters. Numbers are used to perform mathematical operations, and characters are used to print data 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 this 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*. It is used to *define* one or more variables that will be used in the program and to indicate the type of data they will hold. The variable definition causes the variables to be created in memory, so all variables must be defined before they can be used. If you review the listing of Program 1-1, you will see that the variable definitions come before any other statements using those variables.



NOTE: Programmers often use the term “variable declaration” to mean the same thing as “variable definition.” Strictly speaking, there is a difference between the two terms. A definition statement always causes a variable to be created in memory. Some types of declaration statements, however, do not cause a variable to be created in memory. You will learn more about declarations later in this book.

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, then producing output. Input is information a

program collects from the outside world. It can be sent to the program from 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 pieces 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, and the result is assigned to the `pay` variable:

```
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 perform output:

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

These lines use the `cout` (pronounced “see out”) object to display messages on the computer’s screen. You will learn more details about the `cin` and `cout` objects in Chapter 2.



Checkpoint

- 1.18 Describe the difference between a key word and a programmer-defined identifier.
- 1.19 Describe the difference between operators and punctuation symbols.
- 1.20 Describe the difference between a program line and a statement.
- 1.21 Why are variables called “variable”?
- 1.22 What happens to a variable’s current contents when a new value is stored there?
- 1.23 What must take place in a program before a variable is used?
- 1.24 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 1-11 may help. These are the steps recommended for the process of writing a program.

Figure 1-11 Steps for writing a program

1. Clearly define what the program is to do.
2. Visualize the program running on the computer.
3. Use design tools such as a hierarchy chart, flowcharts, or pseudocode to create a model of the program.
4. Check the model for logical errors.
5. Type the code, save it, and compile it.
6. Correct any errors found during compilation. Repeat Steps 5 and 6 as many times as necessary.
7. Run the program with test data for input.
8. Correct any errors found while running the program. Repeat Steps 5 through 8 as many times as necessary.
9. Validate the results of the program.

The steps listed in Figure 1-11 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. Clearly define what the program is to do.

This step requires that you identify the purpose of the program, the information that is to be input, the processing that is to take place, and the desired output. Let's examine each of these requirements for the example program:

<i>Purpose</i>	To calculate the user's gross pay.
<i>Input</i>	Number of hours worked, hourly pay rate.
<i>Process</i>	Multiply number of hours worked by hourly pay rate. The result is the user's gross pay.
<i>Output</i>	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:

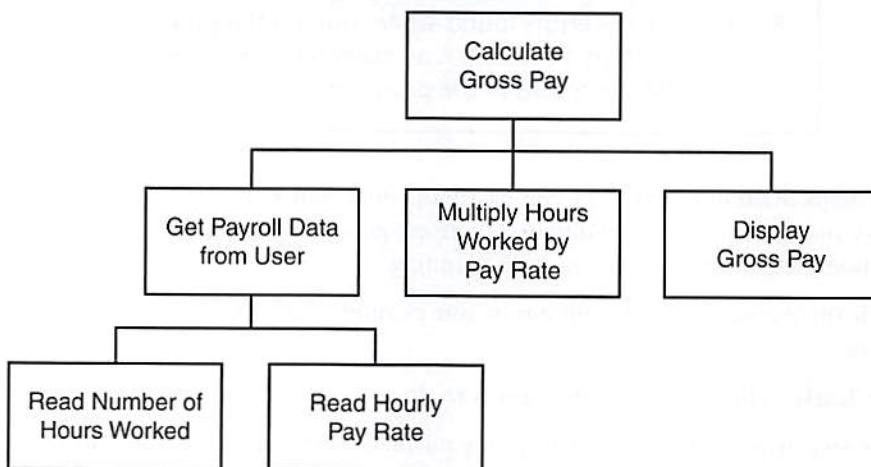
```
How many hours did you work? 10
How much do you get paid per hour? 15
You have earned $150
```

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 concerns, you will have already determined most of the program's output.

3. Use design tools such as a hierarchy chart, flowcharts, or pseudocode 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 1-12 shows a hierarchy chart for the pay-calculating program.

Figure 1-12 Hierarchy chart



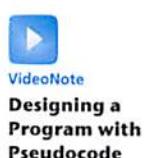
A hierarchy chart begins with the overall task then refines it into smaller subtasks. Each of the subtasks is then refined into even smaller sets of subtasks, until each is small enough to be easily performed. For instance, in Figure 1-12, the overall task “Calculate Gross Pay” is listed in the top-level 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*.

A *flowchart* is a diagram that shows the logical flow of a program. It is a useful tool for planning each operation a program performs and the order in which the operations are to occur. For more information see Appendix C, 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 in a language that’s “almost” a programming language, but still very similar to natural language. For example, here is pseudocode that describes the pay-calculating program:

*Get payroll data.
Calculate gross pay.
Display gross pay.*

Although the pseudocode above gives a broad view of the program, it doesn’t reveal all the program’s details. A more detailed version of the pseudocode follows:



Display "How many hours did you work?".

Input hours.

Display "How much do you get paid per hour?".

Input rate.

Store the value of hours times rate in the pay variable.

Display the value in the pay variable.

Notice the pseudocode contains statements that look more like commands than the English statements that describe the algorithm in Section 1.4 (What Is a Program Made of?). The pseudocode even names variables and describes mathematical operations.

4. Check the model for logical errors.

Logical errors are mistakes that cause the program to produce erroneous results. Once a hierarchy chart, flowchart, or pseudocode model of the program is assembled, it should be checked for these 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 next step is attempted.

5. Type the code, save it, and compile it.

Once a model of the program (hierarchy chart, flowchart, or pseudocode) has been created, checked, and corrected, the programmer is ready to write source code on the computer. The programmer saves the source code to a file and begins the process of translating it to machine language. During this step, the compiler will find any syntax errors that may exist in the program.

6. Correct any errors found during compilation. Repeat Steps 5 and 6 as many times as necessary.

If the compiler reports any errors, they must be corrected. Steps 5 and 6 must be repeated until the program is free of compile-time errors.

7. Run the program with test data for input.

Once an executable file is generated, the program is ready to be tested for runtime errors. A runtime error is an error that occurs while the program is running. These are usually logical errors, such as mathematical mistakes.

Testing for runtime errors requires that the program be executed with sample data or sample input. The sample data should be such that the correct output can be predicted. If the program does not produce the correct output, a logical error is present in the program.

8. Correct any errors found while running the program. Repeat Steps 5 through 8 as many times as necessary.

When runtime errors are found 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 runtime 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, many errors can be located and corrected. If an 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 an error is due to an incomplete understanding of the program requirements, then you must restate the program purpose and modify the hierarchy and/or flowcharts, pseudocode, and source code. The program must then be saved, recompiled, and retested. This means Steps 5 through 8 must be repeated until the program reliably produces satisfactory results.

9. Validate the results of the program.

When you believe you have corrected all the runtime errors, enter test data and determine whether the program solves the original problem.

What Is Software Engineering?

The field of software engineering encompasses the whole 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 and flowcharts
- 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. The subtasks identified in a top-down design can easily become 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 finally become a single software application.

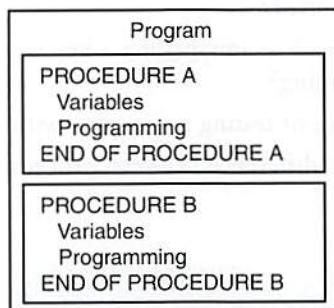
1.7

Procedural and Object-Oriented Programming

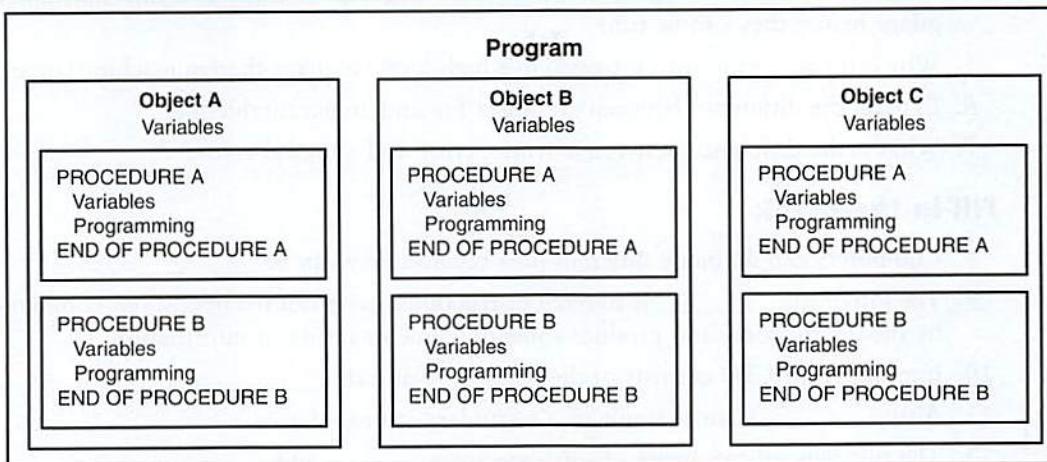
CONCEPT: Procedural programming and object-oriented programming are two ways of thinking about software development and program design.

C++ is a language that can be used for two methods of writing computer programs: *procedural programming* and *object-oriented programming*. This book is designed to teach you some of both.

In procedural programming, the programmer constructs procedures (or functions, as they are called in C++). The procedures are collections of programming statements that perform a specific task. The procedures each contain their own variables and commonly share variables with other procedures. This is illustrated in Figure 1-13.

Figure 1-13 Procedures in a program

Procedural programming is centered on the procedure, or function. Object-oriented programming (OOP), on the other hand, is centered on the object. An object is a programming element that contains data and the procedures that operate on the data. It is a self-contained unit. This is illustrated in Figure 1-14.

Figure 1-14 Objects in a program

The objects contain, within themselves, both information and the ability to manipulate the information. Operations are carried out on the information in an object by sending the object a *message*. When an object receives a message instructing it to perform some operation, it carries out the instruction. As you study this text, you will encounter many other aspects of object-oriented programming.



Checkpoint

- 1.25 What four items should you identify when defining what a program is to do?
- 1.26 What does it mean to “visualize a program running”? What is the value of such an activity?
- 1.27 What is a hierarchy chart?
- 1.28 Describe the process of desk-checking.

- 1.29 Describe what a compiler does with a program's source code.
- 1.30 What is a runtime error?
- 1.31 Is a syntax error (such as misspelling a key word) found by the compiler or when the program is running?
- 1.32 What is the purpose of testing a program with sample data or input?
- 1.33 Briefly describe the difference between procedural and object-oriented programming.

Review Questions and Exercises

Short Answer

1. Both main memory and secondary storage are types of memory. Describe the difference between the two.
2. What is the difference between system software and application software?
3. What type of software controls the internal operations of the computer's hardware?
4. Why must programs written in a high-level language be translated into machine language before they can be run?
5. Why is it easier to write a program in a high-level language than in machine language?
6. Explain the difference between an object file and an executable file.
7. What is the difference between a syntax error and a logical error?

Fill-in-the-Blank

8. Computers can do many different jobs because they can be _____.
9. The job of the _____ is to fetch instructions, carry out the operations commanded by the instructions, and produce some outcome or resultant information.
10. Internally, the CPU consists of the _____ and the _____.
11. A(n) _____ is an example of a secondary storage device.
12. The two general categories of software are _____ and _____.
13. A program is a set of _____.
14. Since computers can't be programmed in natural human language, algorithms must be written in a(n) _____ language.
15. _____ is the only language computers really process.
16. _____ languages are close to the level of humans in terms of readability.
17. _____ languages are close to the level of the computer.
18. A program's ability to run on several different types of computer systems is called _____.
19. Words that have special meaning in a programming language are called _____.
20. Words or names defined by the programmer are called _____.
21. _____ are characters or symbols that perform operations on one or more operands.

22. _____ characters or symbols mark the beginning or end of programming statements, or separate items in a list.
23. The rules that must be followed when constructing a program are called _____.
24. A(n) _____ is a named storage location.
25. A variable must be _____ before it can be used in a program.
26. The three primary activities of a program are _____, _____, and _____.
27. _____ is information a program gathers from the outside world.
28. _____ is information a program sends to the outside world.
29. A(n) _____ is a diagram that graphically illustrates the structure of a program.

Algorithm Workbench

Draw hierarchy charts or flowcharts that depict the programs described below. (See Appendix C for instructions on creating flowcharts.)

30. Available Credit

The following steps should be followed in a program that calculates a customer's available credit:

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.

31. Sales Tax

Design a hierarchy chart or flowchart 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:

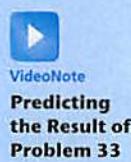
- The sales tax for the purchase
- The total of the sale

32. Account Balance

Design a hierarchy chart or flowchart for a program that calculates the current balance in a savings 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
- The monthly interest rate

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



Predict the Result

Questions 33–35 are programs expressed as English statements. What would each display on the screen if they were actual programs?

33. 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.
34. 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.
35. 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

36. The following *pseudocode algorithm* has an error. The program is supposed to ask the user for the length and width of a rectangular room, then display the room's area. The program must multiply the width by the length in order to determine the 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.

```

TOPICS

- | | |
|--|---|
| 2.1 The Parts of a C++ Program | 2.10 The <code>bool</code> Data Type |
| 2.2 The <code>cout</code> Object | 2.11 Determining the Size of a Data Type |
| 2.3 The <code>#include</code> Directive | 2.12 More about Variable Assignments and Initialization |
| 2.4 Variables, Literals, and Assignment Statements | 2.13 Scope |
| 2.5 Identifiers | 2.14 Arithmetic Operators |
| 2.6 Integer Data Types | 2.15 Comments |
| 2.7 The <code>char</code> Data Type | 2.16 Named Constants |
| 2.8 The C++ <code>string</code> Class | 2.17 Programming Style |
| 2.9 Floating-Point Data Types | |

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.

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.

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!

Line 2 looks like this:

```
#include <iostream>
```

Because this line starts with a #, it is called 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, known as a *header file*, in the program. It is called a header file because it should be included at the head, or top, of a program. The word that is enclosed in brackets, `iostream`, is the name of the header file that is to be included. (The name of the file is `iostream`. The brackets `<>` indicate that it is a standard C++ header file.) The `<iostream>` header 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>` header file must be included. The contents of the `<iostream>` file are included in the program at the point the `#include` statement appears.

Line 3 reads:

```
using namespace std;
```

Programs usually contain several items with unique names. In this chapter, you will learn to create variables. In Chapter 6, you will learn to create functions. In Chapter 13, 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 reason the program needs access to the `std` namespace is 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.

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 collectively has a name. The name of this function is `main`, and the set of parentheses that follows the name indicate 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 are ever reading someone else’s C++ program and want to find where it starts, just look for the function named `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 “INT” the same as “int.” This is true for all the C++ key words.

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 content 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* or *string constant*.



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.

At the end of the line is a semicolon. Just as a period marks the end of a sentence, a semicolon marks the end of a complete statement in C++. Comments are ignored by the compiler, so the semicolon isn't required at the end of a comment. Preprocessor directives, like `#include` statements, simply end at the end of the line and never require semicolons. The beginning of a function, like `int main()`, is not a complete statement, so you don't place a semicolon there either.

It might seem that the rules for where to put a semicolon are not clear at all. Rather than worry about it now, just concentrate on learning the parts of a program. You'll soon get a feel for where you should and should not use semicolons.

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.

Line 9 contains the closing brace:

```
}
```

This brace marks the end of the `main` function. Since `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
<code>//</code>	Double slash	Marks the beginning of a comment.
<code>#</code>	Pound sign	Marks the beginning of a preprocessor directive.
<code>< ></code>	Opening and closing brackets	Enclose a filename when used with the <code>#include</code> directive.
<code>()</code>	Opening and closing parentheses	Used in naming a function, as in <code>int main()</code> .
<code>{ }</code>	Opening and closing braces	Enclose a group of statements, such as the contents of a function.
<code>" "</code>	Opening and closing quotation marks	Enclose a string of characters, such as a message that is to be printed on the screen.
<code>;</code>	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
return 0;
```

```
#include <iostream>
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

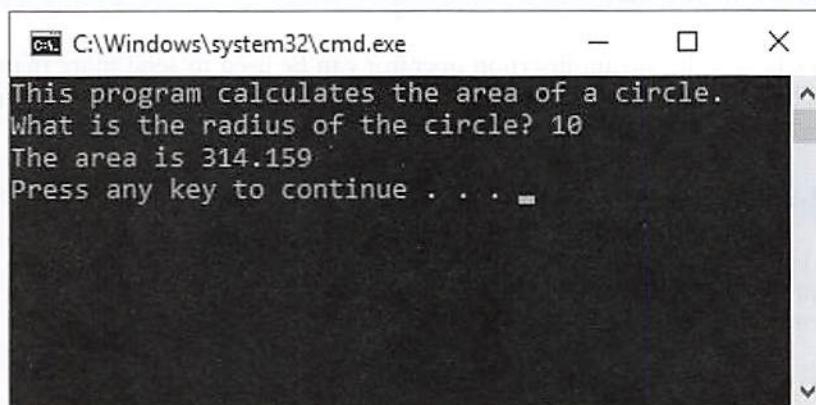
The cout Object

CONCEPT: Use the `cout` object 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. In C++, you use the `cout` object to produce console output. (You can think of the word `cout` as meaning *console output*.)

Figure 2-1 A console window



`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 Program 2-1:

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



Notice 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` then displayed on the screen.

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`. This is illustrated in Figure 2-2.

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

Figure 2-2 Think of `<<` as an arrow pointing to `cout`

```
cout << "Programming is great fun!";  
Think of the stream insertion operator as an  
arrow that points toward cout.  
cout ← "Programming is great fun!";
```

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 that of 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 up into two programming statements, this program will still display the message on one 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 is an example.

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.

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 sellers during the month of June: Computer
games Coffee Aspirin

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 on the screen, it wraps around to a second line when displayed. The reason the output comes out as one long line is because 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 (which is pronounced “end-line” or “end-L”). Program 2-5 is an example.

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;
```

(program continues)

Program 2-5*(continued)*

```
9     cout << "Computer games" << endl;
10    cout << "Coffee" << endl;
11    cout << "Aspirin" << endl;
12    return 0;
13 }
```

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 position 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. The following statements show an example.

```
cout << "My pets are" << endl << "dog";
cout << endl << "cat" << endl << "bird" << endl;
```

Another way to cause cout to go to a new line is to insert an *escape sequence* in the string itself. An escape sequence starts with the backslash character (\) and is followed by one or more control characters. It allows you to control the way output is displayed by embedding commands within the string itself. Program 2-6 is an example.

Program 2-6

```
1 // Yet 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

The *newline escape sequence* is `\n`. 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 many programmers prefer it.

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

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

In this code, the programmer accidentally wrote `/n` when he or she meant to write `\n`. The `cout` object will simply display the `/n` characters on the screen. This code will display 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:

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

This code will result in an error because the `\n` sequences are not inside quotation marks. We can correct the code by placing the `\n` sequences inside the string literals, as shown here:

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

There are many escape sequences in C++. They 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
<code>\n</code>	Newline	Causes the cursor to go to the next line for subsequent printing.
<code>\t</code>	Horizontal tab	Causes the cursor to skip over to the next tab stop.
<code>\a</code>	Alarm	Causes the computer to beep.
<code>\b</code>	Backspace	Causes the cursor to back up, or move left one position.
<code>\r</code>	Return	Causes the cursor to go to the beginning of the current line, not the next line.
<code>\\\</code>	Backslash	Causes a backslash to be printed.
<code>\'</code>	Single quote	Causes a single quotation mark to be printed.
<code>\\"</code>	Double quote	Causes a double quotation mark to be printed.



WARNING! When using escape sequences, do not put a space between the backslash and the control character.

When you type an escape sequence in a string, you type two characters (a backslash followed by another character). However, an escape sequence is stored in memory as a single character. For example, consider the following string literal:

```
"One\nTwo\nThree\n"
```

The diagram in Figure 2-3 breaks this string into its individual characters. Notice how each of the \n escape sequences are considered one character.

Figure 2-3 Individual characters in a string



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 `iostream`. 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 quickly become tiring as well. The solution is to let the preprocessor insert the contents of `iostream` automatically.



WARNING! Do not put semicolons at the end of processor directives. Because preprocessor directives are not C++ statements, they do not require semicolons. In many cases, an error will result from a preprocessor directive 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

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

```
cout << "Success\n";
cout << " Success\n\n";
int main()
cout << "Success";
}
using namespace std;
// It's a mad, mad program
#include <iostream>
cout << "Success\n";
{
return 0;
```

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

```
Success
Success Success
```

```
Success
```

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

- 2.3 Study the following program and show what it will print on the screen:

```
// The Works of Wolfgang
#include <iostream>
using namespace std;
int main()
{
    cout << "The works of Wolfgang\ninclude the following";
    cout << "\nThe Turkish March" << endl;
    cout << "and Symphony No. 40 ";
    cout << "in G minor." << endl;
    return 0;
}
```

- 2.4 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. Place a comment with today's date at the top of the program. Test your program by compiling and running it.

2.4

Variables, Literals, and Assignment Statements

CONCEPT: Variables represent storage locations in the computer's memory. Literals are constant values that are assigned to variables.

As you discovered in Chapter 1, 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 types of information they will hold. Program 2-7 is an example of a C++ program with a variable. Take a look 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. This line indicates the variable's name is `number`. The word `int` stands for integer, so `number` will only be used to hold integer numbers. Later in this chapter, you will learn all the types of data that C++ allows.

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 in number is " << number << endl;  
11    return 0;  
12 }
```

Program Output

The value in number is 5



NOTE: You must have a definition for every variable you intend to use in a program. In C++, variable definitions can appear at any point in the program. Later in this chapter, and throughout the book, you will learn the best places to define variables.

Notice variable definitions end with a semicolon. Now look at line 9:

```
number = 5;
```

This is called an *assignment*. The equal sign is an operator that copies the value on its right (5) into the variable named on its left (*number*). After this line executes, *number* will be set to 5.



NOTE: This line does not print anything on the computer's screen. It runs silently behind the scenes, storing a value in RAM.

Look at line 10:

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

The second item sent to cout is the variable name *number*. When you send a variable name to cout, it prints the variable's contents. Notice there are no quotation marks around *number*. Look at what happens in Program 2-8.

Program 2-8

```
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 in number is " << "number" << endl;  
11    return 0;  
12 }
```

Program Output

```
The value in number is number
```

When double quotation marks are placed around the word *number*, it becomes a string literal and is no longer a variable name. 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 has no quotation marks around it, for the same reason.

Sometimes a Number Isn't a Number

As shown in Program 2-8, just placing quotation marks around a variable name changes the program's results. 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 2-8, the number 5 was assigned to the variable *number*. It would have been incorrect to perform the assignment this way:

```
number = "5";
```

In this line, 5 is no longer an integer, but a string literal. Because `number` was defined as 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 students 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. Before numbers can be displayed on the screen, they must first be converted to strings. (Fortunately, `cout` handles the conversion automatically when you send a number to it.)

Literals

A literal is a piece of data that is written directly into a program’s code. One of the most common uses of literals is to assign a value to a variable. For example, in the following statement, assume `number` is an `int` variable. The statement assigns the literal value 100 to the variable `number`.

```
number = 100;
```

Another common use of literals is to display something on the screen. For example, the following statement displays the string literal “Welcome to my program.”

```
cout << "Welcome to my program." << endl;
```

Program 2-9 shows an example that uses a variable and several literals.

Program 2-9

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

Program Output

Today we sold 20 bushels of apples.

Of course, the variable is `apples`. It is defined as an integer. Table 2-3 lists the literals found in the program.

Table 2-3 Literals and Their Types

Literal	Type of Literal
20	Integer literal
"Today we sold"	String literal
"bushels of apples.\n"	String literal
0	Integer literal



NOTE: Literals are also called constants.



Checkpoint

- 2.5 Examine the following program:

```
// This program uses variables and literals.
#include <iostream>
using namespace std;
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;
}
```

List all the variables and literals that appear in the program.

- 2.6 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;
}
```

2.5

Identifiers

CONCEPT: Choose variable names that indicate what the variables are 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 they are all lowercase.

Table 2-4 The C++ Key Words

alignas	const	for	private	throw
alignof	constexpr	friend	protected	true
and	const_cast	goto	public	try
and_eq	continue	if	register	typedef
asm	decltype	inline	reinterpret_cast	typeid
auto	default	int	return	typename
bitand	delete	long	short	union
bitor	do	mutable	signed	unsigned
bool	double	namespace	sizeof	using
break	dynamic_cast	new	static	virtual
case	else	noexcept	static_assert	void
catch	enum	not	static_cast	volatile
char	explicit	not_eq	struct	wchar_t
char16_t	export	nullptr	switch	while
char32_t	extern	operator	template	xor
class	false	or	this	xor_eq
compl	float	or_eq	thread_local	

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

```
int x;
```

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 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 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. Unfortunately, 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 succeeding words makes `itemsOrdered` easier to read. It should be mentioned that 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 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 tells whether each is legal or illegal in C++.

Table 2-5 Some Variable Names

Variable Name	Legal or Illegal?
<code>dayOfWeek</code>	Legal.
<code>3dGraph</code>	Illegal. Variable names cannot begin with a digit.
<code>_employee_num</code>	Legal.
<code>June1997</code>	Legal.
<code>Mixture#3</code>	Illegal. Variable names may only use letters, digits, or underscores.

2.6

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, 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 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. Integers are whole numbers like 12, 157, -34, and 2. Floating-point numbers have a decimal point, like 23.7, 189.0231, and 0.987. Additionally, the integer and floating-point data types are broken into even more classifications. Before we discuss the character data type, let's carefully examine the variations of numeric data.

Your primary considerations for selecting a numeric data type are

- The largest and smallest numbers that may be stored in the variable
- How much memory the variable uses
- Whether the variable stores signed or unsigned numbers
- The number of decimal places of precision the variable has

The size of a variable is the number of bytes of memory it uses. Typically, the larger a variable is, the greater the range it can hold.

Table 2-6 shows the C++ integer data types with their typical sizes and ranges.



NOTE: The data type sizes and ranges shown in Table 2-6 are typical on many systems. Depending on your operating system, the sizes and ranges may be different.

Table 2-6 Integer Data Types

Data Type	Typical Size	Typical Range
short int	2 bytes	-32,768 to +32,767
unsigned short int	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 int	4 bytes	-2,147,483,648 to +2,147,483,647
unsigned long int	4 bytes	0 to 4,294,967,295
long long int	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long int	8 bytes	0 to 18,446,744,073,709,551,615

Here are some examples of variable definitions:

```
int days;  
unsigned int speed;  
short int month;  
unsigned short int amount;  
long int deficit;  
unsigned long int insects;
```

Each of the data types in Table 2-6, except `int`, can be abbreviated as follows:

- `short int` can be abbreviated as `short`
- `unsigned short int` can be abbreviated as `unsigned short`
- `unsigned int` can be abbreviated as `unsigned`
- `long int` can be abbreviated as `long`
- `unsigned long int` can be abbreviated as `unsigned long`
- `long long int` can be abbreviated as `long long`
- `unsigned long long int` can be abbreviated as `unsigned long long`

Because they simplify definition statements, programmers commonly use the abbreviated data type names. Here are some examples:

```
unsigned speed;  
short month;  
unsigned short amount;  
long deficit;  
unsigned long insects;  
long long grandTotal;  
unsigned long long lightYearDistance;
```

Unsigned data types can only store nonnegative values. They can be used when you know your program will not encounter negative values. For example, variables that hold ages or weights would rarely hold numbers less than 0.

Notice in Table 2-6 the `int` and `long` data types have the same sizes and ranges, and that 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.
- The `long long int` and the `unsigned long long int` data types are guaranteed to be at least 8 bytes (64 bits) in size.

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



NOTE: The `long long int` and the `unsigned long long int` data types were introduced in C++ 11.

As mentioned before, variables are defined by stating the data type key word followed by the name of the variable. In Program 2-10, an integer, an unsigned integer, and a long integer have been defined.

Program 2-10

```
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  
11     checking = -20;  
12     miles = 4276;  
13     diameter = 100000;  
14     cout << "We have made a long journey of " << miles;  
15     cout << " miles.\n";  
16     cout << "Our checking account balance is " << checking;  
17     cout << "\nThe galaxy is about " << diameter;  
18     cout << " light years in diameter.\n";  
19  
20 }
```

Program Output

We have made a long journey of 4276 miles.

Our checking account balance is -20

The galaxy is about 100000 light years in diameter.

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 could be defined separately, like this:

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

It is easier, however, to combine both variable definitions on one line:

```
int length, width;
```

You can define several variables of the same type like this, simply by separating their names with commas. Program 2-11 illustrates this.

Program 2-11

```
1 // This program shows three variables defined on the same line.  
2 #include <iostream>  
3 using namespace std;
```

```
4 int main()
5 {
6     int floors, rooms, suites;
7
8     floors = 15;
9     rooms = 300;
10    suites = 30;
11    cout << "The Grande Hotel has " << floors << " floors\n";
12    cout << "with " << rooms << " rooms and " << suites;
13    cout << " suites.\n";
14
15    return 0;
16 }
```

Program Output

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

Integer and Long Integer Literals

In C++, if a numeric literal is an integer (not written with a decimal point) and it fits within the range of an `int` (see Table 2-6 for the minimum and maximum values), then the numeric literal is treated as an `int`. A numeric literal that is treated as an `int` is called an *integer literal*. For example, look at lines 9, 10, and 11 in Program 2-11:

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

Each of these statements assigns an integer literal to a variable.

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 literal, 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 literal to be stored as a long integer by placing the letter L at the end of the number. Here is an example:

```
long amount;
amount = 32L;
```

The first statement defines a `long` variable named `amount`. The second statement assigns the literal value 32 to the `amount` variable. In the second statement, the literal is written as `32L`, which makes it a *long integer literal*. This means the literal is treated as a `long`.

11

If you want an integer literal to be treated as a `long long` or `int`, you can append LL at the end of the number. Here is an example:

```
long long amount;
amount = 32LL;
```

The first statement defines a `long long` variable named `amount`. The second statement assigns the literal value 32 to the `amount` variable. In the second statement, the literal is written as `32LL`, which makes it a *long long integer literal*. This means the literal is treated as a `long long int`.



TIP: When writing long integer literals or long long integer literals, you can use either an uppercase or a lowercase L. Because the lowercase l looks like the number 1, you should always use the uppercase L.

If You Plan to Continue in Computer Science: Hexadecimal and Octal Literals

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 literals 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. It is important, however, that you understand this material. Good programmers should 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 literals.



Checkpoint

- 2.7 Which of the following are illegal variable names, and why?
`X`
`99bottles`
`july97`
`theSalesFigureForFiscalYear98`
`r&d`
`grade_report`
- 2.8 Is the variable name `Sales` the same as `sales`? Why or why not?
- 2.9 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 240,000 to 140,000, what data type would be best?
 - C) Which of the following literals uses more memory? 20 or 20L

- 2.10 On any computer, which data type uses more memory, an integer or an unsigned integer?

2.7

The char Data Type

The `char` data type is used to store individual characters. A variable of the `char` data type can hold only one character at a time. Here is an example of how you might declare a `char` variable:

```
char letter;
```

This statement declares a `char` variable named `letter`, which can store one character. In C++, *character literals* are enclosed in single quotation marks. Here is an example showing how we would assign a character to the `letter` variable:

```
letter = 'g';
```

This statement assigns the character 'g' to the `letter` variable. Because `char` variables can hold only one character, they are not compatible with strings. For example, you cannot assign a string to a `char` variable, even if the string contains only one character. The following statement, for example, will not compile because it attempts to assign a string literal to a `char` variable.

```
letter = "g"; // ERROR! Cannot assign a string to a char
```

It is important that you do not confuse character literals, which are enclosed in single quotation marks, with string literals, which are enclosed in double quotation marks.

Program 2-12 shows an example program that works with characters.

Program 2-12

```
1 // This program works with characters.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     char letter;
8
9     letter = 'A';
10    cout << letter << endl;
11    letter = 'B';
12    cout << letter << endl;
13    return 0;
14 }
```

Program Output

```
A
B
```

Although the `char` data type is used for storing characters, it is actually an integer data type that typically uses 1 byte of memory. (The size is system dependent. 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, is 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 the number 65 is the code for A, 66 is the code for B, and so on. Program 2-13 demonstrates that when you work with characters, you are actually working with numbers.

Program 2-13

```
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    letter = 66;
13    cout << letter << endl;
14    return 0;
15 }
```

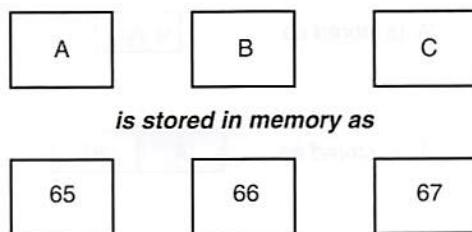
Program Output

```
A
B
```

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

The Difference between String Literals and Character Literals

It is important that you do not confuse character literals with string literals. Strings, which are a series of characters stored in consecutive memory locations, can be virtually any

Figure 2-4 Characters and their ASCII codes

length. This means that there must be some way for the program to know how long a string is. In C++, an extra byte is appended to the end of string literals when they are stored in memory. In this last byte, the number 0 is stored. It is called the *null terminator* or *null character*, and it 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 ASCII code 48 corresponds to the character '0', whereas the null terminator is the same as the 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, however, you use ASCII code 0.

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

Figure 2-5 The string literal "Sebastian"

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.

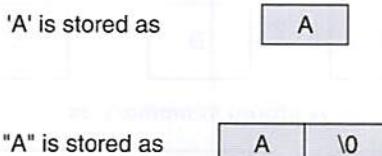
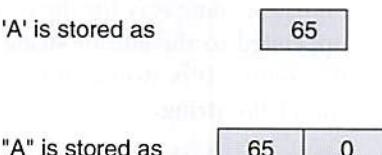
The null terminator is another example of something that sits quietly in the background. It doesn't print on the screen when you display a string, but nevertheless, it is there silently doing its job.



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

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

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 2-6 'A' as a character and "A" as a string**Figure 2-7** ASCII codes

Because `char` variables are only large enough to hold one character, you cannot assign string literals to them. For example, the following code defines a `char` variable named `letter`. The character literal 'A' can be assigned to the variable, but the string literal "A" cannot.

```
char letter;  
  
letter = 'A'; // This will work.  
letter = "A"; // This will not work!
```

One final topic about characters should be discussed. 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 2-14 shows the use of `\n` as a character literal, 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 2-14

```
1 // This program uses character literals.  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main()  
6 {  
7     char letter;  
8  
9     letter = 'A';  
10    cout << letter << '\n';  
11    letter = 'B';
```

```
12     cout << letter << '\n';
13     return 0;
14 }
```

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 literals are always stored in memory with a null terminator at the end. This marks the end of the string.
- Character literals are enclosed in single quotation marks.
- String literals are enclosed in double quotation marks.
- Escape sequences such as '\n' are stored internally as a single character.

2.8

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. Although 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";
```

You can use `cout` to display the value of the `movieTitle` object, as shown in the next statement:

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

Program 2-15 is a complete program that demonstrates the preceding statements.

Program 2-15

```
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

- 2.11 What are the ASCII codes for the following characters? (Refer to Appendix A.)

C
F
W

- 2.12 Which of the following is a character literal?

'B'
"B"

- 2.13 Assuming the `char` data type uses 1 byte of memory, how many bytes do the following literals use?

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

- 2.14 Write a program that has the following character variables: `first`, `middle`, and `last`. Store your initials in these variables then display them on the screen.

- 2.15 What is wrong with the following program statement?

char letter = "Z";

- 2.16 What header file must you include in order to use `string` objects?

- 2.17 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.

2.9

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`. Of course, the exact sizes of these data types are 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 typical sizes and ranges of floating-point data types.

Table 2-8 Floating-Point Data Types on PCs

Data Type	Key Word	Description
Single precision	<code>float</code>	4 bytes. Numbers between $\pm 3.4\text{E}-38$ and $\pm 3.4\text{E}38$
Double precision	<code>double</code>	8 bytes. Numbers between $\pm 1.7\text{E}-308$ and $\pm 1.7\text{E}308$
Long double precision	<code>long double</code>	8 bytes*. Numbers between $\pm 1.7\text{E}-308$ and $\pm 1.7\text{E}308$

*Some compilers use 10 bytes for `long double`s. This allows a range of $\pm 3.4\text{E}-4932$ to $\pm 1.1\text{E}4832$.

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

Floating-Point Literals

Floating-point literals may be expressed in a variety of ways. As shown in Program 2-16, 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 literals were written as `1.495979E11` and `1.989E30`, but the program printed them as `1.49598e+011` and `1.989e+30`. The two sets of numbers are equivalent. (The plus sign in front of the exponent is also optional.) In Chapter 3, you will learn to control the way `cout` displays E notation numbers.

Program 2-16

```
1 // This program uses floating-point data types.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     float distance;
8     double mass;
9
10    distance = 1.495979E11;
11    mass = 1.989E30;
12    cout << "The Sun is " << distance << " meters away.\n";
13    cout << "The Sun's mass is " << mass << " kilograms.\n";
14    return 0;
15 }
```

Program Output

```
The Sun is 1.49598e+011 meters away.
The Sun's mass is 1.989e+030 kilograms.
```

You can also express floating-point literals in decimal notation. The literal `1.495979E11` could have been written as:

`149597900000.00`

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 literals are equivalent:

`1.4959E11`
`1.4959e11`
`1.4959E+11`
`1.4959e+11`
`149590000000.00`

Floating-point literals are normally stored in memory as `doubles`. But remember, C++ provides tools for handling just about any situation. Just in case you need to force a literal to be stored as a `float`, you can append the letter F or f to the end of it. For example, the following literals would be stored as `floats`:

```
1.2F  
45.907f
```



NOTE: Because floating-point literals are normally stored in memory as `doubles`, most compilers issue a warning message when you assign a floating-point literal 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 warning message by appending the f suffix to the floating-point literal, as shown below:

```
num = 14.725f;
```

If you want to force a value to be stored as a `long double`, append an L or l to it, as in the following examples:

```
1034.56L  
89.21
```

The compiler won't confuse these with long integers because they have decimal points. (Remember, the lowercase L looks so much like the number 1 that you should always use the uppercase L when suffixing literals.)

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. For example, look at the following code:

```
int number;  
number = 7.5; // Assigns 7 to number
```

This code attempts to assign the floating-point value 7.5 to the integer variable `number`. As a result, the value 7 will be assigned to `number`, with the fractional part discarded. When part of a value is discarded, it 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 i;  
float f;  
f = 7.5;  
i = f; // Assigns 7 to i.
```

When the `float` variable `f` is assigned to the `int` variable `i`, the value being assigned (7.5) is truncated. After this code executes, `i` will hold the value 7 and `f` will hold the value 7.5.



NOTE: When a floating-point value is truncated, it is not rounded. Assigning the value 7.9 to an `int` variable will result in the value 7 being stored in the variable.



WARNING! Floating-point variables can hold a much larger range of values than integer variables can. If a floating-point value is being 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.

2.10 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 small integer variables that are suitable for holding `true` or `false` values. Program 2-17 demonstrates the definition and assignment of a `bool` variable.

Program 2-17

```

1 // This program demonstrates 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    boolValue = false;
12    cout << boolValue << endl;
13    return 0;
14 }
```

Program Output

```

1
0
```

As you can see from the program output, the value `true` is represented in memory by the number 1, and `false` is represented by 0. You will not be using `bool` variables until Chapter 4, however, so just remember they are useful for evaluating conditions that are either `true` or `false`.

2.11 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 2-18 illustrates its use. The first line that uses the operator is line 10:

```
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 invoked anywhere you can use an unsigned integer, including in mathematical operations.

Program 2-18

```
1 // This program determines the size of integers, long
2 // integers, and long doubles.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     long double apple;
9
10    cout << "The size of an integer is " << sizeof(int);
11    cout << " bytes.\n";
12    cout << "The size of a long integer is " << sizeof(long);
13    cout << " bytes.\n";
14    cout << "An apple can be eaten in " << sizeof(apple);
15    cout << " bytes!\n";
16    return 0;
17 }
```

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

- 2.18 Yes or No: Is there an unsigned floating-point data type? If so, what is it?
- 2.19 How would the following number in scientific notation be represented in E notation?
$$6.31 \times 10^{17}$$
- 2.20 Write a program that defines an integer variable named `age` and a `float` variable named `weight`. Store your age and weight, as literals, in the variables. The program should display these values on the screen in a manner similar to the following:

My age is 26 and my weight is 180 pounds.

(Feel free to lie to the computer about your age and your weight—it'll never know!)

2.12

More about 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.

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 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.

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*. It is called an *lvalue* because it is a value that may appear on the left side of an assignment operator. An *lvalue* is something that identifies a place in memory whose contents may be changed. Most of the time this will be a variable name. The operand on the right side of the `=` symbol must be an *rvalue*. An *rvalue* is any expression that has a value. The assignment statement takes the value of the *rvalue* and puts it in the memory location of the object identified by the *lvalue*.

You may also assign values to variables as part of the definition. This is called *initialization*. Program 2-19 shows how it is done.

Program 2-19

```
1 // This program shows variable initialization.  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main()  
6 {  
7     int month = 2, days = 28;  
8  
9     cout << "Month " << month << " has " << days << " days.\n";  
10    return 0;  
11 }
```

Program Output

Month 2 has 28 days.

As you can see, this simplifies the program and reduces the number of statements that must be typed by the programmer. Here are examples of other definition statements that perform initialization:

```
double interestRate = 12.9;  
char stockCode = 'D';  
long customerNum = 459L;
```

Of course, there are always variations on a theme. C++ allows you to define several variables and only initialize some of them. Here is an example of such a definition:

```
int flightNum = 89, travelTime, departure = 10, distance;
```

The variable `flightNum` is initialized to 89 and `departure` is initialized to 10. The variables `travelTime` and `distance` remain uninitialized.

11**Declaring Variables with the `auto` Key Word**

C++ 11 introduces an alternative way to define variables, using the `auto` key word and an initialization value. Here is an example:

```
auto amount = 100;
```

Notice this statement uses the word `auto` instead of a data type. The `auto` key word tells the compiler to determine the variable's data type from the initialization value. In this example, the initialization value, 100, is an `int`, so `amount` will be an `int` variable. Here are other examples:

```
auto interestRate = 12.0;  
auto stockCode = 'D';  
auto customerNum = 459L;
```

In this code, the `interestRate` variable will be a `double` because the initialization value, 12.0, is a `double`. The `stockCode` variable will be a `char` because the initialization value,

'D', is a char. The customerNum variable will be a long because the initialization value, 459L, is a long.

These examples show how to use the auto key word, but they don't really show its usefulness. The auto key word is intended to simplify the syntax of declarations that are more complex than the ones shown here. Later in the book, you will see examples of how the auto key word can improve the readability of complex definition statements.

Alternative Forms of Variable Initialization

The most common way to initialize a variable is to use the assignment operator in the variable definition statement, as shown here:

```
int value = 5;
```

However, there are two alternative techniques for initializing variables. The first is to enclose the initialization value in a set of parentheses, just after the variable name. Here is how you would define the value variable and initialize it to 5 using this technique:

```
int value(5);
```

11

The second alternative, introduced in C++ 11, is to enclose the initialization value in a set of curly braces, just after the variable name. Here is how you would define the value variable and initialize it to 5 using the brace notation:

```
int value {5}; // This only works with C++ 11 or higher.
```

Most programmers continue to use the assignment operator to initialize regular variables, as we do throughout this book. However, the brace notation offers an advantage. It checks to make sure the value you are initializing the variable with matches the data type of the variable. For example, assume that doubleVal is a double variable with 6.2 stored in it. Using the assignment operator, it is possible to write either of the following statements:

```
int value1 = 4.9; // This will store 4 in value1  
int value2 = doubleVal; // This will store 6 in value2
```

In both cases, the fractional part of the value will be truncated before it is assigned to the variable being defined. This could cause problems, yet C++ compilers allow it. They do issue a warning, but they still build an executable file you can run. If the brace notation is used, however, the compiler indicates that these statements produce an error, and no executable is created. You will have to fix the errors and rebuild the project before you can run the program.

2.13 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 the variable may be used. The rules that define a variable's scope are complex, and you will

only be introduced to the concept here. In other sections of the book, we will revisit this topic and expand on it.

The first rule of scope you should learn is that a variable cannot be used in any part of the program before the definition. Program 2-20 illustrates this.

Program 2-20

```
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 not 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 your 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.

2.14 Arithmetic Operators

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

C++ offers a multitude of 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 literal 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. The assignment operator is in this category. Ternary operators, as you may have guessed, require three operands. C++ has only one ternary operator, which will be discussed in Chapter 4.

Arithmetic operations are very common in programming. Table 2-9 shows the common arithmetic operators in C++.



Table 2-9 Fundamental Arithmetic Operators

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

Each of these operators works as you probably expect. The addition operator returns the sum of its two operands. In the following assignment statement, the variable `amount` will be assigned the value 12:

```
amount = 4 + 8;
```

The subtraction operator returns the value of its right operand subtracted from its left operand. This statement will assign the value 90 to `temperature`:

```
temperature = 120 - 30;
```

The multiplication operator returns the product of its two operands. In the following statement, `markUp` is assigned the value 3:

```
markUp = 12 * 0.25;
```

The division operator returns the quotient of its left operand divided by its right operand. In the next statement, `points` is assigned the value 5:

```
points = 100 / 20;
```

It is important to note that when both of the division operator's operands are integers, the result of the division will also be an integer. If the result has a fractional part, it will be thrown away. We will discuss this behavior, which is known as *integer division*, in greater detail later in this section.

The modulus operator, which only works with integer operands, returns the remainder of an integer division. The following statement assigns 2 to `leftOver`:

```
leftOver = 17 % 3;
```

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 \$27.78 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 2-21 shows the C++ code for the program.

Program 2-21

```
1 // This program calculates hourly wages, including overtime.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     double regularWages,           // To hold regular wages
8         basePayRate = 18.25,       // Base pay rate
9         regularHours = 40.0,        // Hours worked less overtime
10        overtimeWages,          // To hold overtime wages
11        overtimePayRate = 27.78,   // Overtime pay rate
12        overtimeHours = 10,        // Overtime hours worked
13        totalWages;             // To hold total wages
14
15    // Calculate the regular wages.
16    regularWages = basePayRate * regularHours;
17
18    // Calculate the overtime wages.
19    overtimeWages = overtimePayRate * overtimeHours;
20
21    // Calculate the total wages.
22    totalWages = regularWages + overtimeWages;
23
24    // Display the total wages.
25    cout << "Wages for this week are $" << totalWages << endl;
26    return 0;
27 }
```

Program Output

Wages for this week are \$1007.8

Let's take a closer look at the program. As mentioned in the comments, there are variables for regular wages, base pay rate, regular hours worked, overtime wages, overtime pay rate, overtime hours worked, and total wages.

Here is line 16, which multiplies `basePayRate` by `regularHours` and stores the result in `regularWages`:

```
regularWages = basePayRate * regularHours;
```

Here is line 19, which multiplies `overtimePayRate` by `overtimeHours` and stores the result in `overtimeWages`:

```
overtimeWages = overtimePayRate * overtimeHours;
```

Line 22 adds the regular wages and the overtime wages and stores the result in `totalWages`:

```
totalWages = regularWages + overtimeWages;
```

Line 25 displays the message on the screen reporting the week's wages.

Integer Division

When both operands of a division statement are integers, the statement will result in *integer division*. This means the result of the division will be an integer as well. If there is a remainder, it will be discarded. For example, look at the following code:

```
double number;  
number = 5 / 2;
```

This code divides 5 by 2 and assigns the result to the `number` variable. What will be stored in `number`? You would probably assume that 2.5 would be stored in `number` because that is the result your calculator shows when you divide 5 by 2. However, that is not what happens when the previous C++ code is executed. Because the numbers 5 and 2 are both integers, the fractional part of the result will be thrown away, or truncated. As a result, the value 2 will be assigned to the `number` variable.

In the previous code, it doesn't matter that the `number` variable is declared as a `double` because the fractional part of the result is discarded before the assignment takes place. In order for a division operation to return a floating-point value, one of the operands must be of a floating-point data type. For example, the previous code could be written as follows:

```
double number;  
number = 5.0 / 2;
```

In this code, 5.0 is treated as a floating-point number, so the division operation will return a floating-point number. The result of the division is 2.5.

In the Spotlight:

Calculating Percentages and Discounts

Determining percentages is a common calculation in computer programming. Although the % symbol is used in general mathematics to indicate a percentage, most programming languages (including C++) do not use the % symbol for this purpose. In a program, you have to convert a percentage to a floating-point number, just as you would if you were using a calculator. For example, 50 percent would be written as 0.5, and 2 percent would be written as 0.02.

Let's look at an example. Suppose you earn \$6,000 per month and you are allowed to contribute a portion of your gross monthly pay to a retirement plan. You want to determine the amount of your pay that will go into the plan if you contribute 5 percent, 7 percent, or 10 percent of your gross wages. To make this determination, you write the program shown in Program 2-22.

Program 2-22

```
1 // This program calculates the amount of pay that  
2 // will be contributed to a retirement plan if 5%,  
3 // 7%, or 10% of monthly pay is withheld.  
4 #include <iostream>
```

```
5 using namespace std;
6
7 int main()
8 {
9     // Variables to hold the monthly pay and the
10    // amount of contribution.
11    double monthlyPay = 6000.0, contribution;
12
13    // Calculate and display a 5% contribution.
14    contribution = monthlyPay * 0.05;
15    cout << "5 percent is $" << contribution
16        << " per month.\n";
17
18    // Calculate and display a 7% contribution.
19    contribution = monthlyPay * 0.07;
20    cout << "7 percent is $" << contribution
21        << " per month.\n";
22
23    // Calculate and display a 10% contribution.
24    contribution = monthlyPay * 0.1;
25    cout << "10 percent is $" << contribution
26        << " per month.\n";
27
28    return 0;
29 }
```

Program Output

```
5 percent is $300 per month.
7 percent is $420 per month.
10 percent is $600 per month.
```

Line 11 defines two variables: `monthlyPay` and `contribution`. The `monthlyPay` variable, which is initialized with the value 6000.0, holds the amount of your monthly pay. The `contribution` variable will hold the amount of a contribution to the retirement plan.

The statements in lines 14 through 16 calculate and display 5 percent of the monthly pay. The calculation is done in line 14, where the `monthlyPay` variable is multiplied by 0.05. The result is assigned to the `contribution` variable, which is then displayed in line 15.

Similar steps are taken in lines 18 through 21, which calculate and display 7 percent of the monthly pay, and lines 24 through 26, which calculate and display 10 percent of the monthly pay.

Calculating a Percentage Discount

Another common calculation is determining a percentage discount. For example, suppose a retail business sells an item that is regularly priced at \$59.95, and is planning to have a sale where the item's price will be reduced by 20 percent. You have been asked to write a program to calculate the sale price of the item.

To determine the sale price, you perform two calculations:

- First, you get the amount of the discount, which is 20 percent of the item's regular price.
- Second, you subtract the discount amount from the item's regular price. This gives you the sale price.

Program 2-23 shows how this is done in C++.

Program 2-23

```
1 // This program calculates the sale price of an item
2 // that is regularly priced at $59.95, with a 20 percent
3 // discount subtracted.
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     // Variables to hold the regular price, the
10    // amount of a discount, and the sale price.
11    double regularPrice = 59.95, discount, salePrice;
12
13    // Calculate the amount of a 20% discount.
14    discount = regularPrice * 0.2;
15
16    // Calculate the sale price by subtracting the
17    // discount from the regular price.
18    salePrice = regularPrice - discount;
19
20    // Display the results.
21    cout << "Regular price: $" << regularPrice << endl;
22    cout << "Discount amount: $" << discount << endl;
23    cout << "Sale price: $" << salePrice << endl;
24
25 }
```

Program Output

```
Regular price: $59.95
Discount amount: $11.99
Sale price: $47.96
```

Line 11 defines three variables. The `regularPrice` variable holds the item's regular price, and is initialized with the value 59.95. The `discount` variable will hold the amount of the discount once it is calculated. The `salePrice` variable will hold the item's sale price.

Line 14 calculates the amount of the 20 percent discount by multiplying `regularPrice` by 0.2. The result is stored in the `discount` variable. Line 18 calculates the sale price by subtracting `discount` from `regularPrice`. The result is stored in the `salePrice` variable. The `cout` statements in lines 21 through 23 display the item's regular price, the amount of the discount, and the sale price.



In the Spotlight:

Using the Modulus Operator and Integer Division

The modulus operator (%) is surprisingly useful. For example, suppose you need to extract the rightmost digit of a number. If you divide the number by 10, the remainder will be the rightmost digit. For instance, $123 \div 10 = 12$ with a remainder of 3. In a computer program, you would use the modulus operator to perform this operation. Recall that the modulus operator divides an integer by another integer, and gives the remainder. This is demonstrated in Program 2-24. The program extracts the rightmost digit of the number 12345.

Program 2-24

```
1 // This program extracts the rightmost digit of a number.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int number = 12345;
8     int rightMost = number % 10;
9
10    cout << "The rightmost digit in "
11        << number << " is "
12        << rightMost << endl;
13
14    return 0;
15 }
```

Program Output

The rightmost digit in 12345 is 5

Interestingly, the expression `number % 100` will give you the rightmost two digits in `number`, the expression `number % 1000` will give you the rightmost three digits in `number`, and so on.

The modulus operator (%) is useful in many other situations. For example, Program 2-25 converts 125 seconds to an equivalent number of minutes, and seconds.

Program 2-25

```
1 // This program converts seconds to minutes and seconds.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     // The total seconds is 125.
```

(program continues)

Program 2-25

(continued)

```
8     int totalSeconds = 125;
9
10    // Variables for the minutes and seconds
11    int minutes, seconds;
12
13    // Get the number of minutes.
14    minutes = totalSeconds / 60;
15
16    // Get the remaining seconds.
17    seconds = totalSeconds % 60;
18
19    // Display the results.
20    cout << totalSeconds << " seconds is equivalent to:\n";
21    cout << "Minutes: " << minutes << endl;
22    cout << "Seconds: " << seconds << endl;
23
24 }
```

Program Output

```
125 seconds is equivalent to:
Minutes: 2
Seconds: 5
```

Let's take a closer look at the code:

- Line 8 defines an `int` variable named `totalSeconds`, initialized with the value 125.
- Line 11 declares the `int` variables `minutes` and `seconds`.
- Line 14 calculates the number of minutes in the specified number of seconds. There are 60 seconds in a minute, so this statement divides `totalSeconds` by 60. Notice we are performing integer division in this statement. Both `totalSeconds` and the numeric literal 60 are integers, so the division operator will return an integer result. This is intentional because we want the number of minutes with no fractional part.
- Line 17 calculates the number of remaining seconds. There are 60 seconds in a minute, so this statement uses the `%` operator to divide the `totalSeconds` by 60, and get the remainder of the division. The result is the number of remaining seconds.
- Lines 20 through 22 display the number of minutes and seconds.

**Checkpoint**

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

```
72 = amount;
```

2.22 How would you consolidate the following definitions into one statement?

```
int x = 7;
int y = 16;
int z = 28;
```

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

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

int main()
{
    number = 62.7;
    double number;
    cout << number << endl;
    return 0;
}
```

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

```
portion = 70 / 3;
```

2.15 Comments

CONCEPT: Comments are notes of explanation that document lines or sections of a program. Comments are part of the program, but the compiler ignores them. They are intended for people who may be reading the source code.

It may surprise you that one of the most important parts of a program has absolutely no impact on the way it runs. In fact, the compiler ignores this part of a program. Of course, I'm speaking of the comments.

As a beginning programmer, you might be resistant to the idea of liberally writing comments in your programs. After all, it can seem more productive to write code that actually does 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 open the file that contains your source code and stare at thousands of statements that now make no sense at all. If only you had left some notes to yourself explaining the program's code. Of course 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 code will make your life easier, not to mention the other programmers who may have to read your code in the future.

Single-Line Comments

You have already seen one way to place comments in a C++ program. 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. Program 2-26 shows that comments may be placed liberally throughout a program.

Program 2-26

```
1 // PROGRAM: PAYROLL.CPP
2 // Written by Herbert Dorfmann
3 // This program calculates company payroll
4 // Last modification: 8/20/2017
5 #include <iostream>
6 using namespace std;
7
8 int main()
9 {
10    double payRate;    // Holds the hourly pay rate
11    double hours;      // Holds the hours worked
12    int employNumber;  // Holds the employee number
```

(The remainder of this program is left out.)

In addition to telling who wrote the program and describing the purpose of variables, comments can also be used to explain complex procedures in your code.

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 2-27 illustrates how multi-line comments may be used. Notice a multi-line comment starts in line 1 with the /* symbol, and it ends in line 6 with the */ symbol.

Program 2-27

```
1 /*
2  PROGRAM: PAYROLL.CPP
3  Written by Herbert Dorfmann
4  This program calculates company payroll
5  Last modification: 8/20/2017
6 */
7
8 #include <iostream>
9 using namespace std;
10
11 int main()
12 {
13    double payRate;    // Holds the hourly pay rate
14    double hours;      // Holds the hours worked
15    int employNumber;  // Holds the employee number
```

(The remainder of this program is left out.)

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. Consequently, 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 in their programs. Convenience usually dictates which style to use.

Remember the following advice 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.

2.16 Named Constants

CONCEPT: Literals may be given names that symbolically represent them in a program.

Assume the following statement appears in a banking program that calculates data pertaining to loans:

```
amount = balance * 0.069;
```

In such a program, two potential problems arise. First, it is not clear to anyone other than the original programmer what 0.069 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 6.9 percent to 7.2 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* 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 = 0.069;
```

It looks just like a regular variable definition except that the word `const` appears before the data type name, and the name of the variable is written in all uppercase characters. The key word `const` is a qualifier that tells the compiler to make the variable read-only. Its value will remain constant throughout the program's execution. It is not required that the variable name be written in all uppercase characters, but many programmers prefer to write them this way so that they are easily distinguishable from regular variable names.

An initialization value must be given when defining a constant with the `const` qualifier, or an error will result when the program is compiled. A compiler error will also result if there are any statements in the program that attempt to change the value of a named constant.

An advantage of using named constants is that they make programs more self-documenting. The statement

```
amount = balance * 0.069;
```

can be changed to read

```
amount = balance * INTEREST_RATE;
```

A new programmer can read the second statement and know 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. Let's say the interest rate appears in a dozen different statements throughout the program. When the rate changes, the initialization value in the definition of the named constant is the only value that needs to be modified. If the rate increases to 7.2 percent, the definition is changed to the following:

```
const double INTEREST_RATE = 0.072;
```

The program is then ready to be recompiled. Every statement that uses `INTEREST_RATE` will then use the new value.

Named constants can also help prevent typographical errors in a program's code. For example, suppose you use the number 3.14159 as the value of *pi* in a program that performs various geometric calculations. Each time you type the number 3.14159 in the program's code, there is a chance that you will make a mistake with one or more of the digits. As a result, the program will not produce the correct results. To help prevent a mistake such as this, you can define a named constant for *pi*, initialized with the correct value, then use that constant in all of the formulas that require its value. Program 2-28 shows an example. It calculates the circumference of a circle that has a diameter of 10.

Program 2-28

```
1 // This program calculates the circumference of a circle.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     // Constants
8     const double PI = 3.14159;
9     const double DIAMETER = 10.0;
10
11    // Variable to hold the circumference
12    double circumference;
13
14    // Calculate the circumference.
15    circumference = PI * DIAMETER;
16
17    // Display the circumference.
18    cout << "The circumference is: " << circumference << endl;
19
20 }
```

Program Output

The circumference is: 31.4159

Let's take a closer look at the program. Line 8 defines a constant `double` named `PI`, initialized with the value 3.14159. This constant will be used for the value of `pi` in the program's calculation. Line 9 defines a constant `double` named `DIAMETER`, initialized with the value 10. This will be used for the circle's diameter. Line 12 defines a `double` variable named `circumference`, which will be used to hold the circle's circumference. Line 15 calculates the circle's circumference by multiplying `PI` by `DIAMETER`. The result of the calculation is assigned to the `circumference` variable. Line 18 displays the circle's circumference.



Checkpoint

- 2.25 Write statements using the `const` qualifier to create named constants for the following literal values:

Literal 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 ft/s^2)
9.8	The gravitational acceleration constant (in m/s^2)
1609	Number of meters in a mile

2.17

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, generate object code.

When the compiler reads a program, it processes it as one long stream of characters. The compiler doesn't care that each statement is on a separate line, or that 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 2-29 for example.

Program 2-29

```
1 #include <iostream>
2 using namespace std; int main(){double shares=220.0;
3 double avgPrice=14.67; cout<<"There were "<<shares
4 <<" shares sold at $"<<avgPrice<<" per share.\n";
5 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 very difficult to read. The same program is shown in Program 2-30, written in a more reasonable style.

Program 2-30

```
1 // This example is much more readable than Program 2-29.  
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 indentations in a program so visual cues are created. These cues quickly tell a programmer important information about a program.

For example, notice in Program 2-30 that inside the function `main`'s 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.

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 five lines:

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

This statement will work just as if it were typed on one line. Here is an example of variable definitions treated similarly:

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

There are many other issues related to programming style. They will be presented throughout the book.

Review Questions and Exercises

Short Answer

1. How many operands does each of the following types of operators require?
 Unary
 Binary
 Ternary
2. How may the `double` variables `temp`, `weight`, and `age` be defined in one statement?
3. How may the `int` variables `months`, `days`, and `years` be defined in one statement, with `months` initialized to 2 and `years` initialized to 3?
4. Write assignment statements that perform the following operations with the variables `a`, `b`, and `c`:
 - A) Adds 2 to `a` and stores the result in `b`.
 - B) Multiplies `b` by 4 and stores the result in `a`.
 - C) Divides `a` by 3.14 and stores the result in `b`.
 - D) Subtracts 8 from `b` and stores the result in `a`.
 - E) Stores the value 27 in `a`.
 - F) Stores the character 'K' in `c`.
 - G) Stores the ASCII code for 'B' in `c`.
5. Is the following comment written using single-line or multi-line comment symbols?
`/* This program was written by M. A. Codewriter */`
6. Is the following comment written using single-line or multi-line comment symbols?
`// This program was written by M. A. Codewriter`
7. Modify the following program so it prints two blank lines between each line of text.

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Two mandolins like creatures in the";
    cout << "dark";
    cout << "Creating the agony of ecstasy.";
    cout << "                                - George Barker";
    return 0;
}
```
8. 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\nnameze\n";
    cout << "you.";
    return 0;
}

D) #include <iostream>
using namespace std;

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

E) #include <iostream>
using namespace std;

int main()
{
    int a, x = 23;
    a = x % 2;
    cout << x << endl << a << endl;
    return 0;
}
```

Multiple Choice

9. Every complete statement ends with a(n) _____.
A) period
B) # symbol
C) semicolon
D) ending brace
10. Which of the following statements is correct?
A) #include (iostream)
B) #include {iostream}
C) #include <iostream>
D) #include [iostream]
E) All of the above
11. Every C++ program must have a _____.
A) cout statement
B) function main
C) #include statement
D) All of the above
12. Preprocessor directives begin with _____.
A) #
B) !
C) <
D) *
E) None of the above
13. The following data
72
'A'
"Hello World"
2.8712
are all examples of _____.
A) variables
B) literals or constants
C) strings
D) none of the above
14. A group of statements, such as the contents of a function, is enclosed in _____.
A) braces {}
B) parentheses ()
C) brackets <>
D) all of the above will do
15. Which of the following are *not* valid assignment statements? (Select all that apply.)
A) total = 9;
B) 72 = amount;
C) profit = 129
D) letter = 'W';

16. Which of the following are *not* valid cout statements? (Select all that apply.)
- A) `cout << "Hello World";`
 - B) `cout << "Have a nice day"\n;`
 - C) `cout < value;`
 - D) `cout << Programming is great fun;`
17. Assume `w = 5`, `x = 4`, `y = 8`, and `z = 2`. What value will be stored in `result` in each of the following statements?
- A) `result = x + y;`
 - B) `result = z * 2;`
 - C) `result = y / x;`
 - D) `result = y - z;`
 - E) `result = w % 2;`
18. How would each of the following numbers be represented in E notation?
- A) 3.287×10^6
 - B) -978.65×10^{12}
 - C) 7.65491×10^{-3}
 - D) -58710.23×10^{-4}
19. The negation operator is _____.
- A) unary
 - B) binary
 - C) ternary
 - D) none of the above
20. A(n) _____ is like a variable, but its value is read-only and cannot be changed during the program's execution.
- A) secure variable
 - B) uninitialized variable
 - C) named constant
 - D) locked variable
21. When do preprocessor directives execute?
- A) Before the compiler compiles your program
 - B) After the compiler compiles your program
 - C) At the same time as the compiler compiles your program
 - D) None of the above

True or False

22. T F A variable must be defined before it can be used.
23. T F Variable names may begin with a number.
24. T F Variable names may be up to 31 characters long.
25. T F A left brace in a C++ program should always be followed by a right brace later in the program.
26. T F You cannot initialize a named constant that is declared with the `const` modifier.

Algorithm Workbench

27. Convert the following *pseudocode* to C++ code. Be sure to define the appropriate variables.

 Store 20 in the *speed* variable.
 Store 10 in the *time* variable.
 Multiply *speed* by *time* and store the result in the *distance* variable.
 Display the contents of the *distance* variable.

28. Convert the following *pseudocode* to C++ code. Be sure to define the appropriate variables.

 Store 172.5 in the *force* variable.
 Store 27.5 in the *area* variable.
 Divide *area* by *force* and store the result in the *pressure* variable.
 Display the contents of the *pressure* variable.

Find the Error

29. 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;
}
```

Programming Challenges

Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.

1. Sum of Two Numbers

Write a program that stores the integers 50 and 100 in variables, and stores the sum of these two in a variable named *total*.

2. Sales Prediction

The East Coast sales division of a company generates 58 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 \$8.6 million in sales this year.

3. Sales Tax

Write a program that will compute the total sales tax on a \$95 purchase. Assume the state sales tax is 4 percent, and the county sales tax is 2 percent.

4. Restaurant Bill

Write a program that computes the tax and tip on a restaurant bill for a patron with a \$88.67 meal charge. The tax should be 6.75 percent of the meal cost. The tip should

be 20 percent of the total after adding the tax. Display the meal cost, tax amount, tip amount, and total bill on the screen.

5. Average of Values

To get the average of a series of values, you add the values up then divide the sum by the number of values. Write a program that stores the following values in five different variables: 28, 32, 37, 24, and 33. The program should first calculate the sum of these five variables and store the result in a separate variable named `sum`. Then, the program should divide the `sum` variable by 5 to get the average. Display the average on the screen.



TIP: Use the `double` data type for all variables in this program.

6. Annual Pay

Suppose an employee gets paid every two weeks and earns \$2,200 each pay period. In a year, the employee gets paid 26 times. Write a program that defines the following variables:

<code>payAmount</code>	This variable will hold the amount of pay the employee earns each pay period. Initialize the variable with 2200.0.
<code>payPeriods</code>	This variable will hold the number of pay periods in a year. Initialize the variable with 26.
<code>annualPay</code>	This variable will hold the employee's total annual pay, which will be calculated.

The program should calculate the employee's total annual pay by multiplying the employee's pay amount by the number of pay periods in a year and store the result in the `annualPay` variable. Display the total annual pay on the screen.

7. Ocean Levels

Assuming the ocean's level is currently rising at about 1.5 millimeters per year, write a program that displays:

- The number of millimeters higher than the current level that the ocean's level will be in 5 years.
- The number of millimeters higher than the current level that the ocean's level will be in 7 years.
- The number of millimeters higher than the current level that the ocean's level will be in 10 years.

8. Total Purchase

A customer in a store is purchasing five items. The prices of the five items are as follows:

Price of item 1 = \$15.95
Price of item 2 = \$24.95
Price of item 3 = \$6.95
Price of item 4 = \$12.95
Price of item 5 = \$3.95

Write a program that holds the prices of the five items in five variables. Display each item's price, the subtotal of the sale, the amount of sales tax, and the total. Assume the sales tax is 7 percent.

9. 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 technical documentation, so you can't look this information up. Write a C++ program that will determine the amount of memory used by these types and display the information on the screen.

10. Miles per Gallon

A car holds 15 gallons of gasoline and can travel 375 miles before refueling. Write a program that calculates the number of miles per gallon the car gets. Display the result on the screen.

Hint: Use the following formula to calculate miles per gallon (MPG):

$$\text{MPG} = \text{Miles Driven}/\text{Gallons of Gas Used}$$

11. Distance per Tank of Gas

A car with a 20-gallon gas tank averages 23.5 miles per gallon when driven in town, and 28.9 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.

Hint: The following formula can be used to calculate the distance:

$$\text{Distance} = \text{Number of Gallons} \times \text{Average Miles per Gallon}$$

12. Land Calculation

One acre of land is equivalent to 43,560 square feet. Write a program that calculates the number of acres in a tract of land with 391,876 square feet.

13. Circuit Board Price

An electronics company sells circuit boards at a 35 percent profit. Write a program that will calculate the selling price of a circuit board that costs \$14.95. Display the result on the screen.

14. Personal Information

Write a program that displays the following pieces of 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.

15. Triangle Pattern

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

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

16. Diamond Pattern

Write a program that displays the following pattern:

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

17. Stock Commission

Kathryn bought 750 shares of stock at a price of \$35.00 per share. She must pay her stockbroker a 2 percent commission for the transaction. Write a program that calculates and displays the following:

- The amount paid for the stock alone (without the commission).
- The amount of the commission.
- The total amount paid (for the stock plus the commission).

18. Energy Drink Consumption

A soft drink company recently surveyed 16,500 of its customers and found that approximately 15 percent of those surveyed purchase one or more energy drinks per week. Of those customers who purchase energy drinks, approximately 58 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.

19. Annual High Temperatures

The average July high temperature is 85 degrees Fahrenheit in New York City, 88 degrees Fahrenheit in Denver, and 106 degrees Fahrenheit in Phoenix. Write a program that calculates and reports what the new average July high temperature would be for each of these cities if temperatures rise by 2 percent.

20. How Much Paint

A particular brand of paint covers 340 square feet per gallon. Write a program to determine and report approximately how many gallons of paint will be needed to paint two coats on a wooden fence that is 6 feet high and 100 feet long.

Expressions and Interactivity

TOPICS

- | | |
|--|---|
| 3.1 The <code>cin</code> Object | 3.7 Formatting Output |
| 3.2 Mathematical Expressions | 3.8 Working with Characters and <code>string</code> Objects |
| 3.3 When You Mix Apples and Oranges: Type Conversion | 3.9 More Mathematical Library Functions |
| 3.4 Overflow and Underflow | 3.10 Focus on Debugging: Hand Tracing a Program |
| 3.5 Type Casting | 3.11 Focus on Problem Solving: A Case Study |
| 3.6 Multiple Assignment and Combined Assignment | |

3.1

The `cin` Object

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

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

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 data. For example, a program that calculates payroll for a small business might ask the user to enter the name of the employee, the hours worked, and the hourly pay rate. When the paycheck for that employee has been printed, the program could start over again and ask for the name, hours worked, and hourly pay rate of the next employee.

Just as `cout` is C++'s standard output object, `cin` is the standard input object. It reads input from the console (or keyboard) as shown in Program 3-1.



VideoNote
Reading Input
with `cin`

Program 3-1

```

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

Program Output with Example Input Shown in Bold

This program calculates the area of a rectangle.

What is the length of the rectangle? **10**

What is the width of the rectangle? **20**

The area of the rectangle is 200.

Instead of calculating the area of one rectangle, this program can be used to get 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 13 and 14:

```

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

In line 13, the `cout` object is used to display the question “What is the length of the rectangle?” This question is known as a *prompt*, and it tells the user what data he or she should enter. Your program should always display a prompt before it uses `cin` to read input. This way, the user will know that he or she must type a value at the keyboard.

Line 14 uses the `cin` object 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 line, characters are taken from the `cin` object (which gets them from the keyboard) and are stored in the `length` variable.

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

1. Use the `cout` object to display a prompt on the screen.
2. Use the `cin` object 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?

When the user sees this prompt, he or she knows to enter the rectangle's length. After the prompt is displayed, the program uses the `cin` object to read a value from the keyboard and store the value in the `length` variable.

Notice the `<<` and `>>` operators appear to point in the direction that data is flowing. In a statement that uses the `cout` object, the `<<` operator always points toward `cout`. This indicates that data is flowing from a variable or a literal to the `cout` object. In a statement that uses the `cin` object, the `>>` operator always points toward the variable that is receiving the value. This indicates that data is flowing from `cin` to a variable. This is illustrated in Figure 3-1.

Figure 3-1 The `<<` and `>>` operators

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

Think of the `<<` and `>>` operators as arrows that point in
the direction that data is flowing.

```
cout ← "What is the length of the rectangle? ";
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 in the program will be executed until `cin` gets its input.

`cin` automatically converts the data read from the keyboard to the data type of the variable used to store it. If the user types 10, it is read as the characters '1' and '0'. `cin` is smart enough to know this will have to be converted to an `int` value before it is stored in the `length` variable. `cin` is also smart enough to know a value like 10.7 cannot be stored in an integer variable. If the user enters a floating-point value for an integer variable, `cin` will not read the part of the number after the decimal point.



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

Entering Multiple Values

The `cin` object may be used to gather multiple values at once. Look at Program 3-2, which is a modified version of Program 3-1.

Line 15 waits for the user to enter two values. The first is assigned to `length` and the second to `width`.

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

Program 3-2

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

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

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 3-3.

Program 3-3

```
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    cout << "Whole: " << whole << endl;
15    cout << "Fractional: " << fractional << endl;
16    cout << "Letter: " << letter << endl;
17
18    return 0;
19 }
```

Program Output with Example Input Shown in Bold

Enter an integer, a double, and a character: **4 5.7 b**

Whole: 4

Fractional: 5.7

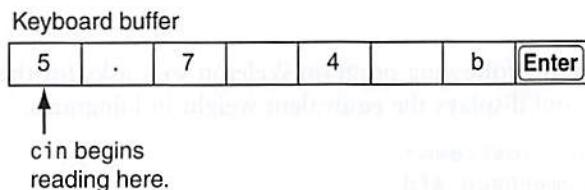
Letter: b

As you can see in the example output, the values are stored in their respective variables. But what if the user had responded in the following way?

Enter an integer, a double, and a character: **5.7 4 b**

When the user types values at the keyboard, those values are first stored in an area of memory known as the *keyboard buffer*. So, when the user enters the values 5.7, 4, and b, they are stored in the keyboard buffer as shown in Figure 3-2.

Figure 3-2 The keyboard buffer



When the user presses the key, `cin` reads the value 5 into the variable `whole`. It does not read the decimal point because `whole` is an integer variable. Next it reads .7 and stores that value in the `double` variable `fractional`. The space is skipped, and 4 is the next value read. It is stored as a character in the variable `letter`. Because this `cin` statement reads only three values, the b is left in the keyboard buffer. So, in this situation the program would have stored 5 in `whole`, 0.7 in `fractional`, and the character '4' in `letter`. It is important that the user enters values in the correct order.



Checkpoint

- 3.1 What header file must be included in programs using `cin`?
- 3.2 True or false: `cin` requires the user to press the key when finished entering data.

- 3.3 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;
A) 3.14
B) 3
C) 0
D) Nothing. An error message is displayed.
```

- 3.4 A program has the following variable definitions.

```
long miles;
int feet;
float inches;
```

Write one **cin** statement that reads a value into each of these variables.

- 3.5 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;
}
```

- 3.6 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 code here that prompts the user
    // to enter his or her weight and reads
    // the input into the pounds variable.

    // 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.

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:

```
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.

Program 3-4 shows how mathematical expressions can be used with the `cout` object.

Program 3-4

```
1 // This program asks the user to enter the numerator  
2 // and denominator of a fraction and it displays the  
3 // decimal value.  
4  
5 #include <iostream>  
6 using namespace std;  
7  
8 int main()  
9 {  
10     double numerator, denominator;  
11  
12     cout << "This program shows the decimal value of ";  
13     cout << "a fraction.\n";
```

(program continues)

Program 3-4

(continued)

```

14     cout << "Enter the numerator: ";
15     cin >> numerator;
16     cout << "Enter the denominator: ";
17     cin >> denominator;
18     cout << "The decimal value is ";
19     cout << (numerator / denominator) << endl;
20
21 }
```

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 will display the value of any legal expression in C++. In Program 3-4, the value of the expression numerator / denominator is displayed.



NOTE: The example input for Program 3-4 shows the user entering 3 and 16. Since these values are assigned to double variables, they are stored as the double values 3.0 and 16.0.



NOTE: When sending an expression that consists of an operator to cout, it is always a good idea to put parentheses around the expression. Some advanced 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? 6 is used as an operand for both the addition and division operators. outcome 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.

Mathematical expressions are evaluated from left to right. When two operators share an operand, the operator with the highest precedence works first. Multiplication and division have higher precedence than addition and subtraction, so the statement above works like this:

- 6 is divided by 3, yielding a result of 2
- 12 is added to 2, yielding a result of 14

It could be diagrammed in the following way:

```

outcome = 12 + 6 / 3
         \ /
outcome = 12 +   2
outcome = 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 them.

Table 3-1 Precedence of Arithmetic Operators (Highest to Lowest)

(unary negation) -

* / %

+ -

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 Simple Expressions and Their Values

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

An operator's *associativity* is either left to right, or right to left. If two operators sharing an operand have the same precedence, they work according to their associativity. Table 3-3 lists the associativity of the arithmetic operators. As an example, look at the following expression:

5 - 3 + 2

Both the - and + operators in this expression have the same precedence, and they have left to right associativity. So, the operators will work from left to right. This expression is the same as:

((5 - 3) + 2)

Here is another example:

12 / 6 * 4

Because the / and * operators have the same precedence, and they have left to right associativity, they will work from left to right. This expression is the same as:

((12 / 6) * 4)

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 $a + b$ is divided by 4.

```
result = (a + b) / 4;
```

Without the parentheses, however, b would be divided by 4 and the result added to a . Table 3-4 shows more expressions and their values.

Table 3-4 More Simple Expressions and Their Values

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
$y = 3\frac{x}{2}$	$y = x / 2 * 3;$
$z = 3bc + 4$	$z = 3 * b * c + 4;$
$a = \frac{3x + 2}{4a - 1}$	$a = (3 * x + 2) / (4 * a - 1)$

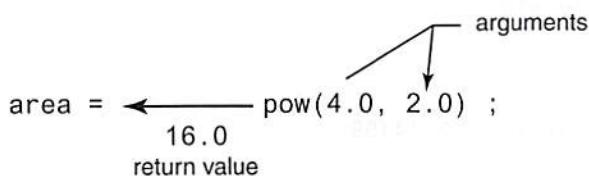
No Exponents Please!

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, 2.0);
```

This statement contains a *call* to the `pow` function. The numbers inside the parentheses are *arguments*. Arguments are data being sent to the function. The `pow` function always raises the first argument to the power of the second argument. In this example, 4 is raised to the power of 2. The result is *returned* from the function and used in the statement where the function call appears. In this case, the value 16 is returned from `pow` and assigned to the variable `area`. This is illustrated in Figure 3-3.

Figure 3-3 The `pow` function



The statement `area = pow(4.0, 2.0)` is equivalent to the following algebraic statement:

```
area = 42
```

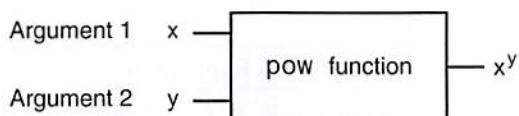
Here is another example of a statement using the `pow` function. It assigns 3 times 6^3 to `x`:

```
x = 3 * pow(6.0, 3.0);
```

And the following statement displays the value of 5 raised to the power of 4:

```
cout << pow(5.0, 4.0);
```

It might be helpful to think of `pow` as a “black box” that you plug two numbers into, and that 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 3-4:

Figure 3-4 The pow function as a “black box”

There are some guidelines that should be followed when the pow function is used. First, the program must include the `<cmath>` header file. Second, the arguments that you pass to the pow function should be `doubles`. Third, the variable used to store pow's return value should be defined as a `double`. For example, in the following statement the variable `area` should be a `double`:

```
area = pow(4.0, 2.0);
```

Program 3-5 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 = PI * pow(radius, 2.0);
```

Program 3-5

```

1 // This program calculates the area of a circle.
2 // The formula for the area of a circle is Pi times
3 // the radius squared. Pi is 3.14159.
4 #include <iostream>
5 #include <cmath> // needed for pow function
6 using namespace std;
7
8 int main()
9 {
10     const double PI = 3.14159;
11     double area, radius;
12
13     cout << "This program calculates the area of a circle.\n";
14     cout << "What is the radius of the circle? ";
15     cin >> radius;
16     area = PI * pow(radius, 2.0);
17     cout << "The area is " << area << endl;
18     return 0;
19 }
```

Program Output with Example Input Shown in Bold

This program calculates the area of a circle.

What is the radius of the circle? **10**

The area is 314.159



NOTE: Program 3-5 is presented as a demonstration of the pow function. In reality, there is no reason to use the pow function in such a simple operation. The math statement could just as easily be written as

```
area = PI * radius * radius;
```

The pow function is useful, however, in operations that involve larger exponents.



In the Spotlight:

Calculating an Average

Determining the average of a group of values is a simple calculation: You add all of the values, then divide the sum by the number of values. Although this is a straightforward calculation, it is easy to make a mistake when writing a program that calculates an average. For example, let's assume that `a`, `b`, and `c` are double variables. Each of the variables holds a value, and we want to calculate the average of those values. If we are careless, we might write a statement such as the following to perform the calculation:

```
average = a + b + c / 3.0;
```

Can you see the error in this statement? When it executes, the division will take place first. The value in `c` will be divided by 3.0, then the result will be added to the sum of `a` + `b`. That is not the correct way to calculate an average. To correct this error, we need to put parentheses around `a` + `b` + `c`, as shown here:

```
average = (a + b + c) / 3.0;
```

Let's step through the process of writing a program that calculates an average. Suppose you have taken three tests in your computer science class, and you want to write a program that will display the average of the test scores. Here is the algorithm in pseudocode:

Get the first test score.

Get the second test score.

Get the third test score.

Calculate the average by adding the three test scores and dividing the sum by 3.

Display the average.

In the first three steps, we prompt the user to enter three test scores. Let's say we store those test scores in the double variables `test1`, `test2`, and `test3`. Then in the fourth step, we calculate the average of the three test scores. We will use the following statement to perform the calculation and store the result in the `average` variable, which is a `double`:

```
average = (test1 + test2 + test3) / 3.0;
```

The last step is to display the average. Program 3-6 shows the program.

Program 3-6

```

1 // This program calculates the average
2 // of three test scores.
3 #include <iostream>
4 #include <cmath>
5 using namespace std;
6
7 int main()
8 {
9     double test1, test2, test3; // To hold the scores
10    double average;           // To hold the average
11
12    // Get the three test scores.
13    cout << "Enter the first test score: ";
14    cin >> test1;
15    cout << "Enter the second test score: ";
16    cin >> test2;
17    cout << "Enter the third test score: ";
18    cin >> test3;
19
20    // Calculate the average of the scores.
21    average = (test1 + test2 + test3) / 3.0;
22
23    // Display the average.
24    cout << "The average score is: " << average << endl;
25
26 }

```

Program Output with Example Input Shown in Bold

Enter the first test score: **90**

Enter the second test score: **80**

Enter the third test score: **100**

The average score is 90

**Checkpoint**

- 3.7 Complete the table below by determining the value of each expression.

Expression	Value
$6 + 3 * 5$	
$12 / 2 - 4$	
$9 + 14 * 2 - 6$	
$5 + 19 \% 3 - 1$	
$(6 + 2) * 3$	
$14 / (11 - 4)$	
$9 + 12 * (8 - 3)$	
$(6 + 17) \% 2 - 1$	
$(9 - 3) * (6 + 9) / 3$	

- 3.8 Write C++ expressions for the following algebraic expressions:

$$y = 6x$$

$$a = 2b + 4c$$

$$y = x^2$$

$$g = \frac{x + 2}{z^2}$$

$$y = \frac{x^2}{z^2}$$

- 3.9 Study the following program and complete the table.

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

int main()
{
    double value1, value2, value3;

    cout << "Enter a number: ";
    cin >> value1;
    value2 = 2 * pow(value1, 2.0);
    value3 = 3 + value2 / 2 - 1;
    cout << value3 << endl;
    return 0;
}
```

If the User Enters...	The Program Will Display What Number (Stored in <code>value3</code>)?
2	
5	
4.3	
6	

- 3.10 Complete the following program skeleton so it displays the volume of a cylindrical fuel tank. The formula for the volume of a cylinder is

$$\text{Volume} = \pi r^2 h$$

where

π is 3.14159,

r is the radius of the tank, and

h is the height of the tank.

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

```

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.
}

```

3.3

When You Mix Apples and Oranges: 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 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`. Table 3-7 lists the data types in order of their rank, from highest to lowest.

Table 3-7 Data Type Ranking

<code>long double</code>
<code>double</code>
<code>float</code>
<code>unsigned long long int</code>
<code>long long int</code>
<code>unsigned long int</code>
<code>long int</code>
<code>unsigned int</code>
<code>int</code>

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 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: `chars`, `shorts`, and `unsigned shorts` are automatically promoted to `int`.

You will notice that `char`, `short`, and `unsigned short` do not appear in Table 3-7. That's because anytime they 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 `shorts` are the same size as `ints`. 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 `float`:

```
years * interestRate
```

Before the multiplication takes place, `years` will be promoted to a `float`.

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.

In the following statement, assume that `area` is a `long int`, while `length` and `width` are both `ints`:

```
area = length * width;
```

Since `length` and `width` are both `ints`, they will not be converted to any other data type. The result of the multiplication, however, will be converted to `long` so that it can be stored in `area`.

Watch out for situations where an expression results in a fractional value being assigned to an integer variable. Here is an example:

```
int x, y = 4;  
float z = 2.7;  
x = y * z;
```

In the expression `y * z`, `y` will be promoted to `float` and 10.8 will result from the multiplication. Since `x` is an integer, however, 10.8 will be truncated and 10 will be stored in `x`.

Integer Division

When you divide an integer by another integer in C++, the result is always an integer. If there is a remainder, it will be discarded. For example, in the following code, `parts` is assigned the value 2.0:

```
double parts;  
parts = 15 / 6;
```

Even though 15 divided by 6 is really 2.5, the .5 part of the result is discarded because we are dividing an integer by an integer. It doesn't matter that `parts` is declared as a `double` because the fractional part of the result is discarded *before* the assignment takes place. In order for a division operation to return a floating-point value, at least one of the operands must be of a floating-point data type. For example, the previous code could be written as:

```
double parts;  
parts = 15.0 / 6;
```

In this code, the literal value 15.0 is interpreted as a floating-point number, so the division operation will return a floating-point number. The value 2.5 will be assigned to `parts`.

3.4

Overflow and Underflow

CONCEPT: When a variable is assigned a value that is too large or too small in range for that variable's data type, the variable overflows or underflows.

Trouble can arise when a variable is being assigned a value that is too large for its type. Here is a statement where `a`, `b`, and `c` are all short integers:

```
a = b * c;
```

If `b` and `c` are set to large enough values, the multiplication will produce a number too big to be stored in `a`. To prepare for this, `a` should have been defined as an `int`, or a `long int`.

When a variable is assigned a number that is too large for its data type, it *overflows*. Likewise, assigning a value that is too small for a variable causes it to *underflow*. Program 3-7 shows what happens when an integer overflows or underflows. (The output shown is from a system with two-byte short integers.)

Program 3-7

```

1 // This program demonstrates integer overflow and underflow.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     // testVar is initialized with the maximum value for a short.
8     short testVar = 32767;
9
10    // Display testVar.
11    cout << testVar << endl;
12
13    // Add 1 to testVar to make it overflow.
14    testVar = testVar + 1;
15    cout << testVar << endl;
16
17    // Subtract 1 from testVar to make it underflow.
18    testVar = testVar - 1;
19    cout << testVar << endl;
20
21 }
```

Program Output

```
32767
-32768
32767
```

Typically, when an integer overflows, its contents wrap around to that data type's lowest possible value. In Program 3-7, `testVar` wrapped around from 32,767 to -32,768 when 1 was added to it. When 1 was subtracted from `testVar`, it underflowed, which caused its

contents to wrap back around to 32,767. No warning or error message is given, so be careful when working with numbers close to the maximum or minimum range of an integer. If an overflow or underflow occurs, the program will use the incorrect number, and therefore produce incorrect results.

When floating-point variables overflow or underflow, the results depend upon how the compiler is configured. Your system may produce programs that do any of the following:

- Produce an incorrect result and continue running.
- Print an error message and immediately stop when either floating-point overflow or underflow occurs.
- Print an error message and immediately stop when floating-point overflow occurs, but store a 0 in the variable when it underflows.
- Give you a choice of behaviors when overflow or underflow occurs.

You can find out how your system reacts by compiling and running Program 3-8.

Program 3-8

```
1 // This program can be used to see how your system handles
2 // floating-point overflow and underflow.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     float test;
9
10    test = 2.0e38 * 1000;      // Should overflow test.
11    cout << test << endl;
12    test = 2.0e-38 / 2.0e38;  // Should underflow test.
13    cout << test << endl;
14
15 }
```

3.5

Type Casting

CONCEPT: Type casting allows you to perform manual 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 to which you wish to convert *Value*. 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` is converted to an `int`, the fractional part is truncated, so this statement stores 3 in `val`. The original value in `number` is not changed, however.

Type cast expressions are useful in situations where C++ will not perform the desired conversion automatically. Program 3-9 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

```
perMonth = static_cast<double>(books) / months;
```

Program 3-9

```
1 // This program uses a type cast to avoid integer division.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int books;           // Number of books to read
8     int months;          // Number of months spent reading
9     double perMonth;    // Average number of books per month
10
11    cout << "How many books do you plan to read? ";
12    cin >> books;
13    cout << "How many months will it take you to read them? ";
14    cin >> months;
15    perMonth = static_cast<double>(books) / months;
16    cout << "That is " << perMonth << " books per month.\n";
17
18 }
```

Program Output with Example Input Shown in Bold

How many books do you plan to read? **30**

How many months will it take you to read them? **7**

That is 4.28571 books per month.

The variable `books` is an integer, but its value is converted to a `double` before the division takes place. Without the type cast expression in line 15, integer division would have been performed, resulting in an incorrect answer.



WARNING! In Program 3-9, the following statement would still have resulted in integer division:

```
perMonth = 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 3-10 further demonstrates the type cast expression.

Program 3-10

```
1 // This program uses a type cast expression to print a character
2 // from a number.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int number = 65;
9
10    // Display the value of the number variable.
11    cout << number << endl;
12
13    // Display the value of number converted to
14    // the char data type.
15    cout << static_cast<char>(number) << endl;
16
17 }
```

Program Output

```
65
A
```

Let's take a closer look at this program. In line 8, the `int` variable `number` is initialized with the value 65. In line 11, `number` is sent to `cout`, causing 65 to be displayed. In line 15, a type cast expression is used to convert the value in `number` to the `char` data type. Recall from Chapter 2 that characters are stored in memory as integer ASCII codes. The number 65 is the ASCII code for the letter 'A', so this statement causes the letter 'A' to be displayed.



NOTE: C++ provides several different type cast expressions. `static_cast` is the most commonly used type cast expression, so we will primarily use it in this book.

**Checkpoint**

3.11 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) $b / \text{static_cast}<\text{double}>(\text{a})$
- F) `static_cast<double>(b) / static_cast<double>(a)`
- G) $b / \text{static_cast}<\text{int}>(x)$
- H) `static_cast<int>(x) * static_cast<int>(z)`
- I) `static_cast<int>(x * z)`
- J) `static_cast<double>(\text{static_cast}<\text{int}>(x) * static_cast<\text{int}>(z))`

- 3.12 Complete the following program skeleton so it asks the user to enter a character. Store the character in the variable `letter`. Use a type cast expression with the variable in a `cout` statement to display the character's ASCII code on the screen.

```
#include <iostream>
using namespace std;
int main()
{
    char letter;

    // Finish this program
    // as specified above.
    return 0;
}
```

- 3.13 What will the following program display?

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

int main()
{
    int integer1, integer2;
    double result;
    integer1 = 19;
    integer2 = 2;
    result = integer1 / integer2;
    cout << result << endl;
    result = static_cast<double>(integer1) / integer2;
    cout << result << endl;
    result = static_cast<double>(integer1 / integer2);
    cout << result << endl;
    return 0;
}
```

3.6

Multiple Assignment 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.*

*The assignment operator works from right to left. 12 is first assigned to `d`, then to `c`, then to `b`, then to `a`.

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 (Assume $x = 6$)

Statement	What It Does	Value of x After the Statement
<code>x = x + 4;</code>	Adds 4 to x	10
<code>x = x - 3;</code>	Subtracts 3 from x	3
<code>x = x * 10;</code>	Multiplies x by 10	60
<code>x = x / 2;</code>	Divides x by 2	3
<code>x = x % 4</code>	Makes x the remainder of $x / 4$	2

These types of operations are very common in programming. For convenience, 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*, and *arithmetic assignment operators*.

Table 3-9 Combined Assignment Operators

Operator	Example Usage	Equivalent to
<code>+=</code>	<code>x += 5;</code>	<code>x = x + 5;</code>
<code>-=</code>	<code>y -= 2;</code>	<code>y = y - 2;</code>
<code>*=</code>	<code>z *= 10;</code>	<code>z = z * 10;</code>
<code>/=</code>	<code>a /= b;</code>	<code>a = a / b;</code>
<code>%=</code>	<code>c %= 3;</code>	<code>c = c % 3;</code>

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 3-11 uses combined assignment operators.

Program 3-11

```
1 // This program tracks the inventory of three widget stores
2 // that opened at the same time. Each store started with the
3 // same number of widgets in inventory. By subtracting the
```

(program continues)

Program 3-11

(continued)

```

4 // number of widgets each store has sold from its inventory,
5 // the current inventory can be calculated.
6 #include <iostream>
7 using namespace std;
8
9 int main()
10 {
11     int begInv,    // Beginning inventory for all stores
12         sold,      // Number of widgets sold
13         store1,    // Store 1's inventory
14         store2,    // Store 2's inventory
15         store3;    // Store 3's inventory
16
17     // Get the beginning inventory for all the stores.
18     cout << "One week ago, 3 new widget stores opened\n";
19     cout << "at the same time with the same beginning\n";
20     cout << "inventory. What was the beginning inventory? ";
21     cin >> begInv;
22
23     // Set each store's inventory.
24     store1 = store2 = store3 = begInv;
25
26     // Get the number of widgets sold at store 1.
27     cout << "How many widgets has store 1 sold? ";
28     cin >> sold;
29     store1 -= sold; // Adjust store 1's inventory.
30
31     // Get the number of widgets sold at store 2.
32     cout << "How many widgets has store 2 sold? ";
33     cin >> sold;
34     store2 -= sold; // Adjust store 2's inventory.
35
36     // Get the number of widgets sold at store 3.
37     cout << "How many widgets has store 3 sold? ";
38     cin >> sold;
39     store3 -= sold; // Adjust store 3's inventory.
40
41     // Display each store's current inventory.
42     cout << "\nThe current inventory of each store:\n";
43     cout << "Store 1: " << store1 << endl;
44     cout << "Store 2: " << store2 << endl;
45     cout << "Store 3: " << store3 << endl;
46
47 }

```

Program Output with Example Input Shown in Bold

One week ago, 3 new widget stores opened
 at the same time with the same beginning
 inventory. What was the beginning inventory? **100**

```
How many widgets has store 1 sold? 25 [Enter]
How many widgets has store 2 sold? 15 [Enter]
How many widgets has store 3 sold? 45 [Enter]
```

The current inventory of each store:

```
Store 1: 75
Store 2: 85
Store 3: 55
```

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`. When constructing such statements, you must realize the precedence of the combined assignment operators is lower than that of the regular math operators. The statement above is equivalent to

```
result = result * (a + 5);
```

which is different from

```
result = result * a + 5;
```

Table 3-10 shows other examples of such statements and their assignment statement equivalencies.

Table 3-10 Example Usage of the Combined Assignment Operators

Example Usage	Equivalent to
<code>x += b + 5;</code>	<code>x = x + (b + 5);</code>
<code>y -= a * 2;</code>	<code>y = y - (a * 2);</code>
<code>z *= 10 - c;</code>	<code>z = z * (10 - c);</code>
<code>a /= b + c;</code>	<code>a = a / (b + c);</code>
<code>c %= d - 3;</code>	<code>c = c % (d - 3);</code>



Checkpoint

- 3.14 Write a multiple assignment statement that assigns 0 to the variables `total`, `subtotal`, `tax`, and `shipping`.
- 3.15 Write statements using combined assignment operators to perform the following:
- Add 6 to `x`.
 - Subtract 4 from `amount`.
 - Multiply `y` by 4.
 - Divide `total` by 27.
 - Store in `x` the remainder of `x` divided by 7.
 - Add `y * 5` to `x`.
 - Subtract `discount` times 4 from `total`.
 - Multiply `increase` by `salesRep` times 5.
 - Divide `profit` by `shares` minus 1000.

3.16 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;
}
```

3.7

Formatting Output

CONCEPT: The `cout` object 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 3-12, for example, which displays three rows of numbers with a space between each one.

Program 3-12

```
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 // Display the first row of numbers
11 cout << num1 << " " << num2 << " " << num3 << endl;
12
13 // Display the second row of numbers
14 cout << num4 << " " << num5 << " " << num6 << endl;
15
16 // Display the third row of numbers
17 cout << num7 << " " << num8 << " " << num9 << endl;
18
19 return 0;
20 }

```

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 the example above, the number 23 will be displayed in a field of 5 spaces. Since 23 only occupies 2 positions on the screen, 3 blank spaces will be printed before it. To further clarify how this works, look at the following statements:

```

value = 23;
cout << "(" << setw(5) << value << ")";

```

This will cause the following output:

```
( 23)
```

Notice the number occupies the last two positions in the field. Since the number did not use the entire field, `cout` filled the extra 3 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 3-13 shows how the numbers in Program 3-12 can be printed in columns that line up perfectly by using `setw`.

Program 3-13

```

1 // This program displays three rows of numbers.
2 #include <iostream>
3 #include <iomanip>      // Required for 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;
11
12    // Display the first row of numbers
13    cout << setw(6) << num1 << setw(6)
14        << num2 << setw(6) << num3 << endl;
15
16    // Display the second row of numbers
17    cout << setw(6) << num4 << setw(6)
18        << num5 << setw(6) << num6 << endl;
19
20    // Display the third row of numbers
21    cout << setw(6) << num7 << setw(6)
22        << num8 << setw(6) << num9 << endl;
23
24    return 0;
}

```

Program Output

2897	5	837
34	7	1623
390	3456	12

By printing each number in a field of 6 positions, they are displayed in perfect columns.



NOTE: A new header file, `<iomanip>`, is included in Program 3-13. It must be used in any program that uses `setw`.

Notice how a `setw` manipulator is used with each value, 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 3-14 shows `setw` being used with an integer, a floating-point number, and a `string` object.

Program 3-14

```

1 // This program demonstrates the setw manipulator being
2 // used with values of various data types.
3 #include <iostream>
4 #include <iomanip>
5 #include <string>
6 using namespace std;
7
8 int main()
9 {
10     int intValue = 3928;
11     double doubleValue = 91.5;
12     string stringValue = "John J. Smith";
13
14     cout << "(" << setw(5) << intValue << ")" << endl;
15     cout << "(" << setw(8) << doubleValue << ")" << endl;
16     cout << "(" << setw(16) << stringValue << ")" << endl;
17     return 0;
18 }
```

Program Output

```
( 3928)
(    91.5)
(    John J. Smith)
```

Program 3-14 can be used to illustrate the following points:

- The field width of a floating-point number includes a position for the decimal point.
- The field width of a `string` object 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 3-15 shows the results of a division operation displayed with different numbers of significant digits.



**Formatting
Numbers with
`setprecision`**

Program 3-15

```

1 // This program demonstrates how setprecision rounds a
2 // floating-point value.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     double quotient, number1 = 132.364, number2 = 26.91;
10
11     quotient = number1 / number2;
12     cout << quotient << endl;
13     cout << setprecision(5) << quotient << endl;
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
19 }
```

Program Output

```

4.91877
4.9188
4.919
4.92
4.9
5
```

The first value is displayed in line 12 without the `setprecision` manipulator. (By default, the system in the illustration displays floating-point values with 6 significant digits.) The subsequent `cout` statements print the same value, but rounded to 5, 4, 3, 2, and 1 significant digits.

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.

Table 3-11 The `setprecision` Manipulator

Number	Manipulator	Value Displayed
28.92786	<code>setprecision(3)</code>	28.9
21	<code>setprecision(5)</code>	21
109.5	<code>setprecision(4)</code>	109.5
34.28596	<code>setprecision(2)</code>	34

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 3-16 shows how the `setw` and `setprecision` manipulators may be combined to fully control the way floating-point numbers are displayed.

Program 3-16

```
1 // This program asks for sales amounts for 3 days. The total
2 // 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 the total sales.
20    total = day1 + day2 + day3;
21
22    // Display the sales amounts.
23    cout << "\nSales Amounts\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
31 }
```

Program Output with Example Input Shown in Bold

```
Enter the sales for day 1: 321.57 Enter
Enter the sales for day 2: 269.62 Enter
Enter the sales for day 3: 307.77 Enter
```

Sales Amounts

```
-----
Day 1: 321.57
Day 2: 269.62
Day 3: 307.77
Total: 898.96
```

The fixed Manipulator

The `setprecision` manipulator can sometimes surprise you in an undesirable way. When the precision of a number is set to a lower value, numbers tend to be printed in scientific notation. For example, here is the output of Program 3-16 with larger numbers being input:

Program 3-16

Program Output with Example Input Shown in Bold

Enter the sales for day 1: **145678.99**

Enter the sales for day 2: **205614.85**

Enter the sales for day 3: **198645.22**

Sales Amounts

Day 1: 1.4568e+005

Day 2: 2.0561e+005

Day 3: 1.9865e+005

Total: 5.4994e+005

Another stream manipulator, `fixed`, forces `cout` to print the digits in *fixed-point notation*, or decimal. Program 3-17 shows how the `fixed` manipulator is used.

Program 3-17

```

1 // This program asks for sales amounts for 3 days. The total
2 // 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 the total sales.
20    total = day1 + day2 + day3;
21

```

```

22     // Display the sales amounts.
23     cout << "\nSales Amounts\n";
24     cout << "-----\n";
25     cout << setprecision(2) << fixed;
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
31 }

```

Program Output with Example Input Shown in Bold

Enter the sales for day 1: **1321.87** Enter
 Enter the sales for day 2: **1869.26** Enter
 Enter the sales for day 3: **1403.77** Enter

Sales Amounts

 Day 1: 1321.87
 Day 2: 1869.26
 Day 3: 1403.77
 Total: 4594.90

The statement in line 25 uses the **fixed** manipulator:

```
cout << setprecision(2) << fixed;
```

When the **fixed** manipulator is used, all floating-point numbers that are subsequently printed will be displayed in fixed-point notation, with the number of digits to the right of the decimal point specified by the **setprecision** manipulator.

When the **fixed** and **setprecision** manipulators are used together, the value specified by the **setprecision** manipulator will be the number of digits to appear after the decimal point, not the number of significant digits. For example, look at the following code:

```
double x = 123.4567;
cout << setprecision(2) << fixed << x << endl;
```

Because the **fixed** manipulator is used, the **setprecision** manipulator will cause the number to be displayed with two digits after the decimal point. The value will be displayed as 123.46.

The **showpoint** Manipulator

By default, floating-point numbers are not displayed with trailing zeros, and floating-point numbers that do not have a fractional part are not displayed with a decimal point. For example, look at the following code:

```
double x = 123.4, y = 456.0;
cout << setprecision(6) << x << endl;
cout << y << endl;
```

The cout statements will produce the following output:

```
123.4
456
```

Although six significant digits are specified for both numbers, neither number is displayed with trailing zeros. If we want the numbers padded with trailing zeros, we must use the `showpoint` manipulator as shown in the following code:

```
double x = 123.4, y = 456.0;
cout << setprecision(6) << showpoint << x << endl;
cout << y << endl;
```

These cout statements will produce the following output:

```
123.400
456.000
```



NOTE: With most compilers, trailing zeros are displayed when the `setprecision` and `fixed` manipulators are used together.

The left and right Manipulators

Normally output is right-justified. For example, look at the following code:

```
double x = 146.789, y = 24.2, z = 1.783;
cout << setw(10) << x << endl;
cout << setw(10) << y << endl;
cout << setw(10) << z << endl;
```

Each of the variables, x, y, and z, is displayed in a print field of 10 spaces. The output of the cout statements is

```
146.789
 24.2
 1.783
```

Notice each value is right-justified, or aligned to the right of its print field. You can cause the values to be left-justified by using the `left` manipulator, as shown in the following code.

```
double x = 146.789, y = 24.2, z = 1.783;
cout << left << setw(10) << x << endl;
cout << setw(10) << y << endl;
cout << setw(10) << z << endl;
```

The output of these cout statements is

```
146.789
24.2
1.783
```

In this case, the numbers are aligned to the left of their print fields. The `left` manipulator remains in effect until you use the `right` manipulator, which causes all subsequent output to be right-justified.

Table 3-12 summarizes the manipulators we have discussed.

Table 3-12 Stream Manipulators

Stream Manipulator	Description
<code>setw(n)</code>	Establishes a print field of <i>n</i> spaces.
<code>fixed</code>	Displays floating-point numbers in fixed-point notation.
<code>showpoint</code>	Causes a decimal point and trailing zeros to be displayed, even if there is no fractional part.
<code>setprecision(n)</code>	Sets the precision of floating-point numbers.
<code>left</code>	Causes subsequent output to be left-justified.
<code>right</code>	Causes subsequent output to be right-justified.



Checkpoint

- 3.17 Write `cout` statements with stream manipulators that perform the following:
- Display the number 34.789 in a field of nine spaces with two decimal places of precision.
 - Display the number 7.0 in a field of five spaces with three decimal places of precision.
The decimal point and any trailing zeros should be displayed.
 - Display the number 5.789e+12 in fixed-point notation.
 - Display the number 67 left-justified in a field of seven spaces.
- 3.18 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>
return 0;
cout << left;
using namespace std;
cout << setw(20);
cout << right;
```

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.

- 3.19 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 4 places of precision, in a field
    // 5 spaces wide, making sure the decimal point is always
    // displayed.
    return 0;
}
```

3.8

Working with Characters and string Objects

CONCEPT: Special functions exist for working with characters and `string` objects.

Although it is possible to use `cin` with the `>>` operator to input strings, it can cause problems of which you need to be aware. When `cin` reads input, it passes over and ignores any leading *whitespace* characters (spaces, tabs, or line breaks). Once it comes to the first nonblank character and starts reading, it stops reading when it gets to the next whitespace character. Program 3-18 illustrates this problem.

Program 3-18

```
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
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: **Kate Smith**

Enter the city you live in: Hello, Kate

You live in Smith

Notice the user was never given the opportunity to enter the city. In the first input statement, when `cin` came to the space between `Kate` and `Smith`, it stopped reading, storing just `Kate` as the value of `name`. In the second input statement, `cin` used the leftover characters it found in the keyboard buffer and stored `Smith` as the value of `city`.

To work around this problem, you can use a C++ function named `getline`. The `getline` function reads an entire line, including leading and embedded spaces, and stores 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` object receiving the input:

```
getline(cin, inputLine);
```

Program 3-19 illustrates using the `getline` function.

Program 3-19

```
1 // This program demonstrates 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);
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: **Kate Smith**

Enter the city you live in: **Raleigh**

Hello, Kate Smith

You live in Raleigh

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 are 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 3-20.

Program 3-20

```

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**
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 you 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, tab key, or `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, the `cin` object has a built-in function named `get` that is helpful. Because the `get` function is built into the `cin` object, we say that it is a *member function* of `cin`. The `get` member function reads a single character, including any whitespace character. If the program needs to store the character being read, the `get` member function can be called in either of the following ways. In both examples, assume `ch` is the name of a `char` variable into which the character is being read.

```

cin.get(ch);
ch = cin.get();
```

If the program is using the `cin.get` function simply to pause 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 3-21 illustrates all three ways to use the `cin.get` function.

Program 3-21

```
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
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 Program 3-22.

Program 3-22

```
1 // This program demonstrates a problem that occurs
2 // when you mix cin >> with cin.get().
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     char ch;           // Define a character variable
9     int number;        // Define an integer variable
10}
```

(program continues)

Program 3-22 (continued)

```

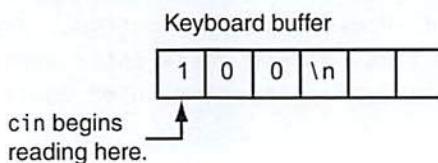
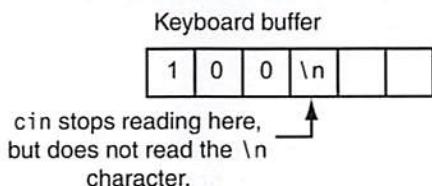
11     cout << "Enter a number: ";
12     cin >> number;           // Read an integer
13     cout << "Enter a character: ";
14     ch = cin.get();          // Read a character
15     cout << "Thank You!\n";
16
17 }
```

Program Output with Example Input Shown in BoldEnter a number: **100**

Enter a character: Thank You!

When this program runs, line 12 lets the user enter a number, but it appears as though the statement in line 14 is skipped. This happens because `cin >>` and `cin.get()` use slightly different techniques for reading data.

In the example run of the program, when line 12 executed, the user entered 100 and pressed the key. Pressing the key causes a newline character ('`\n`') to be stored in the keyboard buffer, as shown in Figure 3-5. The `cin >>` statement in line 12 begins reading the data that the user entered, and stops reading when it comes to the newline character. This is shown in Figure 3-6. The newline character is not read, but remains in the keyboard buffer.

Figure 3-5 Contents of the keyboard buffer**Figure 3-6** `cin` stops reading at the newline character

When the `cin.get()` function in line 14 executes, it begins reading the keyboard buffer where the previous input operation stopped. That means `cin.get()` reads the newline character, without giving the user a chance to enter any more input. You can remedy this situation by using the `cin.ignore` function, described in the following section.

Using `cin.ignore`

To solve the problem previously described, you can use another of the `cin` object's member functions named `ignore`. The `cin.ignore` function tells the `cin` object to skip one or more characters in the keyboard buffer. Here is its general form:

```
cin.ignore(n, c);
```

The arguments shown in the parentheses are optional. If 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 skip only the very next character. Here's an example:

```
cin.ignore();
```

Program 3-23, which is a modified version of Program 3-22, demonstrates the function. Notice a call to `cin.ignore` has been inserted in line 13, right after the `cin >>` statement.

Program 3-23

```
1 // This program successfully uses both
2 // cin >> and cin.get() for keyboard input.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     char ch;
9     int number;
10
11    cout << "Enter a number: ";
12    cin >> number;
13    cin.ignore();           // Skip the newline character
14    cout << "Enter a character: ";
15    ch = cin.get();
16    cout << "Thank You!\n";
17
18 }
```

Program Output with Example Input Shown in Bold

```
Enter a number: 100 
Enter a character: Z 
Thank You!
```

string Member Functions and Operators

C++ `string` objects also have a number of member functions. For example, if you want to know the length of the string that is stored in a `string` object, you can call the object's `length` member function. Here is an example of how to use it:

```
string state = "Texas";
int size = state.length();
```

The first statement creates a `string` object named `state` and initializes it with the string "Texas". The second statement defines an `int` variable named `size` and initializes it with the length of the string in the `state` object. After this code executes, the `size` variable will hold the value 5.

Certain operators also work with `string` objects. 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 ";
string 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 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 in the previous example could also have been written using the `+=` combined assignment operator, to achieve the same result:

```
greeting1 += name2;
```

You will learn about other useful `string` member functions and operators in Chapter 10.

3.9

More Mathematical Library Functions

CONCEPT: The C++ runtime 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 <cmath> 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)	The argument, x, can be a double, a float, or a long double. Returns the value of x rounded to the nearest whole number. For example, if x is 2.8, the function returns 3.0, or if x is 2.1, the function returns 2.0. The return type is the same as the type of the argument.
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.
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.

Each of these functions is as simple to use as the pow function. The following program segment demonstrates the sqrt function, which returns the square root of a number:

```
cout << "Enter a number: ";
cin >> num;
s = sqrt(num);
cout << "The square root of " << num << " is " << s << endl;
```

Here is the output of the program segment, with 25 as the number entered by the user:

```
Enter a number: 25
The square root of 25 is 5
```

Program 3-24 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 3-24

```

1 // This program asks for the lengths of the two sides of a
2 // right triangle. The length of the hypotenuse is then
3 // calculated and displayed.
4 #include <iostream>
5 #include <iomanip>      // For setprecision
6 #include <cmath>         // For the sqrt and pow functions
7 using namespace std;
8
9 int main()
10 {
11     double a, b, c;
12
13     cout << "Enter the length of side a: ";
14     cin >> a;
15     cout << "Enter the length of side b: ";
16     cin >> b;
17     c = sqrt(pow(a, 2.0) + pow(b, 2.0));
18     cout << "The length of the hypotenuse is ";
19     cout << setprecision(2) << c << endl;
20
21 }

```

Program Output with Example Input Shown in Bold

Enter the length of side a: **5.0**

Enter the length of side b: **12.0**

The length of the hypotenuse is 13

The following statement, taken from Program 3-24, 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 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

Random numbers are useful for lots of different programming tasks. The following are just a few examples:

- Random numbers are commonly used in games. For example, computer games that let the player roll dice use random numbers to represent the values of the dice.

- Programs that show cards being drawn from a shuffled deck use random numbers to represent the face values of the cards.
- Random numbers are useful in simulation programs. In some simulations, the computer must randomly decide how a person, animal, insect, or other living being will behave. Formulas can be constructed in which a random number is used to determine various actions and events that take place in the program.
- Random numbers are useful in statistical programs that must randomly select data for analysis.
- Random numbers are commonly used in computer security to encrypt sensitive data.

The C++ library has a function, `rand()`, that you can use to generate random numbers. (The `rand()` function requires the `<cstdlib>` header file.) The number returned from the function is an `int`. Here is an example of its usage:

```
y = rand();
```

After this statement executes, the variable `y` will contain a random number. In actuality, the numbers produced by `rand()` are pseudorandom. The function uses an algorithm that produces the same sequence of numbers each time the program is repeated on the same system. For example, suppose the following statements are executed:

```
cout << rand() << endl;
cout << rand() << endl;
cout << rand() << endl;
```

The three numbers displayed will appear to be random, but each time the program runs, the same three values will be generated. In order to randomize the results of `rand()`, the `srand()` function must be used. `srand()` accepts an `unsigned int` argument, which acts as a seed value for the algorithm. By specifying different seed values, `rand()` will generate different sequences of random numbers.

A common practice for getting unique seed values is to call the `time` function, which is part of the standard library. The `time` function returns the number of seconds that have elapsed since midnight, January 1, 1970. The `time` function requires the `<ctime>` header file, and you pass 0 as an argument to the function. Program 3-25 demonstrates this. The program should generate three different random numbers each time it is executed.

Program 3-25

```
1 // This program demonstrates random numbers.
2 #include <iostream>
3 #include <cstdlib>      // For rand and srand
4 #include <ctime>        // For the time function
5 using namespace std;
6
7 int main()
8 {
9     // Get the system time.
10    unsigned seed = time(0);
11
12    // Seed the random number generator.
13    srand(seed);
14}
```

(program continues)

Program 3-25 (continued)

```

15     // Display three random numbers.
16     cout << rand() << endl;
17     cout << rand() << endl;
18     cout << rand() << endl;
19     return 0;
20 }
```

Program Output

23861
20884
21941

If you wish to limit the range of the random number, use the following formula:

```
y = (rand() % (maxValue - minValue + 1)) + minValue;
```

In the formula, *minValue* is the lowest number in the range, and *maxValue* is the highest number in the range. For example, the following code assigns a random number in the range of 1 through 100 to the variable *y*:

```

const int MIN_VALUE = 1;
const int MAX_VALUE = 100;
y = (rand() % (MAX_VALUE - MIN_VALUE + 1)) + MIN_VALUE;
```

As another example, the following code assigns a random number in the range of 100 through 200 to the variable *y*:

```

const int MIN_VALUE = 100;
const int MAX_VALUE = 200;
y = (rand() % (MAX_VALUE - MIN_VALUE + 1)) + MIN_VALUE;
```

The following *In the Spotlight* section demonstrates how to use random numbers to simulate rolling dice.

In the Spotlight: Using Random Numbers



Dr. Kimura teaches an introductory statistics class and has asked you to write a program that he can use in class to simulate the rolling of dice. The program should randomly generate two numbers in the range of 1 through 6 and display them. Program 3-26 shows the program, with three examples of program output.

Program 3-26

```

1 // This program simulates rolling dice.
2 #include <iostream>
3 #include <cstdlib>    // For rand and srand
4 #include <ctime>      // For the time function
5 using namespace std;
```

```

6
7 int main()
8 {
9     // Constants
10    const int MIN_VALUE = 1; // Minimum die value
11    const int MAX_VALUE = 6; // Maximum die value
12
13    // Variables
14    int die1; // To hold the value of die #1
15    int die2; // To hold the value of die #2
16
17    // Get the system time.
18    unsigned seed = time(0);
19
20    // Seed the random number generator.
21    srand(seed);
22
23    cout << "Rolling the dice...\n";
24    die1 = (rand() % (MAX_VALUE - MIN_VALUE + 1)) + MIN_VALUE;
25    die2 = (rand() % (MAX_VALUE - MIN_VALUE + 1)) + MIN_VALUE;
26    cout << die1 << endl;
27    cout << die2 << endl;
28
29    return 0;
}

```

Program Output

Rolling the dice...
5
2

Program Output

Rolling the dice...
4
6

Program Output

Rolling the dice...
3
1

**Checkpoint**

- 3.20 Write a short description of each of the following functions:

cos	log	sin
exp	log10	sqrt
fmod	pow	tan

- 3.21 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.

- 3.22 To find the cube root (the third root) of a number, raise it to the power of $\frac{1}{3}$. To find the fourth root of a number, raise it to the power of $\frac{1}{4}$. Write a statement that will find the fifth root of the variable x and store the result in the variable y .

3.23 The cosecant of the angle a is

$$\frac{1}{\sin \alpha}$$

Write a statement that calculates the cosecant of the angle stored in the variable *a*, and stores it in the variable *y*.

3.10 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 3-27 is shown with a hand trace chart. The program uses the following four variables: num1, num2, num3, and avg. Notice the hand trace chart has a column for each variable, and a row for each line of code in function main.

Program 3-27

```
1 // This program asks for three numbers, then
2 // displays the average of the numbers.
3 #include <iostream>
4 using namespace std;

5 int main()
6 {
7     double num1, num2, num3, avg;
8     cout << "Enter the first number: ";
9     cin >> num1;
10    cout << "Enter the second number: ";
11    cin >> num2;
12    cout << "Enter the third number: ";
13    cin >> num3;
14    avg = num1 + num2 + num3 / 3;
15    cout << "The average is " << avg << endl;
16    return 0;
17 }
```

This program, which asks the user to enter three numbers and then displays the average of the numbers, has a bug. It does not display the correct average. The output of a sample session with the program follows.

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
```

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 this program, you step through each statement, observing the operation that is taking place, 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 know the contents of a variable.

Program 3-27 (with hand trace chart filled)

```
1 // This program asks for three numbers, then
2 // displays the average of the numbers.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     double num1, num2, num3, avg;
9     cout << "Enter the first number: ";
10    cin >> num1;
11    cout << "Enter the second number: ";
12    cin >> num2;
13    cout << "Enter the third number: ";
14    cin >> num3;
15    avg = num1 + num2 + num3 / 3;
16    cout << "The average is " << avg << endl;
17 }
```

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 that performs the math operation in line 14, 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.

3.11

Focus on Problem Solving: A Case Study

General Crates, Inc. builds custom-designed wooden crates. With materials and labor, it costs GCI \$0.23 per cubic foot to build a crate. In turn, they charge their customers \$0.50 per cubic foot for the crate. You have been asked to write a program that calculates the volume (in cubic feet), cost, customer price, and profit of any crate GCI builds.

Variables

Table 3-14 shows the named constants and variables needed.

Table 3-14 Named Constants and Variables

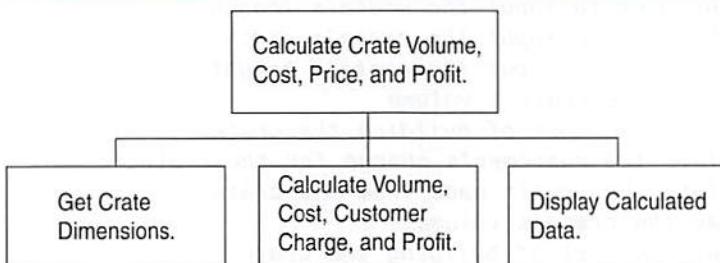
Constant or Variable	Description
COST_PER_CUBIC_FOOT	A named constant, declared as a <code>double</code> and initialized with the value 0.23. This represents the cost to build a crate, per cubic foot.
CHARGE_PER_CUBIC_FOOT	A named constant, declared as a <code>double</code> and initialized with the value 0.5. This represents the amount charged for a crate, per cubic foot.
length	A <code>double</code> variable to hold the length of the crate, which is input by the user.
width	A <code>double</code> variable to hold the width of the crate, which is input by the user.
height	A <code>double</code> variable to hold the height of the crate, which is input by the user.
volume	A <code>double</code> variable to hold the volume of the crate. The value stored in this variable is calculated.
cost	A <code>double</code> variable to hold the cost of building the crate. The value stored in this variable is calculated.
charge	A <code>double</code> variable to hold the amount charged to the customer for the crate. The value stored in this variable is calculated.
profit	A <code>double</code> variable to hold the profit GCI makes from the crate. The value stored in this variable is calculated.

Program Design

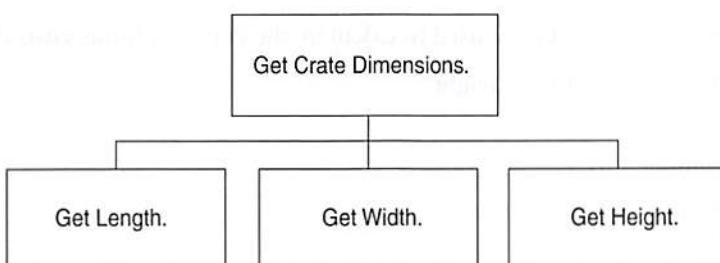
The program must perform the following general steps:

1. Ask the user to enter the dimensions of the crate (the crate's length, width, and height).
2. Calculate the crate's volume, the cost of building the crate, the customer's charge, and the profit made.
3. Display the data calculated in Step 2.

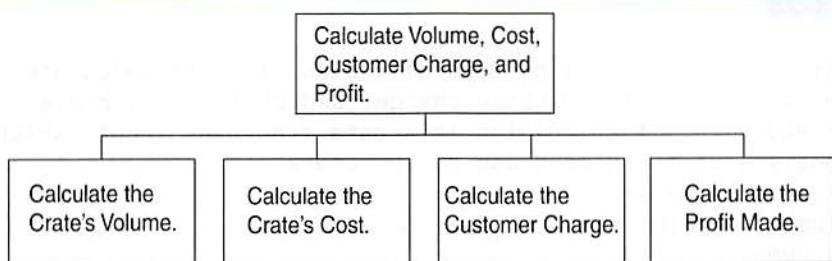
A general hierarchy chart for this program is shown in Figure 3-7.

Figure 3-7 Hierarchy chart for the program

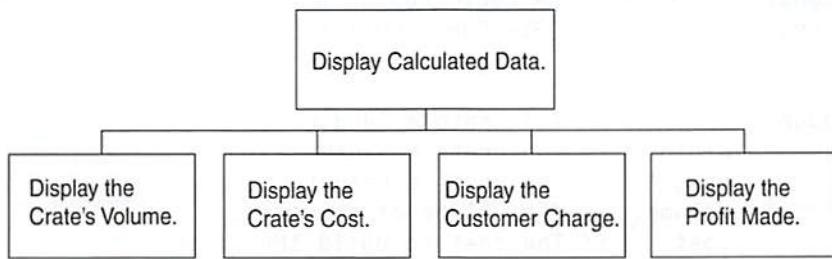
The “Get Crate Dimensions” step is shown in greater detail in Figure 3-8.

Figure 3-8 Hierarchy chart for the “Get Crate Dimensions” step

The “Calculate Volume, Cost, Customer Charge, and Profit” step is shown in greater detail in Figure 3-9.

Figure 3-9 Hierarchy chart for the “Calculate Volume, Cost, Customer Charge, and Profit” step

The “Display Calculated Data” step is shown in greater detail in Figure 3-10.

Figure 3-10 Hierarchy chart for the “Display Calculated Data” step

Pseudocode for the program is as follows:

```

Ask the user to input the crate's length.
Ask the user to input the crate's width.
Ask the user to input the crate's height.
Calculate the crate's volume.
Calculate the cost of building the crate.
Calculate the customer's charge for the crate.
Calculate the profit made from the crate.
Display the crate's volume.
Display the cost of building the crate.
Display the customer's charge for the crate.
Display the profit made from the crate.

```

Calculations

The following formulas will be used to calculate the crate's volume, cost, charge, and profit:

$$\text{volume} = \text{length} \times \text{width} \times \text{height}$$

$$\text{cost} = \text{volume} \times 0.23$$

$$\text{charge} = \text{volume} \times 0.5$$

$$\text{profit} = \text{charge} - \text{cost}$$

The Program

The last step is to expand the pseudocode into the final program, which is shown in Program 3-28.

Program 3-28

```

1 // This program is used by General Crates, Inc. to calculate
2 // the volume, cost, customer charge, and profit of a crate
3 // of any size. It calculates this data from user input, which
4 // consists of the dimensions of the crate.
5 #include <iostream>
6 #include <iomanip>
7 using namespace std;
8
9 int main()
10 {
11     // Constants for cost and amount charged
12     const double COST_PER_CUBIC_FOOT = 0.23;
13     const double CHARGE_PER_CUBIC_FOOT = 0.5;
14
15     // Variables
16     double length, // The crate's length
17             width, // The crate's width
18             height, // The crate's height
19             volume, // The volume of the crate
20             cost,   // The cost to build the crate

```

```
21     charge, // The customer charge for the crate
22     profit; // The profit made on the crate
23
24 // Set the desired output formatting for numbers.
25 cout << setprecision(2) << fixed << showpoint;
26
27 // Prompt the user for the crate's length, width, and height.
28 cout << "Enter the dimensions of the crate (in feet):\n";
29 cout << "Length: ";
30 cin >> length;
31 cout << "Width: ";
32 cin >> width;
33 cout << "Height: ";
34 cin >> height;
35
36 // Calculate the crate's volume, the cost to produce it,
37 // the charge to the customer, and the profit.
38 volume = length * width * height;
39 cost = volume * COST_PER_CUBIC_FOOT;
40 charge = volume * CHARGE_PER_CUBIC_FOOT;
41 profit = charge - cost;
42
43 // Display the calculated data.
44 cout << "The volume of the crate is ";
45 cout << volume << " cubic feet.\n";
46 cout << "Cost to build: $" << cost << endl;
47 cout << "Charge to customer: $" << charge << endl;
48 cout << "Profit: $" << profit << endl;
49
50 }
```

Program Output with Example Input Shown in Bold

Enter the dimensions of the crate (in feet):

Length: **10**

Width: **8**

Height: **4**

The volume of the crate is 320.00 cubic feet.

Cost to build: \$73.60

Charge to customer: \$160.00

Profit: \$86.40

Program Output with Different Example Input Shown in Bold

Enter the dimensions of the crate (in feet):

Length: **12.5**

Width: **10.5**

Height: **8**

The volume of the crate is 1050.00 cubic feet.

Cost to build: \$241.50

Charge to customer: \$525.00

Profit: \$283.50

Review Questions and Exercises

Short Answer

1. 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.

2. Assume a `string` object has been defined as follows:

```
string description;
```

A) Write a `cin` statement that reads in a one-word string.

B) Write a statement that reads in a string that can contain multiple words separated by blanks.

3. What header files must be included in the following program?

```
int main()
{
    double amount = 89.7;
    cout << showpoint << fixed;
    cout << setw(8) << amount << endl;
    return 0;
}
```

4. Complete the following table by determining the value of each expression.

Expression	Value
$28 / 4 - 2$	
$6 + 12 * 2 - 8$	
$4 + 8 * 2$	
$6 + 17 \% 3 - 2$	
$2 + 22 * (9 - 7)$	
$(8 + 7) * 2$	
$(16 + 7) \% 2 - 1$	
$12 / (10 - 6)$	
$(19 - 3) * (2 + 2) / 4$	

5. Write C++ expressions for the following algebraic expressions:

$$a = 12x$$

$$z = 5x + 14y + 6k$$

$$y = x^4$$

$$g = \frac{h + 12}{4k}$$

$$c = \frac{a^3}{b^2 k^4}$$

6. 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 conversion will take place?

- A) `mass` is demoted to an `int`, `units` remains an `int`, and the result of `mass * units` is an `int`.
- B) `units` is promoted to a `float`, `mass` remains a `float`, and the result of `mass * units` is a `float`.
- C) `units` is promoted to a `float`, `mass` remains a `float`, and the result of `mass * units` is a `double`.

7. Assume a program has the following variable definitions:

```
int a, b = 2;  
float c = 4.2;
```

and the following statement:

```
a = b * c;
```

What value will be stored in `a`?

- A) 8.4
- B) 8
- C) 0
- D) None of the above

8. Assume `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;
```

9. Rewrite the following variable definition so that the variable is a named constant.

```
int rate;
```

10. Complete the following table by providing statements with combined assignment operators for 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
<code>x = x + 5;</code> <code>total = total + subtotal;</code> <code>dist = dist / rep;</code> <code>ppl = ppl * period;</code> <code>inv = inv - shrinkage;</code> <code>num = num % 2;</code>	

11. 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;
```

12. Write a cout statement so the variable `divSales` is displayed in a field of 8 spaces, in fixed-point notation, with a precision of 2 decimal places. The decimal point should always be displayed.
13. Write a cout statement so the variable `totalAge` is displayed in a field of 12 spaces, in fixed-point notation, with a precision of 4 decimal places.
14. Write a cout statement so the variable `population` is displayed in a field of 12 spaces, left-justified, with a precision of 8 decimal places. The decimal point should always be displayed.

Fill-in-the-Blank

15. The _____ library function returns the cosine of an angle.
16. The _____ library function returns the sine of an angle.
17. The _____ library function returns the tangent of an angle.
18. The _____ library function returns the exponential function of a number.
19. The _____ library function returns the remainder of a floating-point division.
20. The _____ library function returns the natural logarithm of a number.
21. The _____ library function returns the base-10 logarithm of a number.
22. The _____ library function returns the value of a number raised to a power.
23. The _____ library function returns the square root of a number.
24. The _____ file must be included in a program that uses the mathematical functions.

Algorithm Workbench

25. 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 amount of credit and amount of credit used. The program should then display the customer's available credit.

After you write the pseudocode algorithm, convert it to a complete C++ program.

26. Write a pseudocode algorithm for a program that calculates the total of a retail sale. The program should ask for the amount of the sale and the sales tax rate. The sales tax rate should be entered as a floating-point number. For example, if the sales tax rate is 6 percent, the user should enter 0.06. The program should display the amount of sales tax and the total of the sale.

After you write the pseudocode algorithm, convert it to a complete C++ program.

27. Write a pseudocode algorithm for a program that asks the user to enter a golfer's score for three games of golf, and then displays the average of the three scores.

After you write the pseudocode algorithm, convert it to a complete C++ program.

Find the Errors

Each of the following programs has some errors. Locate as many as you can.

```
28. 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;
    }
```

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

    int main()
    {
        int number1, number2;
        float quotient;
        cout << "Enter two numbers and I will divide\n";
        cout << "the first by the second for you.\n";
        cin >> number1, number2;
        quotient = float<static_cast>(number1) / number2;
        cout << quotient
        return 0;
    }
```

```
30. #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;
    }
```

```
31. #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;
        cout << number1 << " " << number2;
        return 0;
    }

32. #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 >> number;
        half /= 2;
        cout << fixedpoint << showpoint << half << endl;
        return 0;
    }

33. #include <iostream>;
    using namespace std;

    int main()
    {
        char name, go;

        cout << "Enter your name: ";
        getline >> name;
        cout << "Hi " << name << endl;
        return 0;
    }
```

Predict the Output

What will each of the following programs display? (Some should be hand traced and require a calculator.)

34. (Assume the user enters 38700. 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;
}
```

35. `#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;
}`

36. (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;
}
```

37. (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 << setprecision(4) << fixed << showpoint << right;
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;
}

```

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 display the number of miles that may be driven per gallon of gas.

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 fixed-point notation, with two decimal places of precision, and be sure the decimal point is always displayed.

3. 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.

4. Average Rainfall

Write a program that calculates the average 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 each month. The program should display a message similar to the following:

The average rainfall for June, July, and August is 6.72 inches.

5. Male and Female Percentages

Write a program that asks the user for the number of males and the number of females registered in a class. The program should display the percentage of males and females in the class.

Hint: Suppose there are 8 males and 12 females in a class. There are 20 students in the class. The percentage of males can be calculated as $8 \div 20 = 0.4$, or 40 percent. The percentage of females can be calculated as $12 \div 20 = 0.6$, or 60 percent.



6. Ingredient Adjuster

A cookie recipe calls for the following ingredients:

- 1.5 cups of sugar
- 1 cup of butter
- 2.75 cups of flour

The recipe produces 48 cookies with this amount of the ingredients. Write a program that asks the user how many cookies he or she wants to make, then displays the number of cups of each ingredient needed for the specified number of cookies.

7. Box Office

A movie theater only keeps a percentage of the revenue earned from ticket sales. The remainder goes to the movie distributor. 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 \$10.00 and a child's ticket is \$6.00.) It should display a report similar to:

Movie Name:	"Wheels of Fury"
Adult Tickets Sold:	382
Child Tickets Sold:	127
Gross Box Office Profit:	\$ 4,582.00
Net Box Office Profit:	\$ 916.40
Amount Paid to Distributor:	\$ 3,665.60



NOTE: Assume the theater keeps 20 percent of the gross box office profit.

8. How Many Widgets?

The Yukon Widget Company manufactures widgets that weigh 12.5 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.

9. How Many Calories?

A bag of cookies holds 30 cookies. The calorie information on the bag claims 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 he or she actually ate, then reports how many total calories were consumed.

10. How Much Insurance?

Many financial experts advise that property owners should 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, then displays the minimum amount of insurance he or she should buy for the property.

11. Automobile Costs

Write a program that asks the user to enter the monthly costs for the following expenses incurred from operating his or her automobile: loan payment, insurance, gas, oil, tires,

and maintenance. The program should then display the total monthly cost of these expenses, and the total annual cost of these expenses.

12. Celsius to Fahrenheit

Write a program that converts Celsius temperatures to Fahrenheit temperatures. The formula is

$$F = \frac{9}{5}C + 32$$

F is the Fahrenheit temperature, and C is the Celsius temperature.

13. Currency

Write a program that will convert U.S. dollar amounts to Japanese yen and to euros, storing the conversion factors in the constants `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”. If you cannot find the most recent exchange rates, use the following:

1 Dollar = 98.93 Yen

1 Dollar = 0.74 Euros

Format your currency amounts in fixed-point notation, with two decimal places of precision, and be sure the decimal point is always displayed.

14. 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 (i.e. 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:

$$S = \frac{T}{1.06}$$

S is the product sales and T is the total income (product sales plus sales tax).

The program should display a report similar to:

Month: October

Total Collected:	\$ 26572.89
Sales:	\$ 25068.76
County Sales Tax:	\$ 501.38
State Sales Tax:	\$ 1002.75
Total Sales Tax:	\$ 1504.13

15. Property Tax

A county collects property taxes on the assessment value of property, which is 60 percent of the property's actual value. If an acre of land is valued at \$10,000, its assessment value is \$6,000. The property tax is then 75¢ for each \$100 of the assessment value. The tax for the acre assessed at \$6,000 will be \$45. Write a program that asks for the actual value of a piece of property, then displays the assessment value and property tax.

16. Senior Citizen Property Tax

Madison County provides a \$5,000 homeowner exemption for its senior citizens. For example, if a senior's house is valued at \$158,000, its assessed value would be \$94,800, as explained above. However, he would only pay tax on \$89,800. At last year's tax rate of \$2.64 for each \$100 of assessed value, the property tax would be \$2,370.72. In addition to the tax break, senior citizens are allowed to pay their property tax in four 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 the quarterly tax bill will be.

17. Math Tutor

Write a program that can be used as a math tutor for a young student. The program should display two random numbers to be added, such as

$$\begin{array}{r} 247 \\ +129 \\ \hline \end{array}$$

The program should then pause while the student works on the problem. When the student is ready to check the answer, he or she can press a key and the program will display the correct solution:

$$\begin{array}{r} 247 \\ +129 \\ \hline 376 \end{array}$$

18. 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

$$\text{Amount} = \text{Principal} \times \left(1 + \frac{\text{Rate}}{\text{T}}\right)^{\text{T}}$$

Principal is the balance in the savings account, **Rate** is the interest rate, and **T** is the number of times the interest is compounded during a year (**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:

Interest Rate:	4.25%
Times Compounded:	12
Principal:	\$ 1000.00
Interest:	\$ 43.34
Amount in Savings:	\$ 1043.34

19. Monthly Payments

The monthly payment on a loan may be calculated by the following formula:

$$\text{Payment} = \frac{\text{Rate} \times (1 + \text{Rate})^N}{((1 + \text{Rate})^N - 1)} \times L$$

Rate is the monthly interest rate, which is the annual interest rate divided by 12. (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 then displays a report similar to:

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

20. Pizza Pi

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:

- Ask the user for the diameter of the pizza in inches.
- Calculate the number of slices that may be taken from a pizza of that size.
- Display a message telling the number of slices.

To calculate the number of slices that may be taken from the pizza, you must know the following facts:

- Each slice should have an area of 14.125 inches.
- To calculate the number of slices, simply divide the area of the pizza by 14.125.
- The area of the pizza is calculated with this formula:

$$\text{Area} = \pi r^2$$



NOTE: π is the Greek letter pi. 3.14159 can be used as its value. The variable r is the radius of the pizza. Divide the diameter by 2 to get the radius.

Make sure the output of the program displays the number of slices in fixed-point notation, rounded to one decimal place of precision. Use a named constant for pi.

21. How Many Pizzas?

Modify the program you wrote in Programming Challenge 20 (Pizza Pi) so it reports the number of pizzas you need to buy for a party if each person attending is expected to eat an average of four 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.

22. 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.

23. Stock Transaction Program

Last month Joe purchased some stock in Acme Software, Inc. Here are the details of the purchase:

- The number of shares that Joe purchased was 1,000.
- When Joe purchased the stock, he paid \$45.50 per share.

- Joe paid his stockbroker a commission that amounted to 2 percent of the amount he paid for the stock.

Two weeks later, Joe sold the stock. Here are the details of the sale:

- The number of shares that Joe sold was 1,000.
- He sold the stock for \$56.90 per share.
- He paid his stockbroker another commission that amounted to 2 percent 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.
- Display the amount of profit that Joe made after selling his stock and paying the two commissions to his broker. (If the amount of profit that your program displays is a negative number, then Joe lost money on the transaction.)

24. Planting Grapevines

A vineyard owner is planting several new rows of grapevines, and needs to know how many grapevines to plant in each row. She has determined that after measuring the length of a future row, she can use the following formula to calculate the number of vines that will fit in the row, along with the trellis end-post assemblies that will need to be constructed at each end of the row:

$$V = \frac{R - 2E}{S}$$

The terms in the formula are:

V is the number of grapevines that will fit in the row.

R is the length of the row, in feet.

E is the amount of space, in feet, used by an end-post assembly.

S is the space between vines, in feet.

Write a program that makes the calculation for the vineyard owner. The program should ask the user to input the following:

- The length of the row, in feet
- The amount of space used by an end-post assembly, in feet
- The amount of space between the vines, in feet

Once the input data has been entered, the program should calculate and display the number of grapevines that will fit in the row.

25. Word Game

Write a program that plays a word game with the user. The program should ask the user to enter the following:

- His or her name
- His or her age
- The name of a city
- The name of a college
- A profession

- A type of animal
- A pet's name

After the user has entered these items, the program should display the following story, inserting the user's input into the appropriate locations:

There once was a person named *NAME* who lived in *CITY*. At the age of *AGE*, *NAME* went to college at *COLLEGE*. *NAME* graduated and went to work as a *PROFESSION*. Then, *NAME* adopted a(n) *ANIMAL* named *PETNAME*. They both lived happily ever after!

TOPICS

- | | |
|---|--|
| 4.1 Relational Operators | 4.10 Menus |
| 4.2 The if Statement | 4.11 Focus on Software Engineering:
Validating User Input |
| 4.3 Expanding the if Statement | 4.12 Comparing Characters and Strings |
| 4.4 The if/else Statement | 4.13 The Conditional Operator |
| 4.5 Nested if Statements | 4.14 The switch Statement |
| 4.6 The if/else if Statement | 4.15 More about Blocks and Variable
Scope |
| 4.7 Flags | |
| 4.8 Logical Operators | |
| 4.9 Checking Numeric Ranges
with Logical Operators | |

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 whether 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. 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
\geq	Greater than or equal to
\leq	Less than or equal to
$==$	Equal to
\neq	Not equal to

All of the relational operators are binary, which 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 :

$x < y$

Table 4-2 shows examples of several relational expressions that compare the variables x and y .

Table 4-2 Relational Expressions

Expression	What the Expression Means
$x > y$	Is x greater than y ?
$x < y$	Is x less than y ?
$x \geq y$	Is x greater than or equal to y ?
$x \leq y$	Is x less than or equal to y ?
$x == y$	Is x equal to y ?
$x \neq y$	Is x not equal to y ?



NOTE: All the relational operators have left-to-right associativity. Recall that associativity is the order in which an operator works with its operands.

The Value of a Relationship

So, how are relational expressions used in a program? Remember, all expressions have a value. Relational expressions are also known as *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 whether 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 `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 `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-to 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 4-3 shows other relational expressions and their true or false values.

Table 4-3 (Assume x is 10 and y is 7.)

Expression	Value
<code>x < y</code>	False, because x is not less than y.
<code>x > y</code>	True, because x is greater than y.
<code>x >= y</code>	True, because x is greater than or equal to y.
<code>x <= y</code>	False, because x is not less than or equal to y.
<code>y != x</code>	True, because y is not equal to x.

What Is Truth?

The question “what is truth?” is one you would expect to find in a philosophy book, not a C++ programming text. It’s a good question for us to consider, though. If a relational expression can be 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 2-17, those two abstract states are converted to numbers. In C++, relational expressions represent true states with the number 1, and false states with the number 0.



NOTE: As you will see later in this chapter, 1 is not the only value regarded as true.

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
16 }
```

Program Output

```
True is 1
False is 0
```

Let's examine the statements containing the relational expressions, in lines 10 and 11, 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 line 10, the variable `trueValue` is being assigned the result of `x < y`. Since `x` is less than `y`, the expression is true, and the variable `trueValue` is assigned the value 1. In line 11, the expression `y == x` is false, so the variable `falseValue` is set to 0. Table 4-4 shows examples of other statements using relational expressions and their outcomes.



NOTE: Relational expressions have a higher precedence than the assignment operator. In the statement

```
z = x < y;
```

the expression `x < y` is evaluated first, then its value is assigned to `z`.

Table 4-4 (Assume `x` is 10, `y` is 7, and `z`, `a`, and `b` are `ints` or `bools`.)

Statement	Outcome
<code>z = x < y</code>	<code>z</code> is assigned 0 because <code>x</code> is not less than <code>y</code> .
<code>cout << (x > y);</code>	Displays 1 because <code>x</code> is greater than <code>y</code> .
<code>a = x >= y;</code>	<code>a</code> is assigned 1 because <code>x</code> is greater than or equal to <code>y</code> .
<code>cout << (x <= y);</code>	Displays 0 because <code>x</code> is not less than or equal to <code>y</code> .
<code>b = y != x;</code>	<code>b</code> is assigned 1 because <code>y</code> is not equal to <code>x</code> .

When writing statements such as these, it sometimes helps to enclose the relational expression in parentheses, such as:

```
trueValue = (x < y);
falseValue = (y == x);
```

As interesting as relational expressions are, we've only scratched the surface of how to use them. 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:
- `x == 5`
 - `7 <= (x + 2)`
 - `z < 4`
 - `(2 + x) != y`
 - `z != 4`
 - `x >= 9`
 - `x <= (y * 2)`

- 4.2 Indicate whether the following statements about relational expressions are correct or incorrect:
- $x \leq y$ is the same as $y > x$.
 - $x \neq y$ is the same as $y \geq x$.
 - $x \geq y$ is the same as $y \leq x$.
- 4.3 Answer the following questions with a yes or no:
- If it is true that $x > y$ and it is also true that $x < z$, does that mean $y < z$ is true?
 - If it is true that $x \geq y$ and it is also true that $z == x$, does that mean that $z == y$ is true?
 - If it is true that $x \neq y$ and it is also true that $x \neq z$, does that mean that $z \neq y$ is true?
- 4.4 What will the following program display?

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

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

4.2

The if Statement

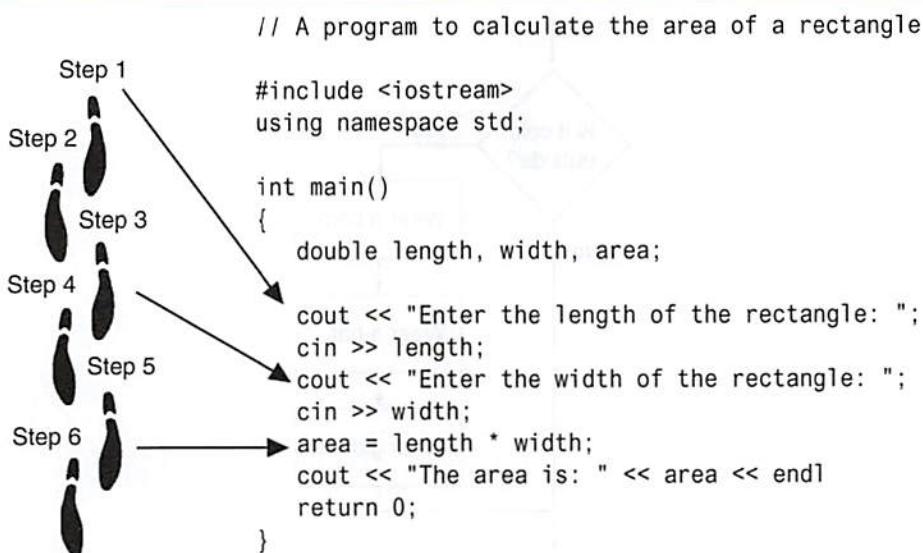
CONCEPT: The *if* statement can cause other statements to execute only under certain conditions.



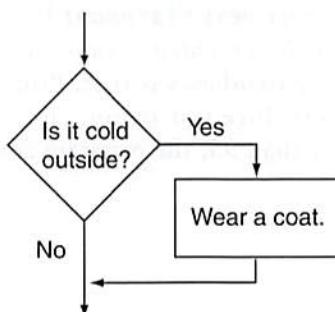
You might think of the statements in a procedural program as individual steps taken as you are walking down a road. To reach the destination, you must start at the beginning and take each step, one after the other, until you reach the destination. The programs you have written so far are like a “path” of execution for the program to follow.

The type of code in Figure 4-1 is called a *sequence structure* because the statements are executed in sequence, 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 is taken only when a specific condition exists. If the condition does not exist, the action is not performed. The flowchart in Figure 4-2 shows the logic of a decision structure. The diamond symbol represents a yes/no

Figure 4-1 Sequence structure

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 an action being performed. If the answer to the question is no (or the condition is false), the program flow follows another path, which skips the action.

Figure 4-2 Decision structure

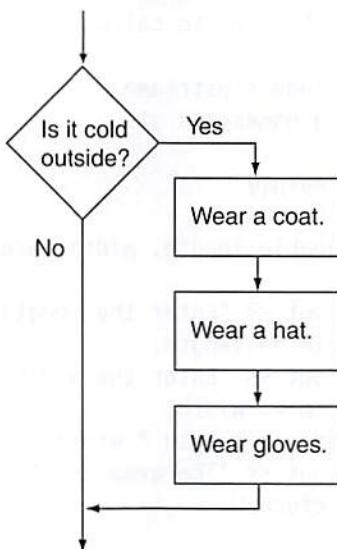
In the flowchart, the action “Wear a coat” is performed only when it is cold outside. If it is not cold outside, the action is skipped. The action is *conditionally executed* because it is performed only when a certain condition (cold outside) exists. Figure 4-3 shows a more elaborate flowchart, where three actions are taken only when it is cold outside.

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.

Figure 4-3 Decision structure

One way to code a decision structure in C++ is with the `if` statement. Here is the general format of the `if` statement:

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

The `if` statement is simple in the way it works: If the value of the expression inside the parentheses is true, the very next `statement` is executed. Otherwise, it is skipped. The `statement` is *conditionally executed* because it only executes under the condition that the `expression` in the parentheses is true. Program 4-2 shows an example of an `if` statement. The user enters three test scores, and the program calculates their average. If the average is greater than 95, the program congratulates the user on obtaining a high score.

Program 4-2

```
1 // This program averages three test scores
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 int main()
7 {
8     const int HIGH_SCORE = 95; // A high score is 95 or greater
9     int score1, score2, score3; // To hold three test scores
10    double average;           // To hold the average score
11}
```

```

12 // Get the three test scores.
13 cout << "Enter 3 test scores and I will average them: ";
14 cin >> score1 >> score2 >> score3;
15
16 // Calculate and display the average score.
17 average = (score1 + score2 + score3) / 3.0;
18 cout << fixed << showpoint << setprecision(1);
19 cout << "Your average is " << average << endl;
20
21 // If the average is a high score, congratulate the user.
22 if (average > HIGH_SCORE)
23     cout << "Congratulations! That's a high score!\n";
24 return 0;
25 }
```

Program Output with Example Input Shown in Bold

Enter 3 test scores and I will average them: **80 90 70**

Your average is 80.0

Program Output with Different Example Input Shown in Bold

Enter 3 test scores and I will average them: **100 100 100**

Your average is 100.0

Congratulations! That's a high score!

Lines 22 and 23 cause the congratulatory message to be printed:

```

if (average > HIGH_SCORE)
    cout << "Congratulations! That's a high score!\n";
```

The cout statement in line 23 is executed only if the average is greater than 95, the value of the HIGH_SCORE constant. If the average is not greater than 95, the cout statement is skipped. Figure 4-4 shows the logic of this if statement.

Figure 4-4 if statement logic

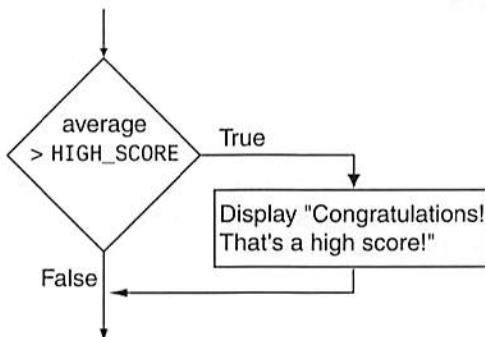


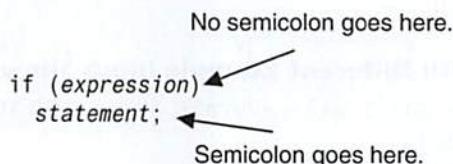
Table 4-5 shows other examples of if statements and their outcomes.

Table 4-5 if Statements and Their Outcomes

Statement	Outcome
if (hours > 40) overTime = true;	Assigns true to the bool variable overTime only if hours is greater than 40
if (value > 32) cout << "Invalid number\n";	Displays the message “Invalid number” only if value is greater than 32
if (overTime == true) payRate *= 2;	Multiplies payRate by 2 only if overTime is equal to true

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 (*expression*) 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 statement that follows it. The statement following the if will always execute, as shown in Program 4-3.

Program 4-3

```

1 // This program demonstrates how a misplaced semicolon
2 // prematurely terminates an if statement.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int x = 0, y = 10;
9
10    cout << "x is " << x << " and y is " << y << endl;
11    if (x > y); // Error! Misplaced semicolon
12        cout << "x is greater than y\n"; //This is always executed.
13    return 0;
14 }
```

Program Output

```
x is 0 and y is 10
x is greater than y
```

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";
```

In both the examples above, the compiler considers the if part and the cout statement as one unit, with a semicolon properly placed at the end. Indention and spacing are for the human readers of a program, not the compiler. Here are two important style rules you should adopt for writing if statements:

- The conditionally executed statement should appear on the line after the if statement.
- The conditionally executed statement 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.

By indenting the conditionally executed statement, you are causing it to stand out visually. This is so that 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.



NOTE: Indentation and spacing are for the human readers of a program, not the compiler. Even though the cout statement following the if statement in Program 4-3 is indented, the semicolon still terminates the if statement.

Comparing Floating-Point Numbers

Because of the way floating-point numbers are stored in memory, rounding errors sometimes occur. This is because some fractional numbers cannot be exactly represented using binary. So, you should be careful when using the equality operator (==) to compare floating-point numbers. For example, Program 4-4 uses two double variables, a and b. Both variables are initialized to the value 1.5. Then, the value 0.000000000000001 is added to a. This should make a's contents different from b's contents. Because of a round-off error, however, the two variables are still the same.

Program 4-4

```
1 // This program demonstrates how floating-point
2 // round-off errors can make equality operations unreliable.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
```

(program continues)

Program 4-4

(continued)

```

8     double a = 1.5;           // a is 1.5.
9     double b = 1.5;           // b is 1.5.
10
11    a += 0.0000000000000001; // Add a little to a.
12    if (a == b)
13        cout << "Both a and b are the same.\n";
14    else
15        cout << "a and b are not the same.\n";
16
17    return 0;
18 }
```

Program Output

Both a and b are the same.

To prevent round-off errors from causing this type of problem, you should stick with greater-than and less-than comparisons with floating-point numbers.

And Now Back to Truth

Now that you've gotten your feet wet with relational expressions and `if` statements, let's look at the subject of truth again. You have seen that a relational expression has the value 1 when it is true and 0 when false. In the world of the `if` statement, however, the concept of truth is expanded. 0 is still false, but all values other than 0 are considered true. This means that any value, even a negative number, represents as 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 the value 1.
- When a relational expression is false, it has the value 0.
- Any expression that has the value 0 is considered false by the `if` statement. This includes the `bool` value `false`, which is equivalent to 0.
- Any expression that has any value other than 0 is considered true by the `if` statement. 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!";
```

The `if` statement above does not test a relational expression, but rather 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 like any other value in an *if* statement: 0 is false and all other values are true. You may also use the return value of function calls as conditional expressions. 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 is executed. This is a powerful programming technique that you will learn more about in Chapter 6.

Don't Confuse == with =

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!";
```

The statement above does not determine whether *x* is equal to 2, it assigns *x* the value 2! Furthermore, the *cout* statement will *always* be executed because the expression *x* = 2 is always 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. That means the value of the expression *x* = 2 is 2. Since 2 is a nonzero value, it is interpreted as a true condition by the *if* statement. Program 4-5 is a version of Program 4-2 that attempts to test for a perfect average of 100. The = operator, however, was mistakenly used in the *if* statement.

Program 4-5

```
1 // This program averages 3 test scores. The if statement
2 // uses the = operator, but the == operator was intended.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     int score1, score2, score3; // To hold three test scores
10    double average;           // To hold the average score
11
12    // Get the three test scores.
13    cout << "Enter 3 test scores and I will average them: ";
14    cin >> score1 >> score2 >> score3;
15
16    // Calculate and display the average score.
17    average = (score1 + score2 + score3) / 3.0;
18    cout << fixed << showpoint << setprecision(1);
19    cout << "Your average is " << average << endl;
20 }
```

(program continues)

Program 4-5 (continued)

```

21     // Our intention is to congratulate the user
22     // for having a perfect score. But, this doesn't work.
23     if (average = 100) // WRONG! This is an assignment!
24         cout << "Congratulations! That's a perfect score!\n";
25     return 0;
26 }
```

Program Output with Example Input Shown in Bold

Enter three test scores and I will average them: **80 90 70** **Enter**
 Your average is 80.0
 Congratulations! That's a perfect score!

Regardless of the average score, this program will print the message congratulating the user on a perfect score.

**Checkpoint**

- 4.5 Write an **if** statement that performs the following logic: if the variable **x** is equal to 20, then assign 0 to the variable **y**.
- 4.6 Write an **if** statement that performs the following logic: if the variable **price** is greater than 500, then assign 0.2 to the variable **discountRate**.
- 4.7 Write an **if** statement that multiplies **payRate** by 1.5 if **hours** is greater than 40.
- 4.8 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;
```

- 4.9 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;
```

4.3**Expanding the **if** Statement**

CONCEPT: The **if** statement can conditionally execute a block of statements enclosed in braces.

What if you want an **if** statement to conditionally execute a group of statements, not just one line? For instance, what if the test-averaging program needed to use several **cout**

statements when a high score was reached? The answer is to enclose all of the conditionally executed statements inside a set of braces. Here is the format:

```
if (expression)
{
    statement;
    statement;
    // Place as many statements here as necessary.
}
```

Program 4-6, another modification of the test-averaging program, demonstrates this type of if statement.

Program 4-6

```
1 // This program averages 3 test scores.
2 // It demonstrates an if statement executing
3 // a block of statements.
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;
7
8 int main()
9 {
10     const int HIGH_SCORE = 95;           // A high score is 95 or greater
11     int score1, score2, score3;         // To hold three test scores
12     double average;                  // To hold the average score
13
14     // Get the three test scores.
15     cout << "Enter 3 test scores and I will average them: ";
16     cin >> score1 >> score2 >> score3;
17
18     // Calculate and display the average score.
19     average = (score1 + score2 + score3) / 3.0;
20     cout << fixed << showpoint << setprecision(1);
21     cout << "Your average is " << average << endl;
22
23     // If the average is high, congratulate the user.
24     if (average > HIGH_SCORE)
25     {
26         cout << "Congratulations!\n";
27         cout << "That's a high score.\n";
28         cout << "You deserve a pat on the back!\n";
29     }
30     return 0;
31 }
```

(program output continues)

Program 4-6

(continued)

Program Output with 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 high score.
 You deserve a pat on the back!

Program Output with Different Example Input Shown in Bold

Enter 3 test scores and I will average them: **80 90 70** **Enter**
 Your average is 80.0

Program 4-6 prints a more elaborate message when the average score is greater than 95. The if statement was expanded to execute three cout statements when highScore is set to true. Enclosing a group of statements inside a set of braces creates a *block* of code. The if statement will execute all the statements in the block, in the order they appear, only when average is greater than 95. Otherwise, the block will be skipped.

Notice all the statements inside the braces are indented. As before, this visually separates the statements from lines that are not indented, making it more obvious they are part of the if statement.



NOTE: Anytime your program has a block of code, all the statements inside the braces should be indented.

Don't Forget the Braces!

If you intend to conditionally execute a block of statements with an if statement, don't forget the braces. Remember, without a set of braces, the if statement only executes the very next statement. Program 4-7 shows the test-averaging program with the braces inadvertently left out of the if statement's block.

Program 4-7

```

1 // This program averages 3 test scores. The braces
2 // were inadvertently left out of the if statement.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     const int HIGH_SCORE = 95;           // A high score is 95 or greater
10    int score1, score2, score3;        // To hold three test scores
11    double average;                  // To hold the average score
12

```

```

13 // Get the three test scores.
14 cout << "Enter 3 test scores and I will average them: ";
15 cin >> score1 >> score2 >> score3;
16
17 // Calculate and display the average score.
18 average = (score1 + score2 + score3) / 3.0;
19 cout << fixed << showpoint << setprecision(1);
20 cout << "Your average is " << average << endl;
21
22 // ERROR! This if statement is missing its braces!
23 if (average > HIGH_SCORE)
24     cout << "Congratulations!\n";
25     cout << "That's a high score.\n";
26     cout << "You deserve a pat on the back!\n";
27 return 0;
28 }
```

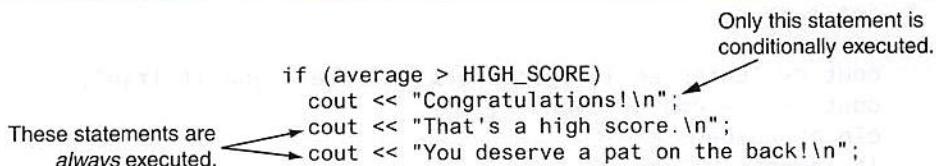
Program Output with Example Input Shown in Bold

Enter 3 test scores and I will average them: **80 90 70**

Your average is 80
 That's a high score.
 You deserve a pat on the back!

The cout statements in lines 25 and 26 are always executed, even when average is not greater than 95. Because the braces have been removed, the if statement only controls execution of line 24. This is illustrated in Figure 4-5.

Figure 4-5 Conditionally executed statement



Checkpoint

- 4.10 Write an if statement that performs the following logic: if the variable sales is greater than 50,000, then assign 0.25 to the commissionRate variable, and assign 250 to the bonus variable.
- 4.11 The following code segment is syntactically correct, but it appears to contain a logic error. Can you find the error?

```

if (interestRate > .07)
    cout << "This account earns a $10 bonus.\n";
    balance += 10.0;

```

4.4

The if/else Statement

CONCEPT: The `if/else` statement will execute one group of statements if the expression is true, or another group of statements if the expression is false.



The `if/else` statement is an expansion of the `if` statement. Here is its format:

```
if (expression)
    statement or block
else
    statement or block
```

As with the `if` statement, an expression is evaluated. If the expression is true, a statement or block of statements is executed. If the expression is false, however, a separate group of statements is executed. Program 4-8 uses the `if/else` statement along with the modulus operator to determine if a number is odd or even.

Program 4-8

```
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     if (number % 2 == 0)
15         cout << number << " is even.\n";
16     else
17         cout << number << " is odd.\n";
18     return 0;
19 }
```

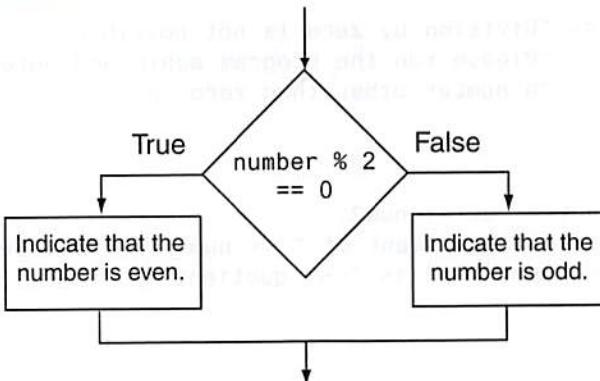
Program Output with Example Input Shown in Bold

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 a statement that is to be executed when the expression is false. When `number % 2` does not equal 0, a message is printed indicating the number is odd. Note 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. Instead of being a momentary detour, like an `if` statement, the `if/else` statement causes program execution to follow one of two exclusive paths. The flowchart in Figure 4-6 shows the logic of this `if/else` statement.

Figure 4-6 `if/else` statement logic



Notice the programming style used to construct the `if/else` statement. The word `else` is at the same level of indentation as `if`. The statement whose execution is controlled by `else` is indented one level. This visually depicts the two paths of execution that may be followed.

Like the `if` part, the `else` part controls a single statement. If you wish to control more than one statement with the `else` part, create a block by writing the lines inside a set of braces. Program 4-9 shows this as a way of handling 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 4-9 shows a way to test the value of a divisor before the division takes place.

Program 4-9

```
1 // This program asks the user for two numbers, num1 and num2.  
2 // num1 is divided by num2 and the result is displayed.  
3 // Before the division operation, however, num2 is tested  
4 // for the value 0. If it contains 0, the division does not  
5 // take place.  
6 #include <iostream>  
7 using namespace std;  
8  
9 int main()  
10 {  
11     double num1, num2, quotient;  
12  
13     // Get the first number.  
14     cout << "Enter a number: ";  
15     cin >> num1;  
16  
17     // Get the second number.  
18     cout << "Enter another number: ";  
19     cin >> num2;
```

(program continues)

Program 4-9

(continued)

```

20
21     // If num2 is not zero, perform the division.
22     if (num2 == 0)
23     {
24         cout << "Division by zero is not possible.\n";
25         cout << "Please run the program again and enter\n";
26         cout << "a number other than zero.\n";
27     }
28     else
29     {
30         quotient = num1 / num2;
31         cout << "The quotient of " << num1 << " divided by ";
32         cout << num2 << " is " << quotient << ".\n";
33     }
34     return 0;
35 }
```

Program Output with Example Input Shown in BoldEnter a number: **10** Enter another number: **0**

Division by zero is not possible.

Please run the program again and enter
a number other than zero.

The value of num2 is tested in line 22 before the division is performed. If the user enters 0, the lines controlled by the if part execute, displaying a message that indicates that the program cannot perform a division by zero. Otherwise, the else part takes control, which divides num1 by num2 and displays the result.

**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 commissionRate unless sales is greater than or equal to 50000.00, in which case it assigns 0.20 to commissionRate.

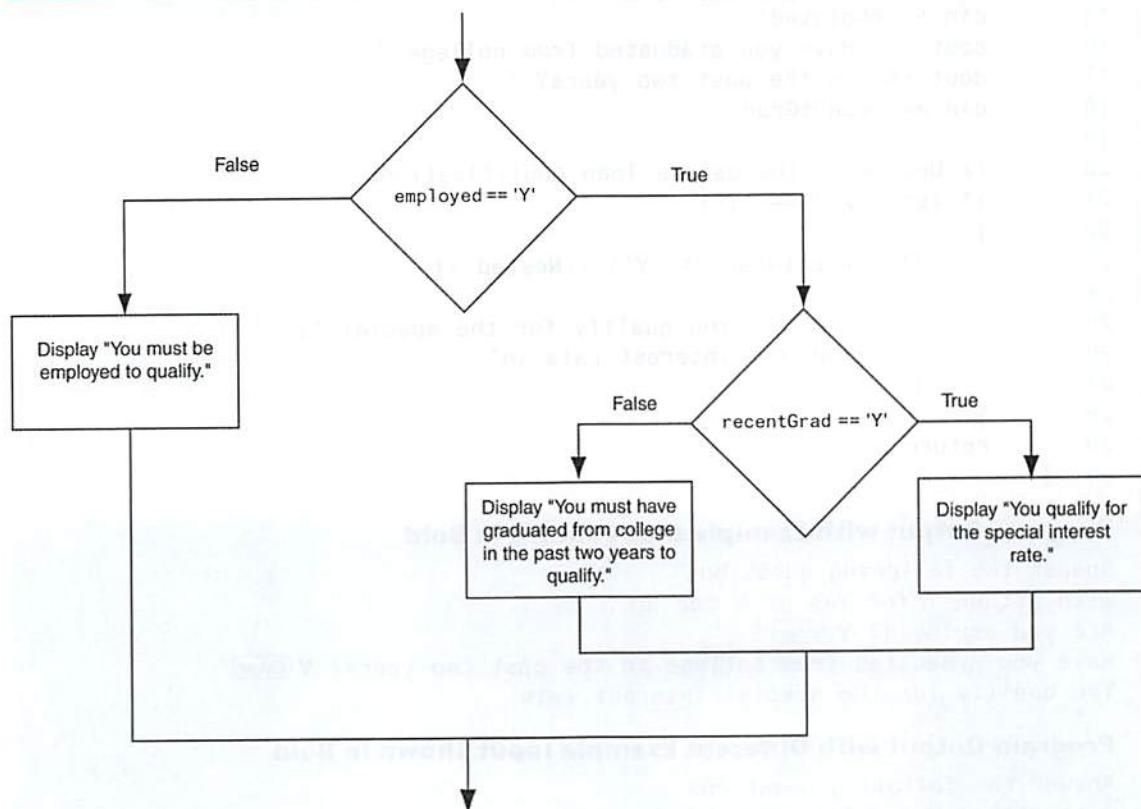
4.5

Nested if Statements

CONCEPT: To test more than one condition, an **if** statement can be nested inside another **if** statement.

Sometimes an **if** statement must be nested inside another **if** statement. 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, and (2) the customer must have recently graduated from college (in the past 2 years). Figure 4-7 shows a flowchart for an algorithm that could be used in such a program.

Figure 4-7 Nested decision structures



If we follow the flow of execution in the flowchart, we see that the expression `employed == 'Y'` is tested. If this expression is false, there is no need to perform further 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, then the customer does not qualify. Program 4-10 shows the code for the complete program.

Program 4-10

```

1 // This program demonstrates the nested if statement.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     char employed,      // Currently employed, Y or N
8         recentGrad;   // Recent graduate, Y or N
9
10    // Is the user employed and a recent graduate?
11    cout << "Answer the following questions\n";
12    cout << "with either Y for Yes or ";
13    cout << "N for No.\n";
14    cout << "Are you employed? ";
15    cin >> employed;
16    cout << "Have you graduated from college ";
17    cout << "in the past two years? ";
18    cin >> recentGrad;
19
20    // Determine the user's loan qualifications.
21    if (employed == 'Y')
22    {
23        if (recentGrad == 'Y') //Nested if
24        {
25            cout << "You qualify for the special ";
26            cout << "interest rate.\n";
27        }
28    }
29    return 0;
30 }
```

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**

Have you graduated from college in the past two years? **Y**
You qualify for the special interest rate.

Program Output with Different Example Input Shown in Bold

Answer the following questions
with either Y for Yes or N for No.

Are you employed? **Y**

Have you graduated from college in the past two years? **N**

Look at the **if** statement that begins in line 21. It tests the expression `employed == 'Y'`. If this expression is true, the **if** statement that begins in line 23 is executed. Otherwise, the program jumps to the `return` statement in line 29 and the program ends.

Notice in the second sample execution of Program 4-10 that the program output does not inform the user whether he or she qualifies for the special interest rate. If the user enters an 'N' (or any character other than 'Y') for `employed` or `recentGrad`, the program does not

print a message letting the user know that he or she does not qualify. An `else` statement should be able to remedy this, as illustrated by Program 4-11.

Program 4-11

```
1 // This program demonstrates the nested if statement.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     char employed, // Currently employed, Y or N
8         recentGrad; // Recent graduate, Y or N
9
10    // Is the user employed and a recent graduate?
11    cout << "Answer the following questions\n";
12    cout << "with either Y for Yes or ";
13    cout << "N for No.\n";
14    cout << "Are you employed? ";
15    cin >> employed;
16    cout << "Have you graduated from college ";
17    cout << "in the past two years? ";
18    cin >> recentGrad;
19
20    // Determine the user's loan qualifications.
21    if (employed == 'Y')
22    {
23        if (recentGrad == 'Y') // Nested if
24        {
25            cout << "You qualify for the special ";
26            cout << "interest rate.\n";
27        }
28        else // Not a recent grad, but employed
29        {
30            cout << "You must have graduated from ";
31            cout << "college in the past two\n";
32            cout << "years to qualify.\n";
33        }
34    }
35    else // Not employed
36    {
37        cout << "You must be employed to qualify.\n";
38    }
39    return 0;
40 }
```

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**

Have you graduated from college in the past two years? **Y**

You must be employed to qualify.

(program output continues)

Program 4-11 (continued)**Program Output with Different Example Input Shown in Bold**

Answer the following questions
with either Y for Yes or N for No.

Are you employed? **Y**

Have you graduated from college in the past two years? **N**

You must have graduated from college in the past two years to qualify.

Program Output with Different Example Input Shown in Bold

Answer the following questions
with either Y for Yes or N for No.

Are you employed? **Y**

Have you graduated from college in the past two years? **Y**

You qualify for the special interest rate.

In this version of the program, both `if` statements have `else` clauses that inform the user why he or she does not qualify for the special interest rate.

Programming Style and Nested Decision Structures

For readability and easier debugging, it's important to use proper alignment and indentation in a set of nested `if` statements. This makes it easier to see which actions are performed by each part of the decision structure. For example, the following code is functionally equivalent to lines 21 through 38 in Program 4-11. Although this code is logically correct, it is very difficult to read, and would be very difficult to debug because it is not properly indented.

```
if (employed == 'Y')
{
    if (recentGrad == 'Y') // Nested if
    {
        cout << "You qualify for the special ";
        cout << "interest rate.\n";
    }
    else // Not a recent grad, but employed
    {
        cout << "You must have graduated from ";
        cout << "college in the past two\n";
        cout << "years to qualify.\n";
    }
}
else // Not employed
{
    cout << "You must be employed to qualify.\n";
}
```

*Don't write code
like this!*

Proper indentation and alignment also makes it easier to see which `if` and `else` clauses belong together, as shown in Figure 4-8.

Figure 4-8 Proper indentation and alignment

```

if (employed == 'Y')
{
    if (recentGrad == 'Y') // Nested if
    {
        cout << "You qualify for the special ";
        cout << "interest rate.\n";
    }
    else // Not a recent grad, but employed
    {
        cout << "You must have graduated from ";
        cout << "college in the past two\n";
        cout << "years to qualify.\n";
    }
}
else // Not employed
{
    cout << "You must be employed to qualify.\n";
}

```

This if and else go together.

This if and else go together.

Testing a Series of Conditions

In the previous example, you saw how a program can use nested decision structures to test more than one condition. It is not uncommon for a program to have a series of conditions to test then perform an action depending on which condition is true. One way to accomplish this is to have a decision structure with numerous other decision structures nested inside it. For example, consider the program presented in the following *In the Spotlight* section.

In the Spotlight: Multiple Nested Decision Structures

Dr. Suarez teaches a literature class and uses the following 10-point grading scale for all of his exams:

Test Score	Grade
90 and above	A
80–89	B
70–79	C
60–69	D
Below 60	F

He has asked you to write a program that will allow a student to enter a test score and then display the grade for that score. Here is the algorithm that you will use:

Ask the user to enter a test score.

Determine the grade in the following manner:

If the score is greater than or equal to 90, then the grade is A.

Otherwise, if the score is greater than or equal to 80, then the grade is B.

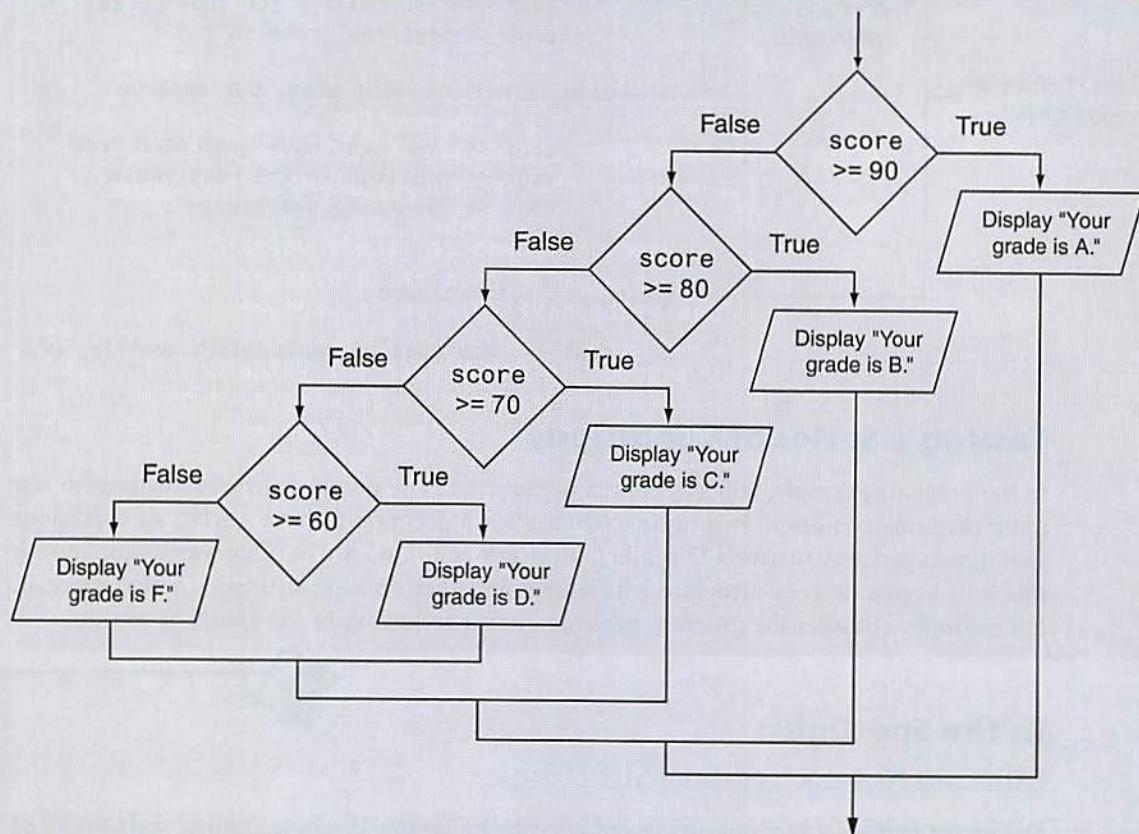
Otherwise, if the score is greater than or equal to 70, then the grade is C.

Otherwise, if the score is greater than or equal to 60, then the grade is D.

Otherwise, the grade is F.

You decide that the process of determining the grade will require several nested decision structures, as shown in Figure 4-9. Program 4-12 shows the code for the complete program. The code for the nested decision structures is in lines 17 through 45.

Figure 4-9 Nested decision structure to determine a grade



Program 4-12

```

1 // This program uses nested if/else statements 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     // Constants for grade thresholds
9     const int A_SCORE = 90,
10        B_SCORE = 80,
11        C_SCORE = 70,
12        D_SCORE = 60;
13
14     int testScore; // To hold a numeric test score
15
16     // Get the numeric test score.
17     cout << "Enter your numeric test score and I will\n";
  
```

```
18     cout << "tell you the letter grade you earned: ";
19     cin >> testScore;
20
21     // Determine the letter grade.
22     if (testScore >= A_SCORE)
23     {
24         cout << "Your grade is A.\n";
25     }
26     else
27     {
28         if (testScore >= B_SCORE)
29         {
30             cout << "Your grade is B.\n";
31         }
32         else
33         {
34             if (testScore >= C_SCORE)
35             {
36                 cout << "Your grade is C.\n";
37             }
38             else
39             {
40                 if (testScore >= D_SCORE)
41                 {
42                     cout << "Your grade is D.\n";
43                 }
44                 else
45                 {
46                     cout << "Your grade is F.\n";
47                 }
48             }
49         }
50     }
51
52     return 0;
53 }
```

Program Output with Example Input Shown in Bold

Enter your numeric test score and I will
tell you the letter grade you earned: **78**

Your grade is C.

Program Output with Different Example Input Shown in Bold

Enter your numeric test score and I will
tell you the letter grade you earned: **84**

Your grade is B.



Checkpoint

- 4.15 If you executed the following code, what would it display if the user enters 5?
What if the user enters 15? What if the user enters 30? What if the user enters -1?

```

int number;
cout << "Enter a number: ";
cin >> number;

if (number > 0)
{
    cout << "Zero\n";

    if (number > 10)
    {
        cout << "Ten\n";

        if (number > 20)
        {
            cout << "Twenty\n";
        }
    }
}

```

4.6

The if/else if Statement

CONCEPT: The `if/else if` statement tests a series of conditions. It is often simpler to test a series of conditions with the `if/else if` statement than with a set of nested `if/else` statements.



Even though Program 4-12 is a simple example, the logic of the nested decision structure is fairly complex. In C++, and many other languages, you can alternatively test a series of conditions using the `if/else if` statement. The `if/else if` statement makes certain types of nested decision logic simpler to write. Here is the general format of the `if/else if` statement:

```

if (expression_1)
{
    statement
    statement
    etc.
}

else if (expression_2)
{
    statement
    statement
    etc.
}

Insert as many else if clauses as necessary

else
{
    statement
    statement
    etc.
}

```

If `expression_1` is true these statements are executed, and the rest of the structure is ignored.

Otherwise, if `expression_2` is true these statements are executed, and the rest of the structure is ignored.

These statements are executed if none of the expressions above are true.

When the statement executes, *expression_1* is tested. If *expression_1* is true, the block of statements that immediately follows is executed, and the rest of the structure is ignored. If *expression_1* is false, however, the program jumps to the very next *else if* clause and tests *expression_2*. If it is true, the block of statements that immediately follows is executed, and then the rest of the structure is ignored. This process continues, from the top of the structure to the bottom, until one of the expressions is found to be true. If none of the expressions are true, the last *else* clause takes over, and the block of statements immediately following it is executed.

The last *else* clause, which does not have an *if* statement following it, is referred to as the *trailing else*. The trailing *else* is optional, but in most cases you will use it.



NOTE: The general format shows braces surrounding each block of conditionally executed statements. As with other forms of the *if* statement, the braces are required only when more than one statement is conditionally executed.

Program 4-13 shows an example of the *if/else if* statement. This program is a modification of Program 4-12, which appears in the previous *In the Spotlight* section.

Program 4-13

```
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     // Constants for grade thresholds
9     const int A_SCORE = 90,
10        B_SCORE = 80,
11        C_SCORE = 70,
12        D_SCORE = 60;
13
14     int testScore; // To hold a numeric test score
15
16     // Get the numeric test score.
17     cout << "Enter your numeric test score and I will\n"
18         << "tell you the letter grade you earned: ";
19     cin >> testScore;
20
21     // Determine the letter grade.
22     if (testScore >= A_SCORE)
23         cout << "Your grade is A.\n";
24     else if (testScore >= B_SCORE)
25         cout << "Your grade is B.\n";
26     else if (testScore >= C_SCORE)
27         cout << "Your grade is C.\n";
28     else if (testScore >= D_SCORE)
29         cout << "Your grade is D.\n";
30     else
31         cout << "Your grade is F.\n";
```

(program continues)

Program 4-13 (continued)

```
33     return 0;
34 }
```

Program Output with Example Input Shown in Bold

Enter your numeric test score and I will tell you the letter grade you earned: **78** **Enter**
Your grade is C.

Program Output with Different Example Input Shown in Bold

Enter your numeric test score and I will tell you the letter grade you earned: **84** **Enter**
Your grade is B.

Let's analyze how the if/else if statement in lines 22 through 31 works. First, the expression `testScore >= A_SCORE` is tested in line 22:

```
→ if (testScore >= A_SCORE)
    cout << "Your grade is A.\n";
else if (testScore >= B_SCORE)
    cout << "Your grade is B.\n";
else if (testScore >= C_SCORE)
    cout << "Your grade is C.\n";
else if (testScore >= D_SCORE)
    cout << "Your grade is D.\n";
else
    cout << "Your grade is F.\n";
```

If `testScore` is greater than or equal to 90, the string "Your grade is A.\n" is displayed and the rest of the if/else if statement is skipped. If `testScore` is not greater than or equal to 90, the else clause in line 24 takes over and causes the next if statement to be executed:

```
if (testScore >= A_SCORE)
    cout << "Your grade is A.\n";
→ else if (testScore >= B_SCORE)
    cout << "Your grade is B.\n";
else if (testScore >= C_SCORE)
    cout << "Your grade is C.\n";
else if (testScore >= D_SCORE)
    cout << "Your grade is D.\n";
else
    cout << "Your grade is F.\n";
```

The first if statement handles all of the grades greater than or equal to 90, so when this if statement executes, `testScore` will have a value of 89 or less. If `testScore` is greater than or equal to 80, the string "Your grade is B.\n" is displayed and the rest of the if/else if statement is skipped. This chain of events continues until one of the expressions is found to be true, or the last else clause at the end of the statement is encountered.

Notice the alignment and indentation that is used with the if/else if statement: The starting if clause, the else if clauses, and the trailing else clause are all aligned, and the conditionally executed statements are indented.

Using the Trailing else to Catch Errors

The trailing `else` clause, which appears at the end of the `if/else if` statement, is optional, but in many situations you will use it to catch errors. For example, Program 4-13 will assign a grade to any number that is entered as the test score, including negative numbers. If a negative test score is entered, however, the user has probably made a mistake. We can modify the code as shown in Program 4-14 so the trailing `else` clause catches any test score that is less than 0 and displays an error message.

Program 4-14

```
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     // Constants for grade thresholds
9     const int A_SCORE = 90,
10        B_SCORE = 80,
11        C_SCORE = 70,
12        D_SCORE = 60;
13
14     int testScore; // To hold a numeric test score
15
16     // Get the numeric test score.
17     cout << "Enter your numeric test score and I will\n"
18         << "tell you the letter grade you earned: ";
19     cin >> testScore;
20
21     // Determine the letter grade.
22     if (testScore >= A_SCORE)
23         cout << "Your grade is A.\n";
24     else if (testScore >= B_SCORE)
25         cout << "Your grade is B.\n";
26     else if (testScore >= C_SCORE)
27         cout << "Your grade is C.\n";
28     else if (testScore >= D_SCORE)
29         cout << "Your grade is D.\n";
30     else if (testScore >= 0)
31         cout << "Your grade is F.\n";
32     else
33         cout << "Invalid test score.\n";
34
35     return 0;
36 }
```

Program Output with Example Input Shown in Bold

Enter your numeric test score and I will
tell you the letter grade you earned: **-1**

Invalid test score.

The if/else if Statement Compared to a Nested Decision Structure

You never have to use the `if/else if` statement because its logic can be coded with nested `if/else` statements. However, a long series of nested `if/else` statements has two particular disadvantages when you are debugging code:

- The code can grow complex and become difficult to understand.
- Because indenting is important in nested statements, a long series of nested `if/else` statements can become too long to be displayed on the computer screen without horizontal scrolling. Also, long statements tend to “wrap around” when printed on paper, making the code even more difficult to read.

The logic of an `if/else if` statement is usually easier to follow than that of a long series of nested `if/else` statements. And, because all of the clauses are aligned in an `if/else if` statement, the lengths of the lines in the statement tend to be shorter.



Checkpoint

4.16 What will the following code 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;
```

4.17 The following code is used in a bookstore program to determine how many discount coupons a customer gets. Complete the table that appears after the program.

```
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;
```

If the customer purchases
this many books

This many coupons are given.

1
3
4
5
10

4.7

Flags

CONCEPT: A flag is a Boolean or integer variable that signals when a condition exists.

A *flag* is typically a `bool` variable that signals when some condition exists in the program. When the flag variable is set to `false`, it indicates that the condition does not exist. When the flag variable is set to `true`, it means the condition does exist.

For example, suppose a program that calculates sales commissions has a `bool` variable, defined and initialized as shown here:

```
bool salesQuotaMet = false;
```

In the program, the `salesQuotaMet` variable is used as a flag to indicate whether a salesperson has met the sales quota. When we define the variable, we initialize it with `false` because we do not yet know if the salesperson has met the sales quota. Assuming a variable named `sales` holds the amount of sales, code similar to the following might be used to set the value of the `salesQuotaMet` variable:

```
if (sales >= QUOTA_AMOUNT)
    salesQuotaMet = true;
else
    salesQuotaMet = false;
```

As a result of this code, the `salesQuotaMet` variable can be used as a flag to indicate whether the sales quota has been met. Later in the program, we might test the flag in the following way:

```
if (salesQuotaMet)
    cout << "You have met your sales quota!\n";
```

This code displays *You have met your sales quota!* if the `bool` variable `salesQuotaMet` is `true`. Notice we did not have to use the `==` operator to explicitly compare the `salesQuotaMet` variable with the value `true`. This code is equivalent to the following:

```
if (salesQuotaMet == true)
    cout << "You have met your sales quota!\n";
```

Integer Flags

Integer variables may also be used as flags. This is because in C++ the value 0 is considered false, and any nonzero value is considered true. In the sales commission program previously described, we could define the `salesQuotaMet` variable with the following statement:

```
int salesQuotaMet = 0;           // 0 means false.
```

As before, we initialize the variable with 0 because we do not yet know if the sales quota has been met. After the sales have been calculated, we can use code similar to the following to set the value of the `salesQuotaMet` variable:

```
if (sales >= QUOTA_AMOUNT)
    salesQuotaMet = 1;
else
    salesQuotaMet = 0;
```

Later in the program, we might test the flag in the following way:

```
if (salesQuotaMet)
    cout << "You have met your sales quota!\n";
```

4.8

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
<code>&&</code>	AND	Connects two expressions into one. Both expressions must be true for the overall expression to be true.
<code> </code>	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.
<code>!</code>	NOT	The <code>!</code> operator 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 subexpressions 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.";
```

In the statement above, the two relational expressions are combined into a single 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.



TIP: You must provide complete expressions on both sides of the `&&` operator. For example, the following statement is not correct because the condition on the right side of the `&&` operator is not a complete expression:

```
temperature > 0 && < 100
```

The expression must be rewritten as

```
temperature > 0 && temperature < 100
```

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.

Table 4-7 Truth Table for the `&&` Operator

Expression	Value of Expression
true && false	false (0)
false && true	false (0)
false && false	false (0)
true && true	true (1)

As the table shows, both subexpressions must be true for the `&&` operator to return a true value.



NOTE: If the subexpression on the left side of an `&&` operator is false, the expression on the right side will not be checked. Since the entire expression is false if only one of the subexpressions 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 4-15 performs a similar operation as Program 4-11, which qualifies a bank customer for a special interest rate. This program uses a logical operator.

Program 4-15

```

1 // This program demonstrates the && logical operator.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     char employed,    // Currently employed, Y or N
8         recentGrad; // Recent graduate, Y or N
9
10    // Is the user employed and a recent graduate?
11    cout << "Answer the following questions\n";
12    cout << "with either Y for Yes or N for No.\n";
13

```

(program continues)

Program 4-15 (continued)

```

14     cout << "Are you employed? ";
15     cin >> employed;
16
17     cout << "Have you graduated from college "
18         << "in the past two years? ";
19     cin >> recentGrad;
20
21     // Determine the user's loan qualifications.
22     if (employed == 'Y' && recentGrad == 'Y')
23     {
24         cout << "You qualify for the special "
25             << "interest rate.\n";
26     }
27     else
28     {
29         cout << "You must be employed and have\n"
30             << "graduated from college in the\n"
31             << "past two years to qualify.\n";
32     }
33     return 0;
34 }
```

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**

Have you graduated from college in the past two years? **N**

You must be employed and have
graduated from college in the
past two years to qualify.

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**

Have you graduated from college in the past two years? **Y**

You must be employed and have
graduated from college in the
past two years to qualify.

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**

Have you graduated from college in the past two years? **Y**

You qualify for the special interest rate.

The message "You qualify for the special interest rate." is displayed only when both the expressions `employed == 'Y'` and `recentGrad == 'Y'` are true. If either of these is false, the message "You must be employed and have graduated from college in the past two years to qualify." is printed.



NOTE: Although it is similar, Program 4-15 is not the logical equivalent of Program 4-11. For example, Program 4-15 doesn't display the message "You must be employed to 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 subexpressions 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.



TIP: You must provide complete expressions on both sides of the `||` operator. For example, the following code is not correct because the condition on the right side of the `||` operator is not a complete expression:

```
temperature < 0 || > 100
```

The expression must be rewritten as

```
temperature < 0 || temperature > 100
```

Table 4-8 shows a truth table for the `||` operator.

Table 4-8 Truth Table for the `||` Operator

Expression	Value of the Expression
<code>true false</code>	<code>true</code> (1)
<code>false true</code>	<code>true</code> (1)
<code>false false</code>	<code>false</code> (0)
<code>true true</code>	<code>true</code> (1)

All it takes for an OR expression to be true is for one of the subexpressions to be true. It doesn't matter if the other subexpression is false or true.



NOTE: The `||` operator also performs short-circuit evaluation. If the subexpression on the left side of an `||` operator is true, the expression on the right side will not be checked. Since it's only necessary for one of the subexpressions to be true, it would waste CPU time to check the remaining expression.

Program 4-16 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 5 years.

Program 4-16

```

1 // This program demonstrates the logical || operator.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     // Constants for minimum income and years
8     const double MIN_INCOME = 35000.0;
9     const int MIN_YEARS = 5;
10
11    double income; // Annual income
12    int years; // Years at the current job
13
14    // Get the annual income
15    cout << "What is your annual income? ";
16    cin >> income;
17
18    // Get the number of years at the current job.
19    cout << "How many years have you worked at "
20        << "your current job? ";
21    cin >> years;
22
23    // Determine the user's loan qualifications.
24    if (income >= MIN_INCOME || years > MIN_YEARS)
25        cout << "You qualify.\n";
26    else
27    {
28        cout << "You must earn at least $"
29                    << MIN_INCOME << " or have been "
30                    << "employed more than " << MIN_YEARS
31                    << " years.\n";
32    }
33    return 0;
34 }
```

Program Output with Example Input Shown in Bold

What is your annual income? **40000**

How many years have you worked at your current job? **2**

You qualify.

Program Output with Example Input Shown in Bold

What is your annual income? **20000**

How many years have you worked at your current job? **7**

You qualify.

Program Output with Example Input Shown in Bold

What is your annual income? **30000**

How many years have you worked at your current job? **3**

You must earn at least \$35000 or have been employed more than 5 years.

The string “You qualify\n.” 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 Truth Table for the ! Operator

Expression	Value of the Expression
<code>!true</code>	<code>false (0)</code>
<code>!false</code>	<code>true (1)</code>

Program 4-17 performs the same task as Program 4-16. 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 5 years.

Program 4-17

```
1 // This program demonstrates the logical ! operator.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     // Constants for minimum income and years
8     const double MIN_INCOME = 35000.0;
9     const int MIN_YEARS = 5;
10
11    double income; // Annual income
12    int years;      // Years at the current job
13
14    // Get the annual income
15    cout << "What is your annual income? ";
16    cin >> income;
17
18    // Get the number of years at the current job.
19    cout << "How many years have you worked at "
20        << "your current job? ";
```

(program continues)

Program 4-17

(continued)

```

21     cin >> years;
22
23     // Determine the user's loan qualifications.
24     if (!(income >= MIN_INCOME || years > MIN_YEARS))
25     {
26         cout << "You must earn at least $"
27             << MIN_INCOME << " or have been "
28             << "employed more than " << MIN_YEARS
29             << " years.\n";
30     }
31     else
32         cout << "You qualify.\n";
33     return 0;
34 }
```

The output of Program 4-17 is the same as Program 4-16.

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

Logical Operators in Order of Precedence

!
&&

The ! operator has a higher precedence than many of the C++ operators. To avoid an error, you should always 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 a catastrophe like this, always use parentheses!

The `&&` and `||` operators rank lower in precedence than the relational operators, so precedence problems are less likely to occur. If you feel unsure, however, it doesn't hurt to use parentheses anyway.

`(a > b) && (x < y)` is the same as `a > b && x < y`
`(x == y) || (b > a)` is the same as `x == y || b > a`

The logical operators have left-to-right associativity. In the following expression, `a < b` is evaluated before `y == z`.

```
a < b || y == z
```

In the following expression, `y == z` is evaluated first, however, because the `&&` operator has higher precedence than `||`.

```
a < b || y == z && m > j
```

The expression is equivalent to

```
(a < b) || ((y == z) && (m > j))
```

4.9

Checking Numeric Ranges with Logical Operators

CONCEPT: Logical operators are effective for determining whether a number is in or out of a range.

When determining whether 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 whether 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. `x` must be within the range of 20 through 40 for this expression to be true.

When determining whether a number is outside a range, the `||` operator is best to use. The following statement determines whether `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` cannot 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

- 4.18 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)
<code>true && false</code>	False
<code>true && true</code>	True
<code>false && true</code>	False
<code>false && false</code>	False
<code>true false</code>	True
<code>true true</code>	True
<code>false true</code>	True
<code>false false</code>	False
<code>!true</code>	False
<code>!false</code>	True

- 4.19 Assume the variables `a = 2`, `b = 4`, and `c = 6`. Determine 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)`
- 4.20 Write an `if` statement that prints the message “The number is valid” if the variable `speed` is within the range 0 through 200.
- 4.21 Write an `if` statement that prints the message “The number is not valid” if the variable `speed` is outside the range 0 through 200.

4.10 Menus

CONCEPT: You can use nested `if/else` statements or the `if/else if` statement to create menu-driven programs. A *menu-driven* program allows the user to determine the course of action by selecting it from a list of actions.

A menu is a screen displaying a set of choices from which the user selects. For example, a program that manages 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. Nested `if/else` statements or an `if/else if` structure can be used to set up such a menu. After the user enters a number, the program compares the number with the available selections and executes the statements that perform that operation.

Program 4-18 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 then calculates the cost of the membership.

Program 4-18

```
1 // This program displays a menu and asks the user to make a
2 // selection. An if/else if statement determines which item
3 // the user has chosen.
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;
7
8 int main()
9 {
10     int choice;      // To hold a menu choice
11     int months;      // To hold the number of months
12     double charges; // To hold the monthly charges
13
14     // Constants for membership rates
15     const double ADULT = 40.0,
16                     SENIOR = 30.0,
17                     CHILD = 20.0;
18
19     // Constants for menu choices
20     const int ADULT_CHOICE = 1,
21             CHILD_CHOICE = 2,
22             SENIOR_CHOICE = 3,
23             QUIT_CHOICE = 4;
24
25     // Display the menu and get a choice.
26     cout << "\t\tHealth Club Membership Menu\n\n"
27         << "1. Standard Adult Membership\n"
28         << "2. Child Membership\n"
29         << "3. Senior Citizen Membership\n"
30         << "4. Quit the Program\n\n"
31         << "Enter your choice: ";
32     cin >> choice;
33
34     // Set the numeric output formatting.
35     cout << fixed << showpoint << setprecision(2);
36
37     // Respond to the user's menu selection.
38     if (choice == ADULT_CHOICE)
39     {
40         cout << "For how many months? ";
41         cin >> months;
42         charges = months * ADULT;
43         cout << "The total charges are $" << charges << endl;
44     }
45     else if (choice == CHILD_CHOICE)
46     {
47         cout << "For how many months? ";
```

(program continues)

Program 4-18 *(continued)*

```

48     cin >> months;
49     charges = months * CHILD;
50     cout << "The total charges are $" << charges << endl;
51 }
52 else if (choice == SENIOR_CHOICE)
53 {
54     cout << "For how many months? ";
55     cin >> months;
56     charges = months * SENIOR;
57     cout << "The total charges are $" << charges << endl;
58 }
59 else if (choice == QUIT_CHOICE)
60 {
61     cout << "Program ending.\n";
62 }
63 else
64 {
65     cout << "The valid choices are 1 through 4. Run the\n"
66         << "program again and select one of those.\n";
67 }
68 return 0;
69 }
```

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**

For how many months? **6**

The total charges are \$180.00

Let's take a closer look at the program:

- Lines 10–12 define the following variables:
 - The choice variable will hold the user's menu choice.
 - The months variable will hold the number of months of health club membership.
 - The charges variable will hold the total charges.
- Lines 15–17 define named constants for the monthly membership rates for adult, senior citizen, and child memberships.
- Lines 20–23 define named constants for the menu choices.
- Lines 26–32 display the menu and get the user's choice.
- Line 35 sets the numeric output formatting for floating-point numbers.
- Lines 38–67 are an if/else if statement that determines the user's menu choice in the following manner:
 - If the user selected 1 from the menu (adult membership), the statements in lines 40–43 are executed.

- Otherwise, if the user selected 2 from the menu (child membership), the statements in lines 47–50 are executed.
- Otherwise, if the user selected 3 from the menu (senior citizen membership), the statements in lines 54–57 are executed.
- Otherwise, if the user selected 4 from the menu (quit the program), the statement in line 61 is executed.
- If the user entered any choice other than 1, 2, 3, or 4, the `else` clause in lines 63–67 executes, displaying an error message.

4.11

Focus on Software Engineering: 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 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. 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 other sets 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 4-19 is a test scoring program that rejects any test score less than 0 or greater than 100.

Program 4-19

```
1 // This test scoring program does not accept test scores
2 // that are less than 0 or greater than 100.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     // Constants for grade thresholds
```

(program continues)

Program 4-19 (continued)

```

9     const int A_SCORE = 90,
10    B_SCORE = 80,
11    C_SCORE = 70,
12    D_SCORE = 60,
13    MIN_SCORE = 0,      // Minimum valid score
14    MAX_SCORE = 100;    // Maximum valid score
15
16    int testScore; // To hold a numeric test score
17
18    // Get the numeric test score.
19    cout << "Enter your numeric test score and I will\n"
20        << "tell you the letter grade you earned: ";
21    cin >> testScore;
22
23    // Validate the input and determine the grade.
24    if (testScore >= MIN_SCORE && testScore <= MAX_SCORE)
25    {
26        // Determine the letter grade.
27        if (testScore >= A_SCORE)
28            cout << "Your grade is A.\n";
29        else if (testScore >= B_SCORE)
30            cout << "Your grade is B.\n";
31        else if (testScore >= C_SCORE)
32            cout << "Your grade is C.\n";
33        else if (testScore >= D_SCORE)
34            cout << "Your grade is D.\n";
35        else
36            cout << "Your grade is F.\n";
37    }
38    else
39    {
40        // An invalid score was entered.
41        cout << "That is an invalid score. Run the program\n"
42            << "again and enter a value in the range of\n"
43            << MIN_SCORE << " through " << MAX_SCORE << ".\n";
44    }
45
46    return 0;
47 }
```

Program Output with Example Input Shown in Bold

Enter your numeric test score and I will
 tell you the letter grade you earned: **-1**
 That is an invalid score. Run the program
 again and enter a value in the range of
 0 through 100.

Program Output with Example Input Shown in Bold

Enter your numeric test score and I will
 tell you the letter grade you earned: **81**
 Your grade is B.

4.12 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 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 ASCII codes than uppercase letters, so 'a' > 'Z'. Program 4-20 shows how characters can be compared with relational operators.

Program 4-20

```
1 // This program demonstrates how characters can be
2 // compared with the relational operators.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     char ch;
```

(program continues)

Program 4-20 (continued)

```

10     // Get a character from the user.
11     cout << "Enter a digit or a letter: ";
12     ch = cin.get();
13
14     // Determine what the user entered.
15     if (ch >= '0' && ch <= '9')
16         cout << "You entered a digit.\n";
17     else if (ch >= 'A' && ch <= 'Z')
18         cout << "You entered an uppercase letter.\n";
19     else if (ch >= 'a' && ch <= 'z')
20         cout << "You entered a lowercase letter.\n";
21     else
22         cout << "That is not a digit or a letter.\n";
23
24     return 0;
25 }
```

Program Output with Example Input Shown in Bold

Enter a digit or a letter: **t**
You entered a lowercase letter.

Program Output with Example Input Shown in Bold

Enter a digit or a letter: **v**
You entered an uppercase letter.

Program Output with Example Input Shown in Bold

Enter a digit or a letter: **5**
You entered a digit.

Program Output with Example Input Shown in Bold

Enter a digit or a letter: **&**
That is not a digit or a letter.

Comparing string Objects

string objects can also be compared with relational operators. As with individual characters, when two string objects are compared, it is actually the ASCII values of the characters making up the strings that are being compared. For example, assume the following definitions exist in a program:

```
string str1 = "ABC";
string str2 = "XYZ";
```

The string object str1 is considered less than the string object str2 because the characters "ABC" alphabetically precede (have lower ASCII values than) the characters "XYZ". So, the following if statement will cause the message "str1 is less than str2." to be displayed on the screen.

```
if (str1 < str2)
    cout << "str1 is less than str2.;"
```

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 `string` objects 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 literals:

```
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 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, // Price for part A
11                     PRICE_B = 199.0; // Price for part B
12
13     string partNum;           // Holds a part number
14
15     // Display available parts and get the user's selection
16     cout << "The headphone part numbers are:\n"
17         << "Noise canceling: part number S-29A\n"
18         << "Wireless: part number S-29B\n"
19         << "Enter the part number of the headphones you\n"
```

(program continues)

Program 4-21 (continued)

```

20      << "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 headphone part numbers are:

Noise canceling: part number S-29A

Wireless: part number S-29B

Enter the part number of the headphones you

wish to purchase: **S-29A** **Enter**

The price is \$249.00

**Checkpoint**

- 4.22 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

- 4.23 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"

4.13 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;
```

The statement above is called a *conditional expression* and consists of three subexpressions separated by the ? and : symbols. The expressions are `x < 0`, `y = 10`, and `z = 20`, as illustrated here:

```
x < 0 ? y = 10 : z = 20;
```



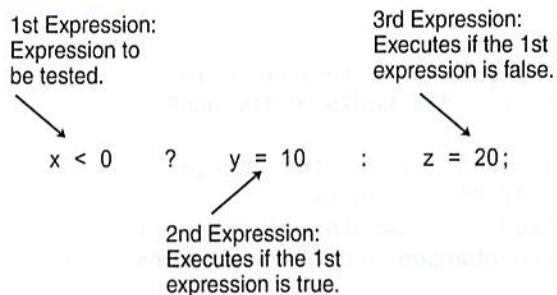
NOTE: Since it takes three operands, the conditional operator is considered a *ternary* operator.

The conditional expression above performs the same operation as the following `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 expression to be tested. It's like the expression in the parentheses of an `if` statement. If the expression is true, the part of the statement between the ? and the : is executed. Otherwise, the part after the : is executed. Figure 4-10 illustrates the roles played by the three subexpressions.

Figure 4-10 Conditional expression



If it helps, you can put parentheses around the subexpressions, as in the following:

```
(x < 0) ? (y = 10) : (z = 20);
```

Using the Value of a Conditional Expression

Remember, in C++ all expressions have a value, and this includes the conditional expression. If the first subexpression is true, the value of the conditional expression is the value of the second subexpression. Otherwise, it is the value of the third subexpression. Here is an example of an assignment statement using the value of a conditional expression:

```
a = x > 100 ? 0 : 1;
```

The value assigned to a will be either 0 or 1, depending upon whether x is greater than 100. This statement could be expressed 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 5 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 calculates a consultant's charges at $50
2 // per hour, for a minimum of 5 hours. The ?: operator
3 // adjusts hours to 5 if less than 5 hours were worked.
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;
7
8 int main()
9 {
10     const double PAY_RATE = 50.0; // Hourly pay rate
11     const int MIN_HOURS = 5;      // Minimum billable hours
12     double hours,                // Hours worked
13         charges;                // Total charges
14
15     // Get the hours worked.
16     cout << "How many hours were worked? ";
17     cin >> hours;
18
19     // Determine the hours to charge for.
20     hours = hours < MIN_HOURS ? MIN_HOURS : hours;
21
22     // Calculate and display the charges.
23     charges = PAY_RATE * hours;
24     cout << fixed << showpoint << setprecision(2)
25         << "The charges are $" << charges << endl;
26
27 }
```

Program Output with Example Input Shown in Bold

How many hours were worked? **10** Enter
The charges are \$500.00

Program Output with Example Input Shown in Bold

How many hours were worked? **2** Enter
The charges are \$250.00

Notice in line 11 a constant named MIN_HOURS is defined to represent the minimum number of hours, which is 5. Here is the statement in line 20, with the conditional expression:

```
hours = hours < MIN_HOURS ? MIN_HOURS : 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. The hours variable 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, the statement above 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

4.24 Rewrite the following if/else statements as conditional expressions:

- if ($x > y$)


```
z = 1;
else
    z = 20;
```
- if ($temp > 45$)


```
population = base * 10;
else
    population = base * 2;
```
- if ($hours > 40$)


```
wages *= 1.5;
else
    wages *= 1;
```
- if ($result \geq 0$)


```
cout << "The result is positive\n";
else
    cout << "The result is negative.\n";
```

- 4.25 The following statements use conditional expressions. Rewrite each with an *if/else* statement.

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");`

- 4.26 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;
}
```

4.14 The switch Statement

CONCEPT: The *switch* statement lets the value of a variable or an expression determine where the program will branch.

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. However, it tests the value of an integer expression and then uses that value to determine to which set of statements to branch. 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), followed by a colon. The constant expression may be an integer literal or an integer named constant. The `case` statement marks the beginning of a section of statements. The program branches to these statements if the value of the `switch` expression matches that of the `case` expression.



WARNING! The expression of each `case` statement in the block must be unique.



NOTE: The expression following the word `case` must be an integer literal or constant. It cannot be a variable, and it cannot be an expression such as `x < 22` or `n == 50`.

An optional `default` section comes after all the `case` statements. The program branches to this section if none of the `case` expressions match the `switch` expression. So, 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 // The switch statement in this program tells the user something  
2 // he or she already knows: the data just 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    switch (choice)  
13    {  
14        case 'A': cout << "You entered A.\n";  
15            break;  
16        case 'B': cout << "You entered B.\n";  
17            break;  
18        case 'C': cout << "You entered C.\n";  
19            break;  
20        default: cout << "You did not enter A, B, or C!\n";  
21    }  
22    return 0;  
23 }
```

(program continues)

Program 4-23

(continued)

Program Output with Example Input Shown in BoldEnter A, B, or C: **B**

You entered B.

Program Output with Example Input Shown in BoldEnter A, B, or C: **F**

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 literals are considered integers.) The default section is branched to if the user enters anything other than A, B, or C.

Notice the break statements in 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 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, without the break statements.

Program 4-24

```
1 // The switch statement in this program tells the user something
2 // he or she already knows: the data just 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     // The following switch is
13     // 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 Output with Example Input Shown in Bold

Enter A, B, or C: **A**
You entered A.
You entered B.
You entered C.
You did not enter A, B, or C!

Program Output with Example Input Shown in Bold

Enter A, B, or C: **C**
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; // Model number
9
10    // Get a model number from the user.
11    cout << "Our TVs come in three models:\n";
12    cout << "The 100, 200, and 300. Which do you want? ";
13    cin >> modelNum;
```

(program continues)

Program 4-25 (continued)

```

15     // Display the model's features.
16     cout << "That model has the following features:\n";
17     switch (modelNum)
18     {
19         case 300: cout << "\tPicture-in-a-picture.\n";
20         case 200: cout << "\tStereo sound.\n";
21         case 100: cout << "\tRemote control.\n";
22             break;
23         default: cout << "You can only choose the 100,";
24             cout << "200, or 300.\n";
25     }
26     return 0;
27 }
```

Program Output with Example Input Shown in Bold

Our TVs come in three models:

The 100, 200, and 300. Which do you want? **100**

That model has the following features:

 Remote control.

Program Output with Example Input Shown in Bold

Our TVs come in three models:

The 100, 200, and 300. Which do you want? **200**

That model has the following features:

 Stereo sound.

 Remote control.

Program Output with Example Input Shown in Bold

Our TVs come in three models:

The 100, 200, and 300. Which do you want? **300**

That model has the following features:

 Picture-in-a-picture.

 Stereo sound.

 Remote control.

Program Output with Example Input Shown in Bold

Our TVs come in three models:

The 100, 200, and 300. Which do you want? **500**

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 pet food. The available choices are A, B, and C. The switch statement will recognize either uppercase or lowercase letters.

Program 4-26

```

1 // The switch statement in this program uses the "fall through"
2 // feature to catch both uppercase and lowercase letters entered
3 // by the user.
4 #include <iostream>
```

```
5  using namespace std;
6
7  int main()
8  {
9      char feedGrade;
10
11     // Get the desired grade of feed.
12     cout << "Our pet food is available in three grades:\n";
13     cout << "A, B, and C. Which do you want pricing for? ";
14     cin >> feedGrade;
15
16     // Display the price.
17     switch(feedGrade)
18     {
19         case 'a':
20             case 'A': cout << "30 cents per pound.\n";
21                 break;
22         case 'b':
23             case 'B': cout << "20 cents per pound.\n";
24                 break;
25         case 'c':
26             case 'C': cout << "15 cents per pound.\n";
27                 break;
28         default: cout << "That is an invalid choice.\n";
29     }
30     return 0;
31 }
```

Program Output with Example Input Shown in Bold

Our pet food is available in three grades:
A, B, and C. Which do you want pricing for? **b**
20 cents per pound.

Program Output with Example Input Shown in Bold

Our pet food is available in three grades:
A, B, and C. Which do you want pricing for? **B**
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 technique is used for 'b' and 'c'.

Using switch in Menu Systems

The switch statement is a natural mechanism for building menu systems. Recall that Program 4-18 gives a menu to select which health club package the user wishes to purchase. The program uses if/else if statements to determine which package the user has selected and displays the calculated charges. Program 4-27 is a modification of that program, using a switch statement instead of if/else if.

Program 4-27

```
1 // This program uses a switch statement to determine
2 // the item selected from a menu.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     int choice;      // To hold a menu choice
10    int months;     // To hold the number of months
11    double charges; // To hold the monthly charges
12
13    // Constants for membership rates
14    const double ADULT = 40.0,
15                  CHILD = 20.0,
16                  SENIOR = 30.0;
17
18    // Constants for menu choices
19    const int ADULT_CHOICE = 1,
20                  CHILD_CHOICE = 2,
21                  SENIOR_CHOICE = 3,
22                  QUIT_CHOICE = 4;
23
24    // Display the menu and get a choice.
25    cout << "\t\tHealth Club Membership Menu\n\n"
26        << "1. Standard Adult Membership\n"
27        << "2. Child Membership\n"
28        << "3. Senior Citizen Membership\n"
29        << "4. Quit the Program\n\n"
30        << "Enter your choice: ";
31    cin >> choice;
32
33    // Set the numeric output formatting.
34    cout << fixed << showpoint << setprecision(2);
35
36    // Respond to the user's menu selection.
37    switch (choice)
38    {
39        case ADULT_CHOICE:
40            cout << "For how many months? ";
41            cin >> months;
42            charges = months * ADULT;
43            cout << "The total charges are $" << charges << endl;
44            break;
45
46        case CHILD_CHOICE:
47            cout << "For how many months? ";
48            cin >> months;
49            charges = months * CHILD;
50            cout << "The total charges are $" << charges << endl;
51            break;
52    }
```

```

53     case SENIOR_CHOICE:
54         cout << "For how many months? ";
55         cin >> months;
56         charges = months * SENIOR;
57         cout << "The total charges are $" << charges << endl;
58         break;
59
60     case QUIT_CHOICE:
61         cout << "Program ending.\n";
62         break;
63
64     default:
65         cout << "The valid choices are 1 through 4. Run the\n"
66             << "program again and select one of those.\n";
67     }
68
69     return 0;
70 }
```

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**

For how many months? **6**

The total charges are \$120.00



Checkpoint

- 4.27 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;
```

- 4.28 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;
}
```

- 4.29 What will the following program display?

```
#include <iostream>
using namespace std;
int main()
{
    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;
    }
    return 0;
}
```

- 4.30 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;
}
```

- 4.31 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;
}
```

4.15 More about Blocks and Variable 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. Program 4-28 is a modification of Program 4-17, 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 4-28

```
1 // This program demonstrates late variable definition
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     // Constants for minimum income and years
8     const double MIN_INCOME = 35000.0;
9     const int MIN_YEARS = 5;
10
11    // Get the annual income.
12    cout << "What is your annual income? ";
13    double income; // Variable definition
14    cin >> income;
15
16    // Get the number of years at the current job.
17    cout << "How many years have you worked at "
18        << "your current job? ";
19    int years; // Variable definition
20    cin >> years;
21
22    // Determine the user's loan qualifications.
23    if (income >= MIN_INCOME || years > MIN_YEARS)
24        cout << "You qualify.\n";
25    else
26    {
27        cout << "You must earn at least $"
28            << MIN_INCOME << " or have been "
29            << "employed more than " << MIN_YEARS
30            << " years.\n";
31    }
32    return 0;
33 }
```

It is a common practice to define all of a function's variables at the top of the function. Sometimes, especially in longer programs, it's a good idea to define variables near the part of the program where they are used. This makes the purpose of the variable more evident.

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 4-28, the scope of the `income` variable is the part of the program in lines 13 through 32. The scope of the `years` variable is the part of the program in lines 19 through 32.

The variables `income` and `years` are defined inside function `main`'s braces. Variables defined inside a set of braces 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 4-29. This version of the loan program has the variable `years` defined inside the block of the `if` statement. The scope of `years` is the part of the program in lines 21 through 31.

Program 4-29

```

1 // This program demonstrates a variable defined in an inner block.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     // Constants for minimum income and years
8     const double MIN_INCOME = 35000.0;
9     const int MIN_YEARS = 5;
10
11    // Get the annual income.
12    cout << "What is your annual income? ";
13    double income; // Variable definition
14    cin >> income;
15
16    if (income >= MIN_INCOME)
17    {
18        // Get the number of years at the current job.
19        cout << "How many years have you worked at "
20            << "your current job? ";
21        int years; // Variable definition
22        cin >> years;
23
24        if (years > MIN_YEARS)
25            cout << "You qualify.\n";
26        else
27        {
28            cout << "You must have been employed for\n"
29                << "more than " << MIN_YEARS
30                << " years to qualify.\n";
31        }
32    }
33 }
```

```

34     {
35         cout << "You must earn at least $" << MIN_INCOME
36             << " to qualify.\n";
37     }
38     return 0;
39 }
```

Notice the scope of `years` is only within the block where 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*, or *goes out of 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 4-30.

Program 4-30

```

1 // This program uses two variables with the name number.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     // Define a variable named number.
8     int number;
9
10    cout << "Enter a number greater than 0: ";
11    cin >> number;
12    if (number > 0)
13    {
14        int number; // Another variable named number.
15        cout << "Now enter another number: ";
16        cin >> number;
17        cout << "The second number you entered was "
18             << 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**

Now enter another number: **7**

The second number you entered was **7**

Your first number was **2**

Program 4-30 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.

Case Study: See the Sales Commission Case Study on the computer science portal at www.pearsonhighered.com/gaddis.

Review Questions and Exercises

Short Answer

1. Describe the difference between the `if/else if` statement and a series of `if` statements.
2. In an `if/else if` statement, what is the purpose of a trailing `else`?
3. What is a flag and how does it work?
4. Can an `if` statement test expressions other than relational expressions? Explain.
5. Briefly describe how the `&&` operator works.
6. Briefly describe how the `||` operator works.
7. Why are the relational operators called relational?
8. Why do most programmers indent the conditionally executed statements in a decision structure?

Fill-in-the-Blank

9. An expression using the greater-than, less-than, greater-than-or-equal-to, less-than-or-equal-to, equal-to, or not-equal-to operator is called a(n) _____ expression.
10. A relational expression is either _____ or _____.
11. The value of a relational expression is 0 if the expression is _____ or 1 if the expression is _____.
12. The `if` statement regards an expression with the value 0 as _____.
13. The `if` statement regards an expression with a nonzero value as _____.
14. For an `if` statement to conditionally execute a group of statements, the statements must be enclosed in a set of _____.
15. In an `if/else` statement, the `if` part executes its statement or block if the expression is _____, and the `else` part executes its statement or block if the expression is _____.
16. The trailing `else` in an `if/else if` statement has a similar purpose as the _____ section of a `switch` statement.

17. The `if/else if` statement is actually a form of the _____ `if` statement.
18. If the subexpression on the left of the _____ logical operator is false, the right subexpression is not checked.
19. If the subexpression on the left of the _____ logical operator is true, the right subexpression is not checked.
20. The _____ logical operator has higher precedence than the other logical operators.
21. The logical operators have _____ associativity.
22. The _____ logical operator works best when testing a number to determine if it is within a range.
23. The _____ logical operator works best when testing a number to determine if it is outside a range.
24. A variable with _____ scope is only visible when the program is executing in the block containing the variable's definition.
25. You use the _____ operator to determine whether one `string` object is greater than another `string` object.
26. An expression using the conditional operator is called a(n) _____ expression.
27. The expression that is tested by a `switch` statement must have a(n) _____ value.
28. The expression following a `case` statement must be a(n) _____.
29. A program will "fall through" a `case` section if it is missing the _____ statement.
30. 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);` _____

Algorithm Workbench

31. Write an `if` statement that assigns 100 to `x` when `y` is equal to 0.
32. Write an `if/else` statement that assigns 0 to `x` when `y` is equal to 10. Otherwise, it should assign 1 to `x`.
33. Using the following chart, write an `if/else if` 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%

34. Write an `if` statement that sets the variable `hours` to 10 when the flag variable `minimum` is set.
35. Write nested `if` statements that perform the following tests: If `amount1` is greater than 10 and `amount2` is less than 100, display the greater of the two.

36. Write an `if` statement that prints the message “The number is valid” if the variable `grade` is within the range 0 through 100.
37. Write an `if` statement that prints the message “The number is valid” if the variable `temperature` is within the range -50 through 150.
38. Write an `if` statement that prints the message “The number is not valid” if the variable `hours` is outside the range 0 through 80.
39. Assume `str1` and `str2` are string objects that have been initialized with different values. Write an `if/else` statement that compares the two objects and displays the one that is alphabetically greatest.
40. 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);
}
```

41. Match the conditional expression with the `if/else` statement that performs the same operation.
- A) `q = x < y ? a + b : x * 2;`
- B) `q = x < y ? x * 2 : a + b;`
- C) `x < y ? q = 0 : q = 1;`
`____ if (x < y)`
`q = 0;`
`else`
`q = 1;`
`____ if (x < y)`
`q = a + b;`
`else`
`q = x * 2;`
`____ if (x < y)`
`q = x * 2;`
`else`
`q = a + b;`

True or False

42. T F The `=` operator and the `==` operator perform the same operation when used in a Boolean expression.

43. T F A variable defined in an inner block may not have the same name as a variable defined in the outer block.
44. T F A conditionally executed statement should be indented one level from the `if` statement.
45. T F All lines in a block should be indented one level.
46. T F It's safe to assume that all uninitialized variables automatically start with 0 as their value.
47. T F When an `if` statement is nested in the `if` part of another statement, the only time the inner `if` is executed is when the expression of the outer `if` is true.
48. T F When an `if` statement is nested in the `else` part of another statement, as in an `if/else if`, the only time the inner `if` is executed is when the expression of the outer `if` is true.
49. T F The scope of a variable is limited to the block in which it is defined.
50. T F You can use the relational operators to compare `string` objects.
51. T F `x != y` is the same as `(x > y || x < y)`
52. T F `y < x` is the same as `x >= y`
53. T F `x >= y` is the same as `(x > y && x = y)`

Assume the variables `x = 5`, `y = 6`, and `z = 8`. Indicate by circling the T or F whether each of the following conditions is True or False:

54. T F `x == 5 || y > 3`
55. T F `7 <= x && z > 4`
56. T F `2 != y && z != y`
57. T F `x >= 0 || x <= y`

Find the Errors

Each of the following programs has errors. Find as many as you can.

```
58. // This program averages 3 test scores.  
// It uses the variable perfectScore as a flag.  
include <iostream>  
using namespace std;  
  
int main()  
{  
    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";  
    return 0;  
}
```

```
59. // This program divides a user-supplied number by another
// user-supplied number. It checks for division by zero.
#include <iostream>
using namespace std;

int main()
{
    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;
    return 0;
}

60. // This program uses an if/else if statement to assign a
// letter grade (A, B, C, D, or F) to a numeric test score.
#include <iostream>
using namespace std;

int main()
{
    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";
    return 0;
}
```

61. // This program uses a switch-case statement to assign a
// letter grade (A, B, C, D, or F) to a numeric test score.
- ```
#include <iostream>
using namespace std;

int main()
{
 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";
 }
 return 0;
}
```
62. The following statement should determine if *x* is not greater than 20. What is wrong with it?
- ```
if (!x > 20)
```
63. 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)
```
64. 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)
```
65. The following statement should assign 0 to *z* if *a* is less than 10, otherwise it should assign 7 to *z*. What is wrong with it?
- ```
z = (a < 10) : 0 ? 7;
```

## 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. 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 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 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.

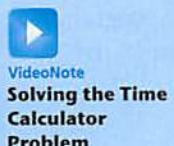
### 6. Mass and Weight

Scientists measure an object's mass in kilograms and its weight in newtons. If you know the amount of mass that an object has, 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, then calculates and displays its weight. If the object weighs more than 1,000 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



Write a program that asks the user to enter a number of seconds.

- There are 60 seconds in a minute. If the number of seconds entered by the user is greater than or equal to 60, the program should display the number of minutes in that many seconds.
- There are 3,600 seconds in an hour. If the number of seconds entered by the user is greater than or equal to 3,600, the program should display the number of hours in that many seconds.
- There are 86,400 seconds in a day. If the number of seconds entered by the user is greater than or equal to 86,400, the program should display the number of days in that many seconds.

## 8. Color Mixer

The colors red, blue, and yellow are known as primary colors because they cannot be made by mixing other colors. When you mix two primary colors, you get a secondary color, as shown here:

When you mix red and blue, you get purple.

When you mix red and yellow, you get orange.

When you mix blue and yellow, you get green.

Write a program that prompts the user to enter the names of two primary colors to mix. If the user enters anything other than “red,” “blue,” or “yellow,” the program should display an error message. Otherwise, the program should display the name of the secondary color that results by mixing two primary colors.

## 9. Change for a Dollar Game

Create a change-counting game that gets the user to enter the number of coins required to make exactly one dollar. The program should ask the user to enter the number of pennies, nickels, dimes, and quarters. If the total value of the coins entered is equal to one dollar, the program should congratulate the user for winning the game. Otherwise, the program should display a message indicating whether the amount entered was more than or less than one dollar.

## 10. Days in a Month

Write a program that asks the user to enter the month (letting the user enter an integer in the range of 1 through 12) and the year. The program should then display the number of days in that month. Use the following criteria to identify leap years:

1. Determine whether the year is divisible by 100. If it is, then it is a leap year if and only if it is divisible by 400. For example, 2000 is a leap year but 2100 is not.
2. If the year is not divisible by 100, then it is a leap year if and only if it is divisible by 4. For example, 2008 is a leap year but 2009 is not.

Here is a sample run of the program:

```
Enter a month (1-12): 2
Enter a year: 2008
29 days
```

**11. Math Tutor**

*This is a modification of Programming Challenge 17 from Chapter 3.* Write a program that can be used as a math tutor for a young student. The program should display two random numbers that are to be added, such as:

$$\begin{array}{r} 247 \\ + 129 \\ \hline \end{array}$$

The program should 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.

**12. 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 sold and computes the total cost of the purchase.

*Input Validation: Make sure the number of units is greater than 0.*

**13. Book Club Points**

Serendipity Booksellers has a book club that awards points to its customers based on the number of books purchased each month. The points are awarded as follows:

- If a customer purchases 0 books, he or she earns 0 points.
- If a customer purchases 1 book, he or she earns 5 points.
- If a customer purchases 2 books, he or she earns 15 points.
- If a customer purchases 3 books, he or she earns 30 points.
- If a customer purchases 4 or more books, he or she earns 60 points.

Write a program that asks the user to enter the number of books he or she has purchased this month then displays the number of points awarded.

**14. 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

The bank also charges an extra \$15 if the balance of the account falls below \$400 (before any check fees are applied). Write a program that asks for the beginning balance and the number of checks written. Compute and display the bank's service fees for the month.

*Input Validation: Do not accept a negative value for the number of checks written. If a negative value is given for the beginning balance, display an urgent message indicating the account is overdrawn.*

#### 15. Shipping Charges

The Fast Freight Shipping Company charges the following rates:

| Weight of Package (in Kilograms)   | Rate per 500 Miles Shipped |
|------------------------------------|----------------------------|
| 2 kg or less                       | \$1.10                     |
| Over 2 kg but not more than 6 kg   | \$2.20                     |
| Over 6 kg but not more than 10 kg  | \$3.70                     |
| Over 10 kg but not more than 20 kg | \$4.80                     |

Write a program that asks for the weight of the package and the distance it is to be shipped, then displays the charges.

*Input Validation: Do not accept values of 0 or less for the weight of the package. Do not accept weights of more than 20 kg (this is the maximum weight the company will ship). Do not accept distances of less than 10 miles or more than 3,000 miles. These are the company's minimum and maximum shipping distances.*

#### 16. 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.

*Input Validation: Only accept positive numbers for the times.*

#### 17. 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 order of height (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.*

#### 18. 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 that the food is low in fat.

One gram of fat has 9 calories, so

Calories from fat = fat grams \* 9

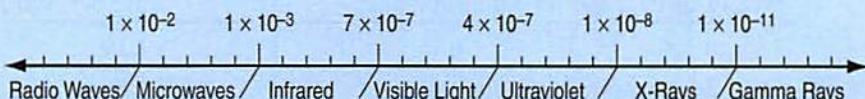
The percentage of calories from fat can be calculated as

Calories from fat ÷ total calories

*Input Validation: Make sure the number of calories and fat grams are not less than 0. 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.*

**19. Spectral Analysis**

If a scientist knows the wavelength of an electromagnetic wave, he or she can determine what type of radiation it is. Write a program that asks for the wavelength of an electromagnetic wave in meters and then displays what that wave is according to the chart below. (For example, a wave with a wavelength of  $1E-10$  meters would be an X-ray.)

**20. The Speed of Sound**

The following table shows the approximate speed of sound in air, water, and steel.

| Medium | Speed                  |
|--------|------------------------|
| Air    | 1,100 feet per second  |
| Water  | 4,900 feet per second  |
| Steel  | 16,400 feet per second |

Write a program that displays a menu allowing the user to select air, water, or steel. After the user has made a selection, he or she should be asked to enter the distance a sound wave will travel in the selected medium. 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 choices from the menu. Do not accept distances less than 0.*

**21. The Speed of Sound in Gases**

When sound travels through a gas, its speed 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 degrees centigrade, measured in meters per second, when traveling through carbon dioxide, air, helium, and hydrogen.

| Medium         | Speed (Meters per Second) |
|----------------|---------------------------|
| Carbon dioxide | 258.0                     |
| Air            | 331.5                     |
| Helium         | 972.0                     |
| Hydrogen       | 1,270.0                   |

Write a program that displays a menu allowing the user to select one of these four 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.*

## 22. 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 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                |

## 23. 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 then display its area. Use the following formula:

$$\text{area} = \pi r^2$$

Use 3.14159 for  $\pi$  and the radius of the circle for  $r$ . If the user enters 2, the program should ask for the length and width of the rectangle, then display the rectangle's area. Use the following formula:

$$\text{area} = \text{length} * \text{width}$$

If the user enters 3, the program should ask for the length of the triangle's base and its height, then display its area. Use the following formula:

$$\text{area} = \text{base} * \text{height} * .5$$

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.*

## 24. Long-Distance Calls

A long-distance carrier charges the following rates for telephone calls:

| Starting Time of Call | Rate per Minute |
|-----------------------|-----------------|
| 00:00–06:59           | 0.05            |
| 07:00–19:00           | 0.45            |
| 19:01–23:59           | 0.20            |

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)
```

### 25. Mobile Service Provider

A mobile phone service provider has three different data plans for its customers:

Package A: For \$39.99 per month, 4 gigabytes are provided. Additional data costs \$10 per gigabyte.

Package B: For \$59.99 per month, 8 gigabytes are provided. Additional data costs \$5 per gigabyte.

Package C: For \$69.99 per month, unlimited data is provided.

Write a program that calculates a customer's monthly bill. It should ask which package the customer has purchased and how many gigabytes were used. It should then display the total amount due.

*Input Validation: Be sure the user only selects package A, B, or C.*

### 26. Mobile Service Provider, Part 2

Modify the Program in Programming Challenge 25 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.

### 27. Wi-Fi Diagnostic Tree

Figure 4-11 shows a simplified flowchart for troubleshooting a bad Wi-Fi connection. Use the flowchart to create a program that leads a person through the steps of fixing a bad Wi-Fi connection. Here is an example of the program's output:

Reboot the computer and try to connect.

Did that fix the problem? **no**

Reboot the router and try to connect.

Did that fix the problem? **yes**

Notice the program ends as soon as a solution is found to the problem. Here is another example of the program's output:

Reboot the computer and try to connect.

Did that fix the problem? **no**

Reboot the router and try to connect.

Did that fix the problem? **no**

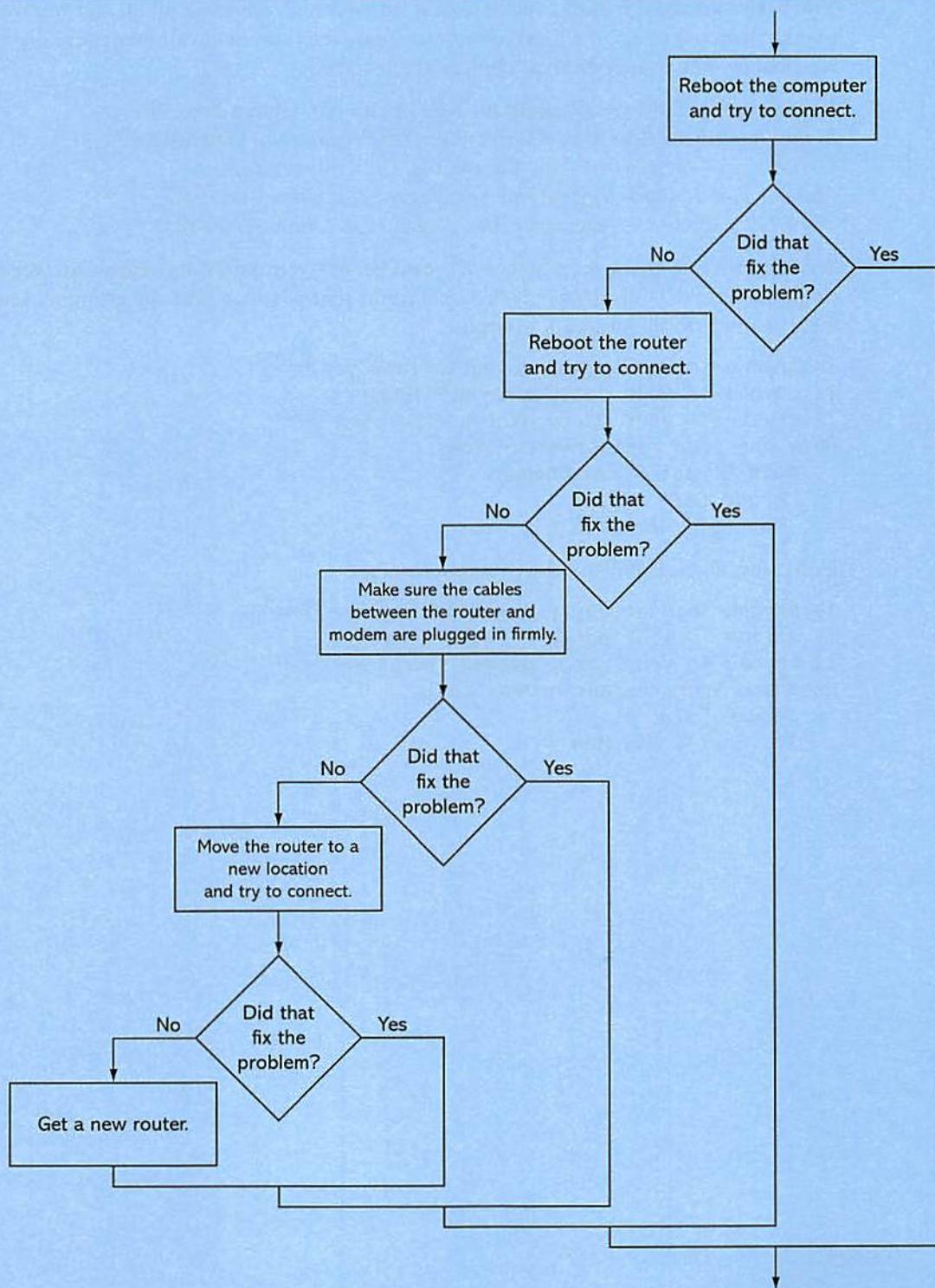
Make sure the cables between the router and modem are plugged in firmly.

Did that fix the problem? **no**

Move the router to a new location.

Did that fix the problem? **no**

Get a new router.

**Figure 4-11** Troubleshooting a bad Wi-Fi connection

### 28. Restaurant Selector

You have a group of friends coming to visit for your high school reunion, and you want to take them out to eat at a local restaurant. You aren't sure if any of them have dietary restrictions, but your restaurant choices are as follows:

*Joe's Gourmet Burgers*—Vegetarian: No, Vegan: No, Gluten-Free: No

*Main Street Pizza Company*—Vegetarian: Yes, Vegan: No, Gluten-Free: Yes

*Corner Café*—Vegetarian: Yes, Vegan: Yes, Gluten-Free: Yes

*Mama's Fine Italian*—Vegetarian: Yes, Vegan: No, Gluten-Free: No

*The Chef's Kitchen*—Vegetarian: Yes, Vegan: Yes, Gluten-Free: Yes

Write a program that asks whether any members of your party are vegetarian, vegan, or gluten-free, then displays only the restaurants that you may take the group to. Here is an example of the program's output:

```
Is anyone in your party a vegetarian? yes Enter
```

```
Is anyone in your party a vegan? no Enter
```

```
Is anyone in your party gluten-free? yes Enter
```

Here are your restaurant choices:

Main Street Pizza Company

Corner Cafe

The Chef's Kitchen

Here is another example of the program's output:

```
Is anyone in your party a vegetarian? yes Enter
```

```
Is anyone in your party a vegan? yes Enter
```

```
Is anyone in your party gluten-free? yes Enter
```

Here are your restaurant choices:

Corner Cafe

The Chef's Kitchen

**TOPICS**

- |                                               |                                                               |
|-----------------------------------------------|---------------------------------------------------------------|
| 5.1 The Increment and Decrement Operators     | 5.7 Keeping a Running Total                                   |
| 5.2 Introduction to Loops: The while Loop     | 5.8 Sentinels                                                 |
| 5.3 Using the while Loop for Input Validation | 5.9 Focus on Software Engineering: Deciding Which Loop to Use |
| 5.4 Counters                                  | 5.10 Nested Loops                                             |
| 5.5 The do-while Loop                         | 5.11 Using Files for Data Storage                             |
| 5.6 The for Loop                              | 5.12 Optional Topics: Breaking and Continuing a Loop          |

**5.1**

## The Increment and Decrement Operators

**CONCEPT:** `++` and `--` are operators that add and subtract 1 from their operands.

To *increment* a value means to increase it by one, and to *decrement* a value means to decrease it by one. Both of the following statements increment the variable `num`:

```
num = num + 1;
num += 1;
```

And `num` is decremented in both of the following statements:

```
num = num - 1;
num -= 1;
```

C++ provides a set of simple unary operators designed just for incrementing and decrementing variables. The increment operator is `++`, and the decrement operator is `--`. The following statement uses the `++` operator to increment `num`:

```
num++;
```

And the following statement decrements `num`:

```
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 postfix and prefix modes, these operators add 1 to or subtract 1 from their operand. Program 5-1 shows 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
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 incrementing or decrementing. For example, look at the following statements:

```
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 be incremented to 5, then cout will display 5:

```
num = 4;
cout << ++num;
```

Program 5-2 illustrates these dynamics further.

### Program 5-2

```
1 // This program demonstrates the prefix and postfix
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 cout << num << endl; // Displays 4
11 cout << num++ << endl; // Displays 4, then adds 1 to num
12 cout << num << endl; // Displays 5
13 cout << ++num << endl; // Adds 1 to num, then displays 6
14 cout << endl; // Displays a blank line
15
```

(program continues)

**Program 5-2**

(continued)

```

16 cout << num << endl; // Displays 6
17 cout << num-- << endl; // Displays 6, then subtracts 1 from num
18 cout << num << endl; // Displays 5
19 cout << --num << endl; // Subtracts 1 from num, then displays 4
20
21 return 0;
22 }
```

**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 10 displays 4. Then, line 11 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 12 executes, `num` will hold the value 5, so 5 is displayed. Then, line 13 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 16 through 19, 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, it acts *after* the assignment takes place. So, this code will store 1 in `y`. After the code has executed, `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, so it acts on the variable `x` before the assignment takes place. So, this code will store 2 in `y`. After the code has executed, `x` will also 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

Sometimes you will see code where the `++` and `--` operators are used in relational expressions. Just as in mathematical expressions, the difference between postfix and prefix modes 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";
```



## 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

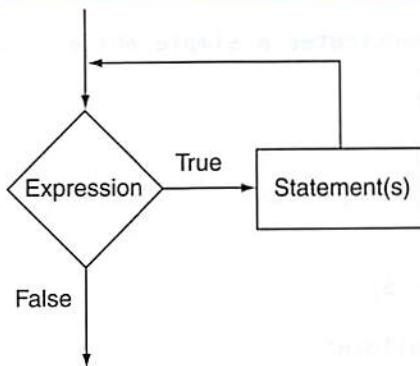
**CONCEPT:** A loop is part of a program that repeats.



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 these structures 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 logic of a `while` loop.

**Figure 5-1** Logic of a `while` loop

Here is the general format of the `while` loop:

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

In the general format, *expression* is any expression that can be evaluated as true or false, and *statement* is any valid C++ statement. The first line shown in the format is sometimes called the *loop header*. It consists of the key word `while` followed by an *expression* enclosed in parentheses.

Here's how the loop works: the *expression* is tested, and if it is true, the *statement* is executed. Then, the *expression* is tested again. If it is true, the *statement* is executed. This cycle repeats until the *expression* is false.

The statement that is repeated is known as the *body* of the loop. It is also considered a conditionally executed statement, because it is executed only under the condition that the *expression* is true.

Notice there is no semicolon after the expression in parentheses. Like the `if` statement, the `while` loop is not complete without the statement that follows it.

If you wish the `while` loop to repeat a block of statements, its format is:

```
while (expression)
{
 statement;
 statement;
 // Place as many statements here
 // as necessary.
}
```

The `while` loop works like an `if` statement that executes over and over. As long as the expression inside the parentheses is true, the conditionally executed statement or block will repeat. Program 5-3 uses the `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 = 0;
8
9 while (number < 5)
10 {
11 cout << "Hello\n";
12 number++;
13 }
14 cout << "That'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 0. In line 9, the `while` loop begins with this statement:

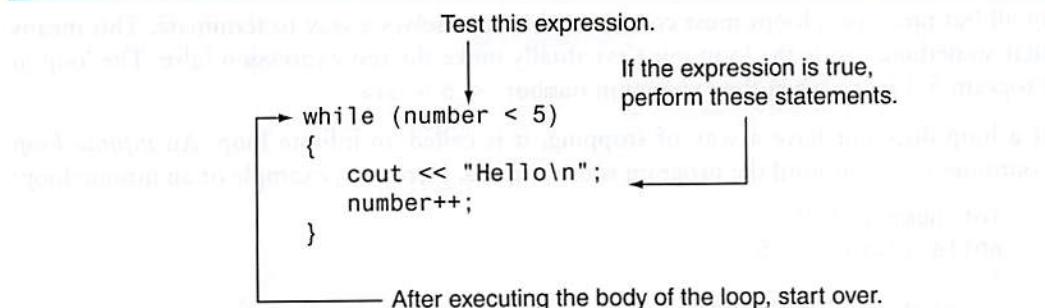
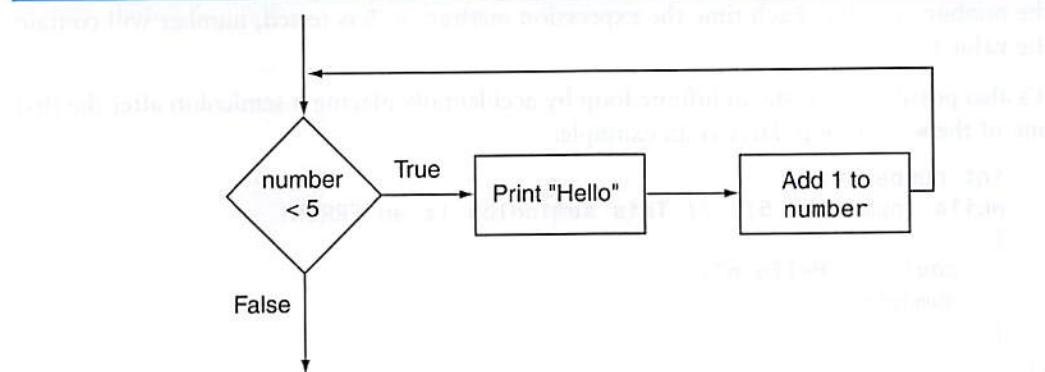
```
while (number < 5)
```

This statement tests the variable `number` to determine whether it is less than 5. If it is, then the statements in the body of the loop (lines 11 and 12) are executed:

```
cout << "Hello\n";
number++;
```

The statement in line 11 prints the word “Hello.” The statement in line 12 uses the increment operator to add one to `number`. 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 if it is true, the statements in the body of the loop are executed again. This cycle repeats until the expression `number < 5` is false. This is illustrated in Figure 5-2.

Each repetition of a loop is known as an *iteration*. This loop will perform five iterations because the variable `number` is initialized with the value 0, and it is incremented each time the body of the loop is executed. When the expression `number < 5` is tested and found to be false, the loop will terminate and the program will resume execution at the statement that immediately follows the loop. Figure 5-3 shows the logic of this loop.

**Figure 5-2** Actions of the `while` loop**Figure 5-3** Flowchart for the `while` loop

In this example, the `number` variable is referred to as the *loop control variable* because it controls the number of times that the loop iterates.

### The `while` Loop 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 in line 7 of Program 5-3:

```
int number = 0;
```

The `number` variable is initialized with the value 0. If `number` had been initialized with the value 5 or greater, as shown in the following program segment, the loop would never execute:

```
int number = 6;
while (number < 5)
{
 cout << "Hello\n";
 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 that 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 expression `number < 5` is false.

If a loop does not have a way of stopping, it is called an infinite loop. An *infinite loop* continues to repeat until the program is interrupted. Here is an example of an infinite loop:

```
int number = 0;
while (number < 5)
{
 cout << "Hello\n";
}
```

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 0.

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 = 0;
while (number < 5); // This semicolon is an ERROR!
{
 cout << "Hello\n";
 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:

```
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 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 = 0;
// This loop is missing its braces!
while (number < 5)
 cout << "Hello\n";
 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 instead of testing whether 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\n"; number++; }
```

Avoid this style of programming. The programming 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 additional level.
- If the loop repeats a block, each line inside the braces should be indented.

This programming style should visually set the body of the loop apart from the surrounding code. In general, you'll find a similar style being used with the other types of loops presented in this chapter.

### In the Spotlight:

#### Designing a Program with a while Loop

A project currently underway at Chemical Labs, Inc. requires that a substance be continually heated in a vat. A technician must check the substance's temperature every 15 minutes. If the substance's temperature does not exceed 102.5 degrees Celsius, then the technician does nothing. However, if the temperature is greater than 102.5 degrees Celsius, the technician must turn down the vat's thermostat, wait 5 minutes, and check the temperature again. The technician repeats these steps until the temperature does not exceed 102.5 degrees Celsius. The director of engineering has asked you to write a program that guides the technician through this process.

Here is the algorithm:

1. *Prompt the user to enter the substance's temperature.*
2. *Repeat the following steps as long as the temperature is greater than 102.5 degrees Celsius:*
  - a. *Tell the technician to turn down the thermostat, wait 5 minutes, and check the temperature again.*
  - b. *Prompt the user to enter the substance's temperature.*
3. *After the loop finishes, tell the technician that the temperature is acceptable and to check it again in 15 minutes.*

After reviewing this algorithm, you realize that steps 2a and 2b should not be performed if the test condition (temperature is greater than 102.5) is false to begin with. The `while` loop will work well in this situation, because it will not execute even once if its condition is false. Program 5-4 shows the code for the program.

### Program 5-4

```
1 // This program assists a technician in the process
2 // of checking a substance's temperature.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8 const double MAX_TEMP = 102.5; // Maximum temperature
9 double temperature; // To hold the temperature
10
11 // Get the current temperature.
12 cout << "Enter the substance's Celsius temperature: ";
13 cin >> temperature;
14
15 // As long as necessary, instruct the technician
16 // to adjust the thermostat.
17 while (temperature > MAX_TEMP)
18 {
19 cout << "The temperature is too high. Turn the\n";
20 cout << "thermostat down and wait 5 minutes.\n";
21 cout << "Then take the Celsius temperature again\n";
22 cout << "and enter it here: ";
23 cin >> temperature;
24 }
25
26 // Remind the technician to check the temperature
27 // again in 15 minutes.
28 cout << "The temperature is acceptable.\n";
29 cout << "Check it again in 15 minutes.\n";
30
31 return 0;
32 }
```

### Program Output with Example Input Shown in Bold

```
Enter the substance's Celsius temperature: 104.7
The temperature is too high. Turn the
thermostat down and wait 5 minutes.
Then take the Celsius temperature again
and enter it here: 103.2
The temperature is too high. Turn the
thermostat down and wait 5 minutes.
Then take the Celsius temperature again
and enter it here: 102.1
The temperature is acceptable.
Check it again in 15 minutes.
```

**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.

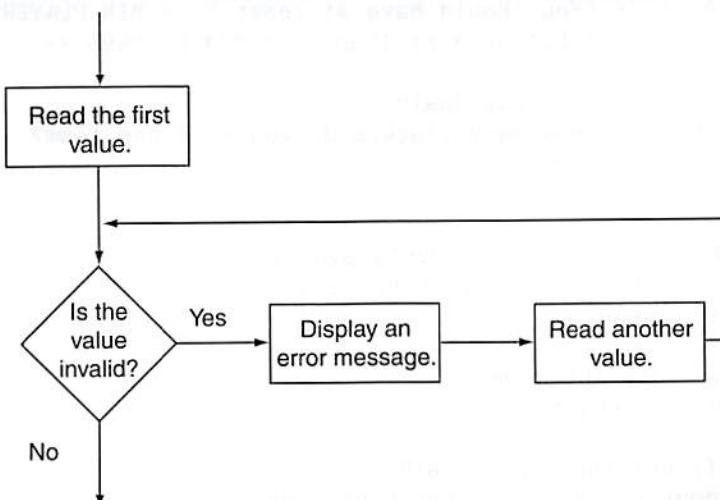
Perhaps the most famous saying of the computer industry is “garbage in, garbage out.” The integrity of a program’s output is only as good as its input, so you should try to make sure garbage does not go into your programs. *Input validation* is the process of inspecting data given to a program by the 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 reenter 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 loop 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 5-4.

**Figure 5-4** Flowchart for input validation



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 5-5 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 25 through 34 and lines 41 through 46) to validate the user's input.

### Program 5-5

```
1 // This program calculates the number of soccer teams
2 // that a youth league may create from the number of
3 // available players. Input validation is demonstrated
4 // with while loops.
5 #include <iostream>
6 using namespace std;
7
8 int main()
9 {
10 // Constants for minimum and maximum players
11 const int MIN_PLAYERS = 9,
12 MAX_PLAYERS = 15;
13
14 // Variables
15 int players, // Number of available players
16 teamPlayers, // Number of desired players per team
17 numTeams, // Number of teams
18 leftOver; // Number of players left over
19
20 // Get the number of players per team.
21 cout << "How many players do you wish per team? ";
22 cin >> teamPlayers;
23
24 // Validate the input.
25 while (teamPlayers < MIN_PLAYERS || teamPlayers > MAX_PLAYERS)
26 {
27 // Explain the error.
28 cout << "You should have at least " << MIN_PLAYERS
29 << " but no more than " << MAX_PLAYERS << " per team.\n";
30
31 // Get the input again.
32 cout << "How many players do you wish per team? ";
33 cin >> teamPlayers;
34 }
35
36 // Get the number of players available.
37 cout << "How many players are available? ";
38 cin >> players;
39
40 // Validate the input.
41 while (players <= 0)
42 {
43 // Get the input again.
44 cout << "Please enter 0 or greater: ";
45 cin >> players;
46 }
47 }
```

```
48 // Calculate the number of teams.
49 numTeams = players / teamPlayers;
50
51 // Calculate the number of leftover players.
52 leftOver = players % teamPlayers;
53
54 // Display the results.
55 cout << "There will be " << numTeams << " teams with "
56 << leftOver << " players left over.\n";
57 return 0;
58 }
```

### Program Output with Example Input Shown in Bold

How many players do you wish per team? **4**   
You should have at least 9 but no more than 15 per team.  
How many players do you wish per team? **12**   
How many players are available? **-142**   
Please enter 0 or greater: **142**   
There will be 11 teams with 10 players left over.



### Checkpoint

- 5.2 Write an input validation loop that asks the user to enter a number in the range of 10 through 25.
- 5.3 Write an input validation loop that asks the user to enter ‘Y’, ‘y’, ‘N’, or ‘n’.
- 5.4 Write an input validation loop that asks the user to enter “Yes” or “No”.

## 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 control or keep track of the number of iterations a loop performs. For example, Program 5-6 displays a table consisting of the numbers 1 through 10 and their squares, so its loop must iterate 10 times.

#### Program 5-6

```
1 // This program displays a list of numbers and
2 // their squares.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8 const int MIN_NUMBER = 1, // Starting number to square
9 MAX_NUMBER = 10; // Maximum number to square
10}
```

(program continues)

**Program 5-6**

(continued)

```

11 int num = MIN_NUMBER; // Counter
12
13 cout << "Number Number Squared\n";
14 cout << "-----\n";
15 while (num <= MAX_NUMBER)
16 {
17 cout << num << "\t\t" << (num * num) << endl;
18 num++; //Increment the counter.
19 }
20 return 0;
21 }
```

**Program Output**

Number Number Squared

| Number | Squared |
|--------|---------|
| 1      | 1       |
| 2      | 4       |
| 3      | 9       |
| 4      | 16      |
| 5      | 25      |
| 6      | 36      |
| 7      | 49      |
| 8      | 64      |
| 9      | 81      |
| 10     | 100     |

In Program 5-6, the variable `num`, which starts at 1, is incremented each time through the loop. When `num` reaches 11, the loop stops. `num` is used as a *counter* variable, which means it is regularly incremented in each iteration of the loop. In essence, `num` keeps count of the number of iterations the loop has performed.



**NOTE:** It's important that `num` be properly initialized. Remember, variables defined inside a function have no guaranteed starting value.

**5.5****The do-while Loop**

**CONCEPT:** The `do-while` loop is a posttest loop, which means its expression is tested after each iteration.

The `do-while` loop looks something like an inverted `while` loop. Here is the `do-while` loop's format when the body of the loop contains only a single statement:

```

do
 statement;
while (expression);
```

Here is the format of the `do-while` loop when the body of the loop contains multiple statements:

```
do
{
 statement;
 statement;
 // Place as many statements here
 // as necessary.
} while (expression);
```

**NOTE:** The `do-while` loop must be terminated with a semicolon.

The `do-while` loop is a *posttest* loop. This means it does not test its expression until it has completed an iteration. As a result, the `do-while` loop always performs at least one iteration, even if the expression is false to begin with. This differs from the behavior of a `while` loop, which you will recall is a *pretest* loop. For example, in the following `while` loop the `cout` statement will not execute at all:

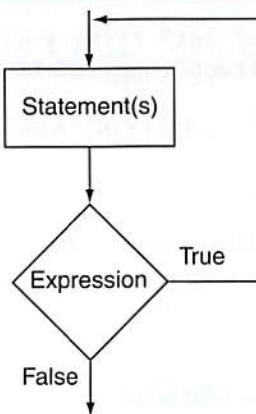
```
int x = 1;
while (x < 0)
 cout << x << endl;
```

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);
```

Figure 5-5 illustrates the logic of the `do-while` loop.

**Figure 5-5** Flowchart for a `do-while` loop



You should use the `do-while` loop when you want to make sure the loop executes at least once. For example, Program 5-7 averages a series of three test scores for a student.

After the average is displayed, it asks the user if he or she wants to average another set of test scores. The program repeats as long as the user enters Y for yes.

### Program 5-7

```

1 // This program averages 3 test scores. It repeats as
2 // many times as the user wishes.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8 int score1, score2, score3; // Three scores
9 double average; // Average score
10 char again; // To hold Y or N input
11
12 do
13 {
14 // Get three scores.
15 cout << "Enter 3 scores and I will average them: ";
16 cin >> score1 >> score2 >> score3;
17
18 // Calculate and display the average.
19 average = (score1 + score2 + score3) / 3.0;
20 cout << "The average is " << average << ".\n";
21
22 // Does the user want to average another set?
23 cout << "Do you want to average another set? (Y/N) ";
24 cin >> again;
25 } while (again == 'Y' || again == 'y');
26 return 0;
27 }
```

### 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
```

When this program was written, the programmer had no way of knowing the number of times the loop would iterate. This is because the loop asks the user if he or she wants to repeat the process. This type of loop is known as a *user-controlled loop*, because it allows the user to decide the number of iterations.

### Using do-while with Menus

The do-while loop is a good choice for repeating a menu. Recall Program 4-27, which displayed a menu of health club packages. Program 5-8 is a modification of that program, which uses a do-while loop to repeat the program until the user selects item 4 from the menu.

**Program 5-8**

```
1 // This program displays a menu and asks the user to make a
2 // selection. A do-while loop repeats the program until the
3 // user selects item 4 from the menu.
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;
7
8 int main()
9 {
10 // Constants for menu choices
11 const int ADULT_CHOICE = 1,
12 CHILD_CHOICE = 2,
13 SENIOR_CHOICE = 3,
14 QUIT_CHOICE = 4;
15
16 // Constants for membership rates
17 const double ADULT = 40.0,
18 CHILD = 20.0,
19 SENIOR = 30.0;
20
21 // Variables
22 int choice; // Menu choice
23 int months; // Number of months
24 double charges; // Monthly charges
25
26 // Set up numeric output formatting.
27 cout << fixed << showpoint << setprecision(2);
28
29 do
30 {
31 // Display the menu.
32 cout << "\n\t\tHealth Club Membership Menu\n\n"
33 << "1. Standard Adult Membership\n"
34 << "2. Child Membership\n"
35 << "3. Senior Citizen Membership\n"
36 << "4. Quit the Program\n\n"
37 << "Enter your choice: ";
38 cin >> choice;
39
40 // Validate the menu selection.
41 while (choice < ADULT_CHOICE || choice > QUIT_CHOICE)
42 {
43 cout << "Please enter a valid menu choice: ";
44 cin >> choice;
45 }
46
47 // Process the user's choice.
48 if (choice != QUIT_CHOICE)
49 {
50 // Get the number of months.
51 cout << "For how many months? ";
```

*(program continues)*

**Program 5-8***(continued)*

```

52 cin >> months;
53
54 // Respond to the user's menu selection.
55 switch (choice)
56 {
57 case ADULT_CHOICE:
58 charges = months * ADULT;
59 break;
60 case CHILD_CHOICE:
61 charges = months * CHILD;
62 break;
63 case SENIOR_CHOICE:
64 charges = months * SENIOR;
65 }
66
67 // Display the monthly charges.
68 cout << "The total charges are $"
69 << charges << endl;
70 }
71 } while (choice != QUIT_CHOICE);
72 return 0;
73 }
```

**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 the following program segments display?

```

A) int count = 10;
 do
 {
 cout << "Hello World\n";
 count++;
 } while (count < 1);
```

```

B) int v = 10;
 do
 cout << v << endl;
 while (v < 5);

C) int count = 0, number = 0, limit = 4;
 do
 {
 number += 2;
 count++;
 } while (count < limit);
 cout << number << " " << count << endl;

```

**5.6**

## The for Loop

**CONCEPT:** The **for** loop is ideal for performing a known number of iterations.

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 12 times. In essence, the loop counts to 12 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 maximum value. When the counter variable reaches its maximum 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 when it is used to repeat a single statement:

```

for (initialization; test; update)
 statement;

```

The format of the **for** loop when it is used to repeat a block is

```

for (initialization; test; update)
{
 statement;
 statement;
 // Place as many statements here
 // as necessary.
}

```

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 not a semicolon after the third expression.) The first expression is the *initialization expression*. It is normally used to initialize a counter variable 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*. This is an expression that controls the execution of the loop. As long as this expression 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 = 0; count < 5; count++)
 cout << "Hello" << endl;
```

In this loop, the initialization expression is `count = 0`, 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 5-6 illustrates the sequence of events that takes place during the loop's execution. Notice Steps 2 through 4 are repeated as long as the test expression is true.

**Figure 5-6** Sequence of events in a `for` loop

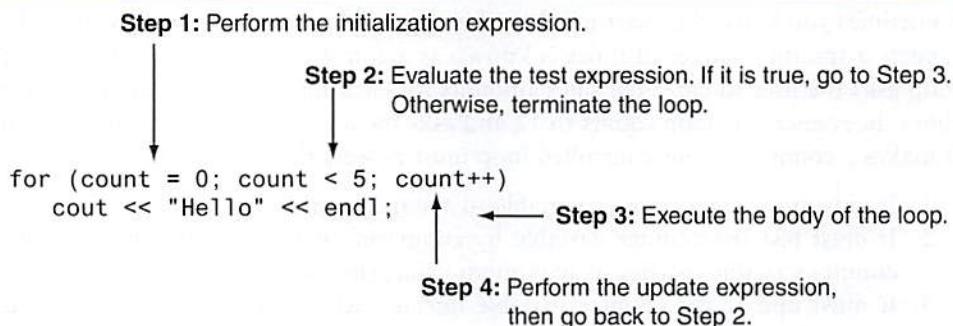
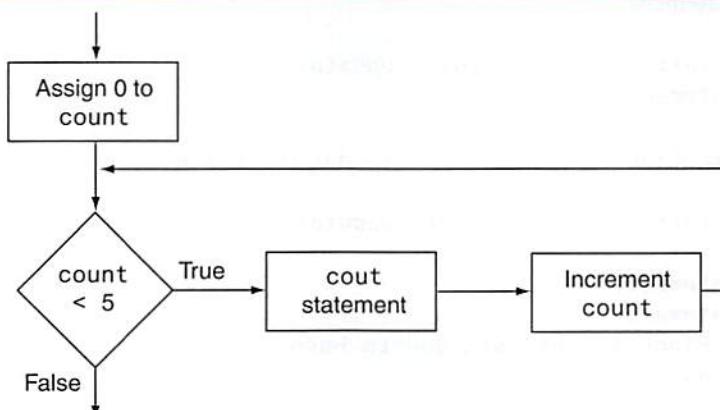


Figure 5-7 shows the loop's logic in the form of a flowchart.

**Figure 5-7** `for` loop logic



Notice how the counter variable, `count`, is used to control the number of times that 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` is false, the loop terminates. Also notice 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. It is also possible to use the counter variable within the body of the loop. For example, look at the following code:

```
for (number = 1; number <= 10; number++)
 cout << number << " " << endl;
```

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 6 7 8 9 10
```

As you can see, the loop displays the contents of the `number` variable during each iteration. Program 5-9 shows another example of a `for` loop that uses its counter variable within the body of the loop. This is yet another program that displays a table showing the numbers 1 through 10 and their squares.

### Program 5-9

```
1 // This program displays the numbers 1 through 10 and
2 // their squares.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8 const int MIN_NUMBER = 1, // Starting value
9 MAX_NUMBER = 10; // Ending value
10 int num;
11
12 cout << "Number Number Squared\n";
13 cout << "-----\n";
14
15 for (num = MIN_NUMBER; num <= MAX_NUMBER; num++)
16 cout << num << "\t\t" << (num * num) << endl;
17
18 return 0;
19 }
```

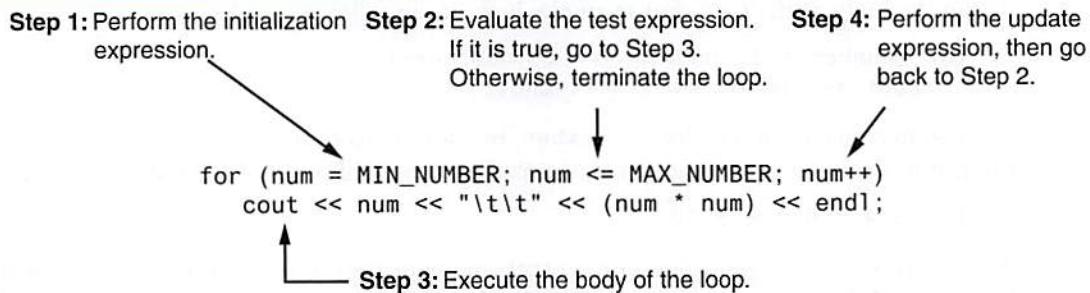
### Program Output

Number Number Squared

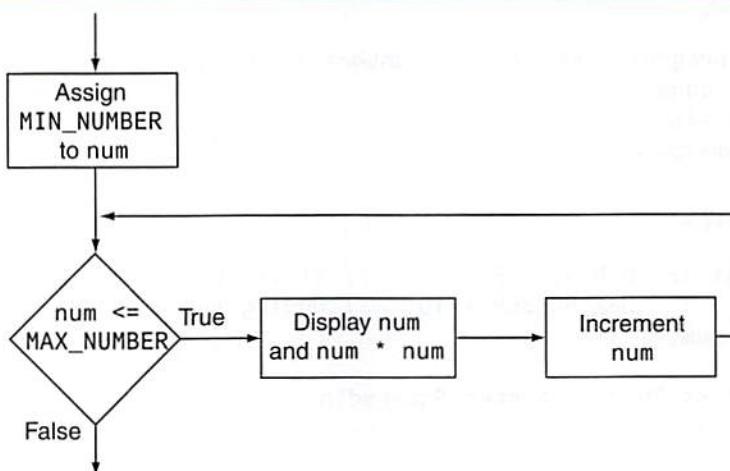
|    | ----- |
|----|-------|
| 1  | 1     |
| 2  | 4     |
| 3  | 9     |
| 4  | 16    |
| 5  | 25    |
| 6  | 36    |
| 7  | 49    |
| 8  | 64    |
| 9  | 81    |
| 10 | 100   |

Figure 5-8 illustrates the sequence of events performed by this `for` loop, and Figure 5-9 shows the logic of the loop as a flowchart.

**Figure 5-8 Sequence of events in the for loop**



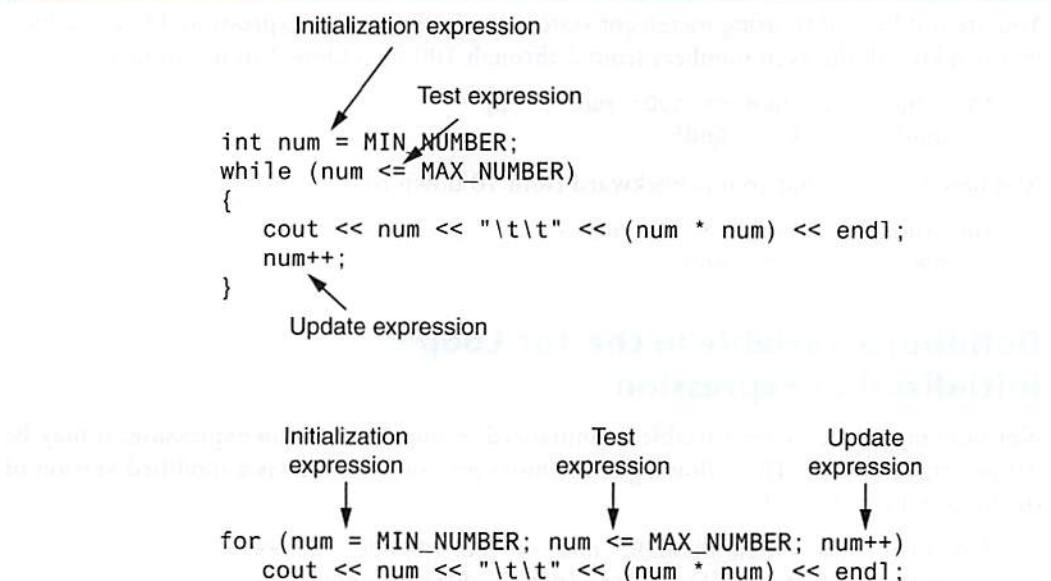
**Figure 5-9 Flowchart for the for loop**



### Using the for Loop instead of while or do-while

You should use the `for` loop instead of the `while` or `do-while` loop in any situation that clearly requires an initialization, uses a false condition to stop the loop, and requires an update to occur at the end of each loop iteration. Program 5-9 is a perfect example. It requires that the `num` variable be initialized to 1, it stops the loop when `num` is greater than 10, and it increments `num` at the end of each loop iteration.

Recall that when we first introduced the idea of a counter variable we examined Program 5-6, which uses a `while` loop to display the table of numbers and their squares. Because the loop in that program requires an initialization, uses a false test expression to stop, and performs an increment at the end of each iteration, it can easily be converted to a `for` loop. Figure 5-10 shows how the `while` loop in Program 5-6 and the `for` loop in Program 5-9 each have initialization, test, and update expressions.

**Figure 5-10** Initialization, test, and update expressions

## The for Loop Is a Pretest Loop

Because the `for` loop tests its test expression before it performs an iteration, it is a pretest loop. 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 using increment statements 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 = MIN_NUMBER; num <= MAX_NUMBER; num++)
 cout << num << "\t\t" << (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 makes sense to define it in the loop header. This makes the variable's purpose more clear.

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

Sometimes you want the user to determine the maximum value of the counter variable in a `for` loop, and therefore determine the number of times the loop iterates. For example, look at Program 5-10. This is another program that displays a list of numbers and their squares. Instead of displaying the numbers 1 through 10, this program allows the user to enter the minimum and maximum values to display.

### Program 5-10

```
1 // This program demonstrates a user-controlled for loop.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7 int minNumber, // Starting number to square
8 maxNumber; // Maximum number to square
```

```

9
10 // Get the minimum and maximum values to display.
11 cout << "I will display a table of numbers and "
12 << "their squares.\n"
13 << "Enter the starting number: ";
14 cin >> minNumber;
15 cout << "Enter the ending number: ";
16 cin >> maxNumber;
17
18 // Display the table.
19 cout << "Number Number Squared\n"
20 << "-----\n";
21
22 for (int num = minNumber; num <= maxNumber; num++)
23 cout << num << "\t\t" << (num * num) << endl;
24
25 return 0;
26 }
```

### Program Output with Example Input Shown in Bold

I will display a table of numbers and their squares.

Enter the starting number: **6**

Enter the ending number: **12**

Number Number Squared

|    | ----- |
|----|-------|
| 6  | 36    |
| 7  | 49    |
| 8  | 64    |
| 9  | 81    |
| 10 | 100   |
| 11 | 121   |
| 12 | 144   |

Before the loop, the code in lines 11 through 16 asks the user to enter the starting and ending numbers. These values are stored in the `minNumber` and `maxNumber` variables. These values are used in the for loop's initialization and test expressions.

```
for (int num = minNumber; num <= maxNumber; num++)
```

In this loop, the `num` variable takes on the values from `maxNumber` through `maxValue`, and then the loop terminates.

## 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:

```

int x, y;
for (x = 1, y = 1; x <= 5; x++)
{
 cout << x << " plus " << y
 << " equals " << (x + y)
 << endl;
}

```

This loop's initialization expression is

```
x = 1, y = 1
```

This initializes two 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:

```

int x, y;
for (x = 1, y = 1; x <= 5; x++, y++)
{
 cout << x << " plus " << y
 << " equals " << (x + y)
 << endl;
}

```

The loop's update expression is

```
x++, y++
```

This update expression increments both the *x* and *y* variables. 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

The initialization expression may be omitted from inside the **for** loop's parentheses if it has already been performed or no initialization is needed. Here is an example of the loop in Program 5-10 with the initialization being performed prior to the loop:

```

int num = 1;
for (; num <= maxValue; num++)
 cout << num << "\t\t" << (num * num) << endl;

```

You may also omit the update expression 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 << "\t\t" << (num * num) << endl;
 num++;
}
```

You can even go so far as to omit all three expressions from the `for` loop's parentheses. Be warned, however, that if you leave out the test expression, the loop has no built-in way of terminating. Here is an example:

```
for (; ;)
 cout << "Hello World\n";
```

Because this loop has no way of stopping, it will display "Hello World\n" forever (or until something interrupts the program).



### In the Spotlight:

#### Designing a Count-Controlled Loop with the for Statement

Your friend Amanda just inherited a European sports car from her uncle. Amanda lives in the United States, and she is afraid she will get a speeding ticket because the car's speedometer indicates kilometers per hour. She has asked you to write a program that displays a table of speeds in kilometers per hour with their values converted to miles per hour. The formula for converting kilometers per hour to miles per hour is:

$$\text{MPH} = \text{KPH} * 0.6214$$

In the formula, `MPH` is the speed in miles per hour and `KPH` is the speed in kilometers per hour.

The table your program displays should show speeds from 60 kilometers per hour through 130 kilometers per hour, in increments of 10, along with their values converted to miles per hour. The table should look something like this:

| KPH | MPH  |
|-----|------|
| 60  | 37.3 |
| 70  | 43.5 |
| 80  | 49.7 |
| .   | .    |
| 130 | 80.8 |

After thinking about this table of values, you decide that you will write a `for` loop that uses a counter variable to hold the kilometer-per-hour speeds. The counter's starting value will be 60, its ending value will be 130, and you will add 10 to the counter variable after each iteration. Inside the loop, you will use the counter variable to calculate a speed in miles per hour. Program 5-11 shows the code.

**Program 5-11**

```

1 // This program converts the speeds 60 kph through
2 // 130 kph (in 10 kph increments) to mph.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9 // Constants for the speeds
10 const int START_KPH = 60, // Starting speed
11 END_KPH = 130, // Ending speed
12 INCREMENT = 10; // Speed increment
13
14 // Constant for the conversion factor
15 const double CONVERSION_FACTOR = 0.6214;
16
17 // Variables
18 int kph; // To hold speeds in kph
19 double mph; // To hold speeds in mph
20
21 // Set the numeric output formatting.
22 cout << fixed << showpoint << setprecision(1);
23
24 // Display the table headings.
25 cout << "KPH\tMPH\n";
26 cout << "-----\n";
27
28 // Display the speeds.
29 for (kph = START_KPH; kph <= END_KPH; kph += INCREMENT)
30 {
31 // Calculate mph
32 mph = kph * CONVERSION_FACTOR;
33
34 // Display the speeds in kph and mph.
35 cout << kph << "\t" << mph << endl;
36 }
37
38 return 0;
39 }
```

**Program Output**

| KPH | MPH  |
|-----|------|
| 60  | 37.3 |
| 70  | 43.5 |
| 80  | 49.7 |
| 90  | 55.9 |
| 100 | 62.1 |
| 110 | 68.4 |
| 120 | 74.6 |
| 130 | 80.8 |



## Checkpoint

- 5.6 Name the three expressions that appear inside the parentheses in the `for` loop's header.
- 5.7 You want to write a `for` loop that displays "I love to program" 50 times. Assume you will use a counter variable named `count`.
- What initialization expression will you use?
  - What test expression will you use?
  - What update expression will you use?
  - Write the loop.
- 5.8 What will the following program segments display?
- ```
for (int count = 0; count < 6; count++)
    cout << (count + count);
```
 - ```
for (int value = -5; value < 5; value++)
 cout << value;
```
  - ```
int x;
for (x = 5; x <= 14; x += 3)
    cout << x << endl;
cout << x << endl;
```
- 5.9 Write a `for` loop that displays your name 10 times.
- 5.10 Write a `for` loop that displays all of the odd numbers, 1 through 49.
- 5.11 Write a `for` loop that displays every fifth number, 0 through 100.

5.7

Keeping a Running Total

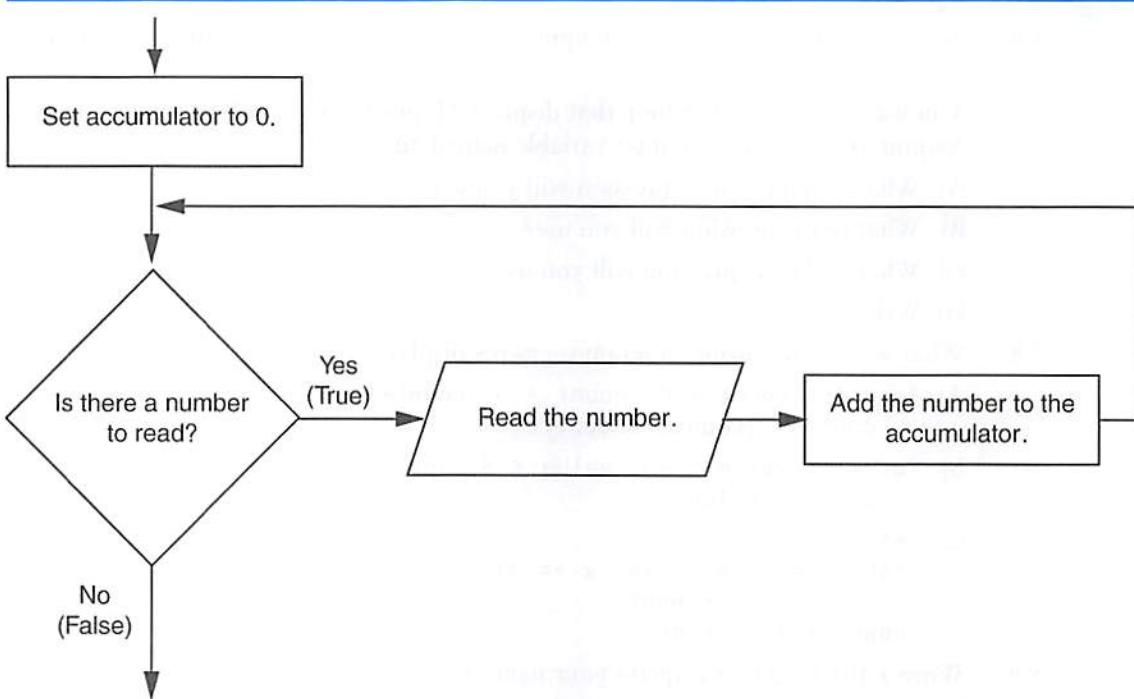
CONCEPT: A *running total* is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an *accumulator*.

Many programming tasks require you to calculate the total of a series of numbers. For example, suppose you are writing a program that calculates a business's total sales for a week. The program would read the sales for each day as input and calculate the total of those numbers.

Programs that calculate the total of 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 5-11 shows the general logic of a loop that calculates a running total.

Figure 5-11 Logic for calculating a running total

When the loop finishes, the accumulator will contain the total of the numbers that were read by the loop. Notice 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 calculates a running total. Program 5-12 calculates a company's total sales over a period of time by taking daily sales figures as input and calculating a running total of them as they are gathered.

Program 5-12

```

1 // This program takes daily sales amounts over a period of time
2 // and calculates their total.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     int days;           // Number of days
10    double total = 0.0; // Accumulator, initialized with 0
11
12    // Get the number of days.
13    cout << "For how many days do you have sales amounts? ";
  
```

```

14     cin >> days;
15
16     // Get the sales for each day and accumulate a total.
17     for (int count = 1; count <= days; count++)
18     {
19         double sales;
20         cout << "Enter the sales for day " << count << ": ";
21         cin >> sales;
22         total += sales; // Accumulate the running total.
23     }
24
25     // Display the total sales.
26     cout << fixed << showpoint << setprecision(2);
27     cout << "The total sales are $" << total << endl;
28     return 0;
29 }
```

Program Output with Example Input Shown in Bold

For how many days do you have sales amounts? **5**

Enter the sales for day 1: **489.32**

Enter the sales for day 2: **421.65**

Enter the sales for day 3: **497.89**

Enter the sales for day 4: **532.37**

Enter the sales for day 5: **506.92**

The total sales are \$2448.15

Let's take a closer look at this program. Line 9 defines the `days` variable, which will hold the number of days for which we have sales figures. Line 10 defines the `total` variable, which will hold the total sales. Because `total` is an accumulator, it is initialized with 0.0.

In line 14, the user enters the number of days for which he or she has sales amounts. The number is assigned to the `days` variable. Next, the `for` loop in lines 17 through 23 executes. In the loop's initialization expression, in line 17, the variable `count` is defined and initialized with 1. The test expression specifies the loop will repeat as long as `count` is less than or equal to `days`. The update expression increments `count` by one at the end of each loop iteration.

Line 19 defines a variable named `sales`. Because the variable is defined in the body of the loop, its scope is limited to the loop. During each loop iteration, the user enters the amount of sales for a specific day, which is assigned to the `sales` variable. This is done in line 21. Then, in line 22, the value of `sales` is added to the existing value in the `total` variable. (Note line 22 does *not* assign `sales` to `total`, but *adds* `sales` to `total`. Put another way, this line increases `total` by the amount in `sales`.)

Because `total` was initially assigned 0.0, after the first iteration of the loop, `total` will be set to the same value as `sales`. After each subsequent iteration, `total` will be increased by the amount in `sales`. After the loop has finished, `total` will contain the total of all the daily sales figures entered. Now it should be clear why we assigned 0.0 to `total` before the loop executed. If `total` started at any other value, the total would be incorrect.

5.8

Sentinels

CONCEPT: A *sentinel* is a special value that marks the end of a list of values.

Program 5-12, in the previous section, requires the user to know in advance the number of days for which he or she wishes to enter sales figures. 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 5-13 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 -1 to signal the end of the list.

Program 5-13

```

1 // This program calculates the total number of points a
2 // soccer team has earned over a series of games. The user
3 // enters a series of 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,        // To hold a number of points
11    total = 0;      // Accumulator
12
13    cout << "Enter the number of points your team has earned\n";
14    cout << "so far in the season, then enter -1 when finished.\n\n";
15    cout << "Enter the points for game " << game << ": ";
16    cin >> points;
17
18    while (points != -1)
19    {
20        total += points;
21        game++;
22        cout << "Enter the points for game " << game << ": ";
23        cin >> points;
24    }
25    cout << "\nThe total points are " << total << endl;
26    return 0;
27 }
```

Program Output with Example Input Shown in Bold

Enter the number of points your team has earned so far in the season, then enter -1 when finished.

```
Enter the points for game 1: 7 Enter  
Enter the points for game 2: 9 Enter  
Enter the points for game 3: 4 Enter  
Enter the points for game 4: 6 Enter  
Enter the points for game 5: 8 Enter  
Enter the points for game 6: -1 Enter
```

The total points are 34

The value -1 was chosen for the sentinel in this program because it is not possible for a team to score negative points. Notice this program performs a priming read in line 18 to get the first value. This makes it possible for the loop to immediately terminate if the user enters -1 as the first value. Also note the sentinel value is not included in the running total.

**Checkpoint**

5.12 Write a **for** loop that repeats seven times, asking the user to enter a number. The loop should also calculate the sum of the numbers entered.

5.13 In the following program segment, which variable is the counter variable and which is the accumulator?

```
int a, x, y = 0;  
for (x = 0; x < 10; x++)  
{  
    cout << "Enter a number: ";  
    cin >> a;  
    y += a;  
}  
cout << "The sum of those numbers is " << y << endl;
```

5.14 Why should you be careful when choosing a sentinel value?

5.15 How would you modify Program 5-13 so any negative value is a sentinel?

5.9**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 the three C++ loops is 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 conditional loop, which means it repeats as long as a particular condition exists. It is also a pretest loop, so it is ideal in situations where you do not want the loop to iterate if the 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.

- **The `do-while` loop.** The `do-while` loop is also a conditional loop. Unlike the `while` loop, however, `do-while` is a posttest 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.
- **The `for` loop.** The `for` loop is a pretest loop that has built-in expressions for initializing, testing, and updating. These expressions make it very convenient to use a counter variable to control the number of iterations that the loop performs. The initialization expression can initialize the counter variable to a starting value, the test expression can test the counter variable to determine whether it holds the maximum value, and the update expression can increment the counter variable. The `for` loop is ideal in situations where the exact number of iterations is known.

5.10 Nested Loops

CONCEPT: A loop that is inside another loop is called a *nested loop*.

A nested loop is a loop that appears inside another loop. A clock is a good example of something that works like a nested loop. The second hand, minute hand, and hour hand all spin around the face of the clock. Each time the hour hand increments, the minute hand increments 60 times. Each time the minute hand increments, the second hand increments 60 times.

Here is a program segment with a `for` loop that partially simulates a digital clock. It displays the seconds from 0 to 59.

```
cout << fixed << right;
cout.fill('0');
for (int seconds = 0; seconds < 60; seconds++)
    cout << setw(2) << seconds << endl;
```



NOTE: The `fill` member function of `cout` changes the fill character, which is a space by default. In the program segment above, the `fill` function causes a zero to be printed in front of all single digit numbers.

We can add a `minutes` variable and nest the loop above inside another loop that cycles through 60 minutes.

```
cout << fixed << right;
cout.fill('0');
for (int minutes = 0; minutes < 60; minutes++)
{
    for (int seconds = 0; seconds < 60; seconds++)
    {
        cout << setw(2) << minutes << ":";
        cout << setw(2) << seconds << endl;
    }
}
```

To make the simulated clock complete, another variable and loop can be added to count the hours.

```
cout << fixed << right;
cout.fill('0');
for (int hours = 0; hours < 24; hours++)
{
    for (int minutes = 0; minutes < 60; minutes++)
    {
        for (int seconds = 0; seconds < 60; seconds++)
        {
            cout << setw(2) << hours << ":";
            cout << setw(2) << minutes << ":";
            cout << setw(2) << seconds << endl;
        }
    }
}
```

The output of the previous program segment follows:

```
00:00:00
00:00:01
00:00:02
```

. . . (The program will count through each second of 24 hours.)

```
23:59:59
```

The innermost loop will iterate 60 times for each iteration of the middle loop. The middle loop will iterate 60 times for each iteration of the outermost loop. When the outermost loop has iterated 24 times, the middle loop will have iterated 1,440 times and the innermost loop will have iterated 86,400 times!

The simulated clock example brings up a few points about nested loops:

- 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 a nested loop, multiply the number of iterations of all the loops.

Program 5-14 is another test-averaging program. It asks the user for the number of students and the number of test scores per student. A nested inner loop, in lines 26 through 33, asks for all the test scores for one student, iterating once for each test score. The outer loop in lines 23 through 37 iterates once for each student.

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 #include <iomanip>
5 using namespace std;
6
```

(program continues)

Program 5-14

(continued)

```

7  int main()
8  {
9      int numStudents,    // Number of students
10     numTests;        // Number of tests per student
11     double total,      // Accumulator for total scores
12     average;         // Average test score
13
14     // Set up numeric output formatting.
15     cout << fixed << showpoint << setprecision(1);
16
17     // Get the number of students.
18     cout << "This program averages test scores.\n";
19     cout << "For how many students do you have scores? ";
20     cin >> numStudents;
21
22     // Get the number of test scores per student.
23     cout << "How many test scores does each student have? ";
24     cin >> numTests;
25
26     // Determine each student's average score.
27     for (int student = 1; student <= numStudents; student++)
28     {
29         total = 0;      // Initialize the accumulator.
30         for (int test = 1; test <= numTests; test++)
31         {
32             double score;
33             cout << "Enter score " << test << " for ";
34             cout << "student " << student << ": ";
35             cin >> score;
36             total += score;
37         }
38         average = total / numTests;
39         cout << "The average score for student " << student;
40         cout << " is " << average << ".\n\n";
41     }
42     return 0;
43 }
```

Program Output with Example Input Shown in Bold

This program averages test scores.

For how many students do you have scores? **2**

How many test scores does each student have? **3**

Enter score 1 for student 1: **84**

Enter score 2 for student 1: **79**

Enter score 3 for student 1: **97**

The average score for student 1 is 86.7.

Enter score 1 for student 2: **92**

Enter score 2 for student 2: **88**

Enter score 3 for student 2: **94**

The average score for student 2 is 91.3.

5.11 Using Files for Data Storage

CONCEPT: When a program needs to save data for later use, it writes the data in a file. The data can then be read from the file at a later time.

The programs you have written so far require the user to reenter data each time the program runs, because data kept in variables and control properties is stored in RAM and disappears once the program stops running. 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 in a file, it will remain there after the program stops running. Data stored in a file can be then retrieved and used at a later time.

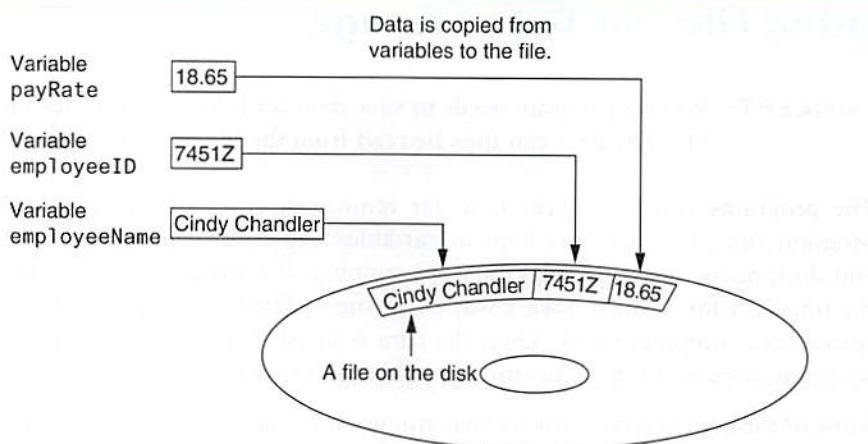
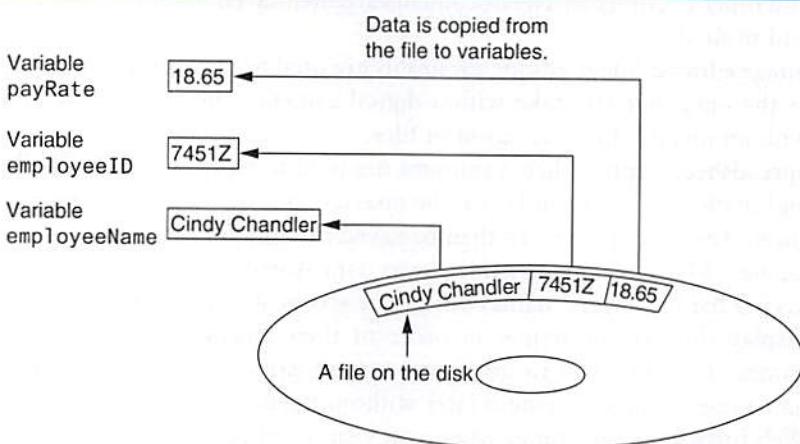
Most of the commercial software that you use on a day-to-day basis store data in files. The following are a few examples:

- **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 later be edited and printed.
- **Image editors:** Image editing programs are used to draw graphics and edit images such as the ones that you take with a digital camera. The images that you create or edit with an image editor are saved in files.
- **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 in a file for later use.
- **Games:** Many computer games keep data stored in files. For example, some games keep a list of players' names with their scores stored in a file. These games typically display the players' names in order of their scores, from highest to lowest. Some games also allow you to save your current game status in a file so you can quit the game then resume playing it later without having to start from the beginning.
- **Web browsers:** Sometimes when you visit a webpage, the browser stores a small file known as a *cookie* on your computer. Cookies typically contain information about the browsing session, such as the contents of a shopping cart.

Programs used in daily business operations rely extensively on files. Payroll programs keep employee data in files, inventory programs keep data about a company's products in files, accounting systems keep data about a company's financial operations in files, and so on.

Programmers usually refer to the process of saving data in a file as *writing data* to the file. When a piece of data is written to a file, it is copied from a variable in RAM to the file. This is illustrated in Figure 5-12. An *output file* is a file to which data is written. It is called an output file because the program stores output in it.

The process of retrieving data from a file is known as *reading data* from the file. When a piece of data is read from a file, it is copied from the file into a variable in RAM. Figure 5-13 illustrates this process. An *input file* is a file from which data is read. It is called an input file because the program gets input from the file.

Figure 5-12 Writing data to a file**Figure 5-13** Reading data from a file

This section discusses ways to create programs that write data to files and read data from files. When a file is used by a program, three steps must be taken.

1. **Open the file**—Opening a file creates a connection between the file and the program. Opening an output file usually creates the file on the disk, and allows the program to write data to it. Opening an input file allows the program to read data from the file.
2. **Process the file**—Data is either written to the file (if it is an output file) or read from the file (if it is an input file).
3. **Close the file**—After the program is finished using the file, the file must be closed. Closing a file disconnects the file from the program.

Types of Files

In general, there are two types of files: text and binary. A *text file* contains data that has been encoded as text, using a scheme such as ASCII or Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. As a result, the file may be opened and viewed in a text editor such as Notepad. A *binary file* contains data

that has not been converted to text. Thus, you cannot view the contents of a binary file with a text editor. In this chapter, we work only with text files. In Chapter 12, you will learn to work with binary files.

File Access Methods

There are two general ways to access data stored in a file: sequential access and direct access. When you work with a *sequential-access file*, you access data from the beginning of the file to the end of the file. If you want to read a piece of data that is stored at the very end of the file, you have to read all of the data that comes before it—you cannot jump directly to the desired data. This is similar to the way that older cassette tape players work. If you want to listen to the last song on a cassette tape, you have to either fast-forward over all of the songs that come before it or listen to them. There is no way to jump directly to a specific song.

When you work with a *random-access file* (also known as a *direct access file*), you can jump directly to any piece of data in the file without reading the data that comes before it. This is similar to the way a CD player or an MP3 player works. You can jump directly to any song that you want to hear.

This chapter focuses on sequential-access files. Sequential-access files are easy to work with, and you can use them to gain an understanding of basic file operations. In Chapter 12, you will learn to work with random-access files.

Filenames and File Stream Objects

Files on a disk are identified by a *filename*. For example, when you create a document with a word processor then save the document in a file, you have to specify a filename. When you use a utility such as Windows Explorer to examine the contents of your disk, you see a list of filenames. Figure 5-14 shows how three files named cat.jpg, notes.txt, and resume.doc might be represented in Windows Explorer.

Figure 5-14 Three files



Each operating system has its own rules for naming files. Many systems, including Windows, support the use of *filename extensions*, which are short sequences of characters that appear at the end of a filename preceded by a period (known as a “dot”). For example, the files depicted in Figure 5-14 have the extensions .jpg, .txt, and .doc. The extension usually indicates the type of data stored in the file. For example, the .jpg extension usually indicates that the file contains a graphic image that is compressed according to the JPEG image standard. The .txt extension usually indicates that the file contains text. The .doc extension usually indicates that the file contains a Microsoft Word document.

In order for a program to work with a file on the computer’s disk, the program must create a file stream object in memory. A *file stream object* is an object that is associated with a specific file and provides a way for the program to work with that file. It is called a “stream” object because a file can be thought of as a stream of data.

File stream objects work very much like the `cin` and `cout` objects. A stream of data may be sent to `cout`, which causes values to be displayed on the screen. A stream of data may be read from the keyboard by `cin`, and stored in variables. Likewise, streams of data may be sent to a file stream object, which writes the data to a file. When data is read from a file, the data flows from the file stream object that is associated with the file into variables.

Setting Up a Program for 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 declarations necessary for file operations. It is included with the following statement:

```
#include <fstream>
```

The `<fstream>` header file defines 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 “linked” with an actual file on the computer’s disk, and the operations that may be performed on the file depend on which of these three data types you pick for the file stream object. Table 5-1 lists and describes the file stream data types.

Table 5-1 File Stream Objects

File Stream Data Type	Description
<code>ofstream</code>	Output file stream. You create an object of this data type when you want to create a file and write data to it.
<code>ifstream</code>	Input file stream. You create an object of this data type when you want to open an existing file and read data from it.
<code>fstream</code>	File stream. Objects of this data type can be used to open files for reading, writing, or both.



NOTE: In this chapter, we discuss only the `ofstream` and `ifstream` types. The `fstream` type will be covered in Chapter 12.

Creating a File Object and Opening a File

Before data can be written to or read from a file, the following things must happen:

- A file stream object must be created.
- The file must be opened and linked to the file stream object.

The following code shows an example of opening a file for input (reading).

```
ifstream inputFile;
inputFile.open("Customers.txt");
```

The first statement defines an `ifstream` object named `inputFile`. The second statement calls the object’s `open` member function, passing the string `"Customers.txt"` as an argument. In this statement, the `open` member function opens the `Customers.txt` file and links it with the `inputFile` object. After this code executes, you will be able to use the `inputFile` object to read data from the `Customers.txt` file.

The following code shows an example of opening a file for output (writing).

```
ofstream outputFile;  
outputFile.open("Employees.txt");
```

The first statement defines an `ofstream` object named `outputFile`. The second statement calls the object's `open` member function, passing the string "Employees.txt" as an argument. In this statement, the `open` member function creates the `Employees.txt` file and links it with the `outputFile` object. After this code executes, you will be able to use the `outputFile` object to write data to the `Employees.txt` file. It's important to remember that when you call an `ofstream` object's `open` member function, the specified file will be created. If the specified file already exists, it will be deleted, and a new file with the same name will be created.

Often, when opening a file, you will need to specify its path as well as its name. For example, on a Windows system the following statement opens the file `C:\data\inventory.txt`:

```
inputFile.open("C:\\data\\inventory.txt")
```

In this statement, the file `C:\data\inventory.txt` is opened and linked with `inputFile`.



NOTE: Notice the use of two backslashes in the file's path. Two backslashes are needed to represent one backslash in a string literal.

It is possible to define a file stream object and open a file in one statement. Here is an example:

```
ifstream inputFile("Customers.txt");
```

This statement defines an `ifstream` object named `inputFile` and opens the `Customer.txt` file. Here is an example that defines an `ofstream` object named `outputFile` and opens the `Employees.txt` file:

```
ofstream outputFile("Employees.txt");
```

Closing a File

The opposite of opening a file is closing it. Although a program's files are automatically closed when the program shuts down, it is a good programming practice to write statements that close them. Here are two reasons a program should close files when it is finished using them:

1. Most operating systems temporarily store data in a *file buffer* before it is written to a file. A file buffer is a small “holding section” of memory to which file-bound data is first written. When the buffer is filled, all the data stored there is written to the file. This technique improves the system's performance. Closing a file causes any unsaved data that may still be held in a buffer to be saved to its file. This means the data will be in the file if you need to read it later in the same program.
2. Some operating systems limit the number of files that may be open at one time. When a program closes files that are no longer being used, it will not deplete more of the operating system's resources than necessary.

Calling the file stream object's `close` member function closes a file. Here is an example:

```
inputFile.close();
```

Writing Data to a File

You already know how to use the stream insertion operator (`<<`) with the `cout` object to write data to the screen. It can also be used with `ofstream` objects to write data to a file. Assuming `outputFile` is an `ofstream` object, the following statement demonstrates using the `<<` operator to write a string literal to a file:

```
outputFile << "I love C++ programming\n";
```

This statement writes the string literal "I love C++ programming\n" to the file associated with `outputFile`. As you can see, the statement looks like a `cout` statement, except the name of the `ofstream` object name replaces `cout`. Here is a statement that writes both a string literal and the contents of a variable to a file:

```
outputFile << "Price: " << price << endl;
```

The statement above writes the stream of data to `outputFile` exactly as `cout` would write it to the screen: It writes the string "Price: ", followed by the value of the `price` variable, followed by a newline character.

Program 5-15 demonstrates opening a file, writing data to the file, and closing the file. After this code has executed, we can open the `demofile.txt` file using a text editor and look at its contents. Figure 5-15 shows how the file's contents will appear in Notepad.

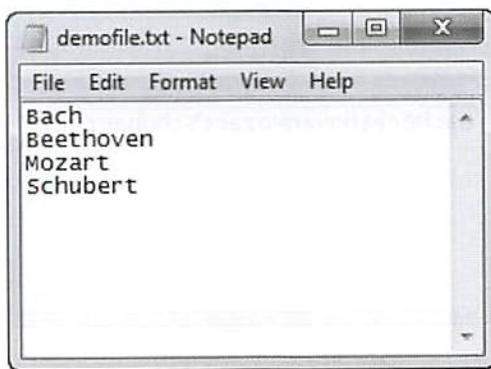
Program 5-15

```

1 // This program writes data to a file.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     ofstream outputFile;
9     outputFile.open("demofile.txt");
10
11    cout << "Now writing data to the file.\n";
12
13    // Write four names to the file.
14    outputFile << "Bach\n";
15    outputFile << "Beethoven\n";
16    outputFile << "Mozart\n";
17    outputFile << "Schubert\n";
18
19    // Close the file
20    outputFile.close();
21    cout << "Done.\n";
22    return 0;
23 }
```

Program Screen Output

Now writing data to the file.
Done.

Figure 5-15 Contents of demofile.txt

Notice in lines 14 through 17 of Program 5-15, each string that was written to the file ends with a newline escape sequence (\n). The newline specifies the end of a line of text. Because a newline is written at the end of each string, the strings appear on separate lines when viewed in a text editor, as shown in Figure 5-15.

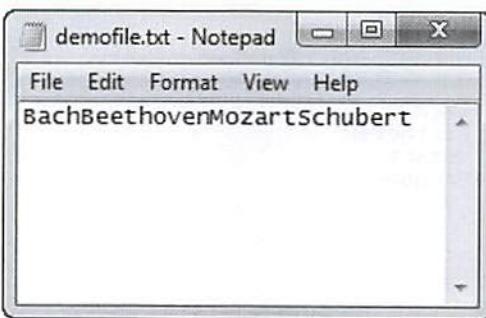
Program 5-16 shows what happens if we write the same four names without the \n escape sequence. Figure 5-16 shows the contents of the file that Program 5-16 creates. As you can see, all of the names appear on the same line in the file.

Program 5-16

```
1 // This program writes data to a single line in a file.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     ofstream outputFile;
9     outputFile.open("demofile.txt");
10
11    cout << "Now writing data to the file.\n";
12
13    // Write four names to the file.
14    outputFile << "Bach";
15    outputFile << "Beethoven";
16    outputFile << "Mozart";
17    outputFile << "Schubert";
18
19    // Close the file
20    outputFile.close();
21    cout << "Done.\n";
22
23 }
```

Program Screen Output

```
Now writing data to the file.
Done.
```

Figure 5-16 Output written as one line

Program 5-17 shows another example. This program reads three numbers from the keyboard as input then saves those numbers in a file named Numbers.txt.

Program 5-17

```

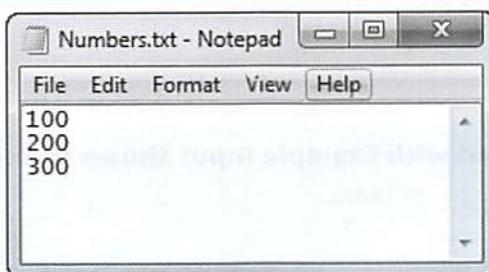
1 // This program writes user input to a file.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     ofstream outputFile;
9     int number1, number2, number3;
10
11    // Open an output file.
12    outputFile.open("Numbers.txt");
13
14    // Get three numbers from the user.
15    cout << "Enter a number: ";
16    cin >> number1;
17    cout << "Enter another number: ";
18    cin >> number2;
19    cout << "One more time. Enter a number: ";
20    cin >> number3;
21
22    // Write the numbers to the file.
23    outputFile << number1 << endl;
24    outputFile << number2 << endl;
25    outputFile << number3 << endl;
26    cout << "The numbers were saved to a file.\n";
27
28    // Close the file
29    outputFile.close();
30    cout << "Done.\n";
31
32 }
```

Program Screen Output with Example Input Shown in Bold

```
Enter a number: 100 Enter  
Enter another number: 200 Enter  
One more time. Enter a number: 300 Enter  
The numbers were saved to a file.  
Done.
```

In Program 5-17, lines 23 through 25 write the contents of the `number1`, `number2`, and `number3` variables to the file. Notice the `endl` manipulator is sent to the `outputFile` object immediately after each item. Sending the `endl` manipulator causes a newline to be written to the file. Figure 5-17 shows the file's contents displayed in Notepad, using the example input values 100, 200, and 300. As you can see, each item appears on a separate line in the file because of the `endl` manipulators.

Figure 5-17 Contents of Numbers.txt



Program 5-18 shows an example that reads strings as input from the keyboard then writes those strings to a file. The program asks the user to enter the first names of three friends, then it writes those names to a file named Friends.txt. Figure 5-18 shows an example of the Friends.txt file opened in Notepad.

Program 5-18

```
1 // This program writes user input to a file.  
2 #include <iostream>  
3 #include <fstream>  
4 #include <string>  
5 using namespace std;  
6  
7 int main()  
8 {  
9     ofstream outputFile;  
10    string name1, name2, name3;  
11  
12    // Open an output file.  
13    outputFile.open("Friends.txt");  
14}
```

(program continues)

Program 5-18

(continued)

```

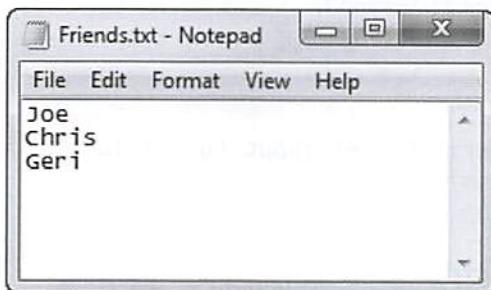
15     // Get the names of three friends.
16     cout << "Enter the names of three friends.\n";
17     cout << "Friend #1: ";
18     cin >> name1;
19     cout << "Friend #2: ";
20     cin >> name2;
21     cout << "Friend #3: ";
22     cin >> name3;
23
24     // Write the names to the file.
25     outputFile << name1 << endl;
26     outputFile << name2 << endl;
27     outputFile << name3 << endl;
28     cout << "The names were saved to a file.\n";
29
30     // Close the file
31     outputFile.close();
32     return 0;
33 }
```

Program Screen Output with Example Input Shown in Bold

Enter the names of three friends.

Friend #1: **Joe** Friend #2: **Chris** Friend #3: **Geri**

The names were saved to a file.

Figure 5-18 Contents of Friends.txt**Reading Data from a File**

The `>>` operator reads not only user input from the `cin` object, but also data from a file. Assuming `inputFile` is an `if stream` object, the following statement shows the `>>` operator reading data from the file into the variable `name`:

```
inputFile >> name;
```

Let's look at an example. Assume the file `Friends.txt` exists, and it contains the names shown in Figure 5-18. Program 5-19 opens the file, reads the names and displays them on the screen, then closes the file.

Program 5-19

```
1 // This program reads data from a file.
2 #include <iostream>
3 #include <fstream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     ifstream inputFile;
10    string name;
11
12    inputFile.open("Friends.txt");
13    cout << "Reading data from the file.\n";
14
15    inputFile >> name;           // Read name 1 from the file
16    cout << name << endl;        // Display name 1
17
18    inputFile >> name;           // Read name 2 from the file
19    cout << name << endl;        // Display name 2
20
21    inputFile >> name;           // Read name 3 from the file
22    cout << name << endl;        // Display name 3
23
24    inputFile.close();          // Close the file
25    return 0;
26 }
```

Program Output

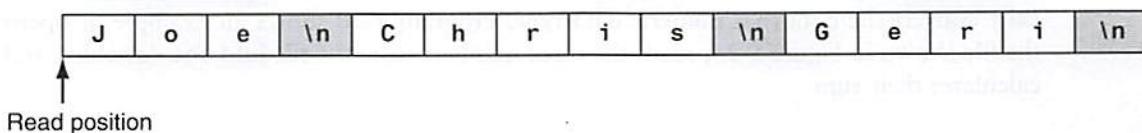
```
Reading data from the file.
Joe
Chris
Geri
```

The Read Position

When a file has been opened for input, the file stream object internally maintains a special value known as a *read position*. A file's read position marks the location of the next byte that will be read from the file. When an input file is opened, its read position is initially set to the first byte in the file. So, the first read operation extracts data starting at the first byte. As data is read from the file, the read position moves forward, toward the end of the file.

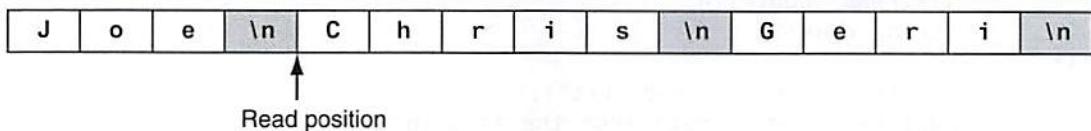
Let's see how this works with the example shown in Program 5-19. When the Friends.txt file is opened by the statement in line 12, the read position for the file will be positioned as shown in Figure 5-19.

Figure 5-19 Initial read position



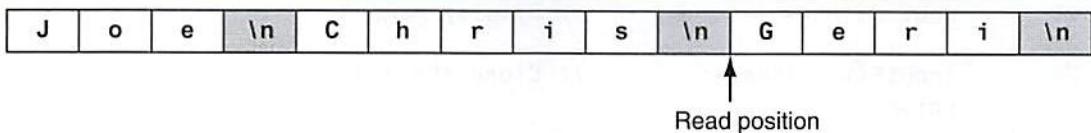
Keep in mind when the `>>` operator extracts data from a file, it expects to read pieces of data that are separated by whitespace characters (spaces, tabs, or newlines). When the statement in line 15 executes, the `>>` operator reads data from the file's current read position, up to the `\n` character. The data read from the file is assigned to the `name` object. The `\n` character is also read from the file, but is not included as part of the data. So, the `name` object will hold the value "Joe" after this statement executes. The file's read position will then be at the location shown in Figure 5-20.

Figure 5-20 The next read position



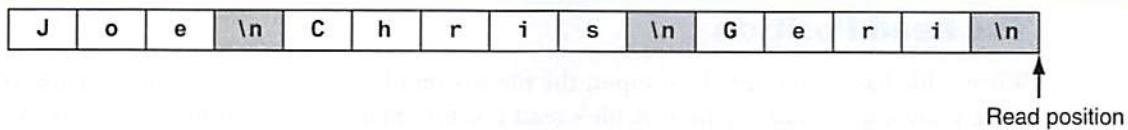
When the statement in line 18 executes, it reads the next item from the file, which is "Chris", and assigns that value to the `name` object. After this statement executes, the file's read position will be advanced to the next item, as shown in Figure 5-21.

Figure 5-21 The next read position



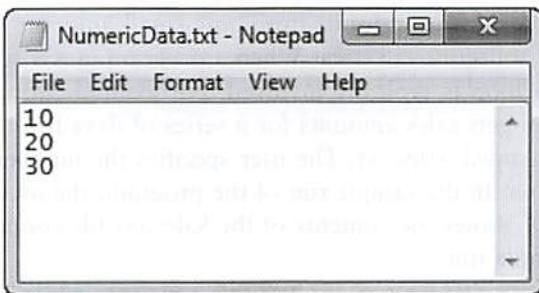
When the statement in line 21 executes, it reads the next item from the file, which is "Geri", and assigns that value to the `name` object. After this statement executes, the file's read position will be advanced to the end of the file, as shown in Figure 5-22.

Figure 5-22 The read position at the end of the file



Reading Numeric Data from a Text File

Remember when data is stored in a text file, it is encoded as text, using a scheme such as ASCII or Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. For example, suppose a text file contains numeric data, such as that shown in Figure 5-17. The numbers that you see displayed in the figure are stored in the file as the strings "10", "20", and "30". Fortunately, you can use the `>>` operator to read data such as this from a text file into a numeric variable, and the `>>` operator will automatically convert the data to a numeric data type. Program 5-20 shows an example. It opens the file shown in Figure 5-23, reads the three numbers from the file into `int` variables, and calculates their sum.

Figure 5-23 Contents of NumericData.txt**Program 5-20**

```
1 // This program reads numbers from a file.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     ifstream inFile;
9     int value1, value2, value3, sum;
10
11    // Open the file.
12    inFile.open("NumericData.txt");
13
14    // Read the three numbers from the file.
15    inFile >> value1;
16    inFile >> value2;
17    inFile >> value3;
18
19    // Close the file.
20    inFile.close();
21
22    // Calculate the sum of the numbers.
23    sum = value1 + value2 + value3;
24
25    // Display the three numbers.
26    cout << "Here are the numbers:\n"
27        << value1 << " " << value2
28        << " " << value3 << endl;
29
30    // Display the sum of the numbers.
31    cout << "Their sum is: " << sum << endl;
32
33 }
```

Program Output

Here are the numbers:
10 20 30
Their sum is: 60

Using Loops to Process Files

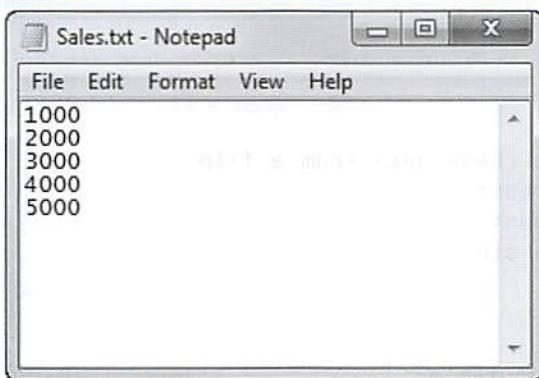
Although some programs use files to store only small amounts of data, files are typically used to hold large collections of data. When a program uses a file to write or read a large amount of data, a loop is typically involved. For example, look at the code in Program 5-21. This program gets sales amounts for a series of days from the user and writes those amounts to a file named Sales.txt. The user specifies the number of days of sales data he or she needs to enter. In the sample run of the program, the user enters sales amounts for 5 days. Figure 5-24 shows the contents of the Sales.txt file containing the data entered by the user in the sample run.

Program 5-21

```
1 // This program reads data from a file.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     ofstream outputFile; // File stream object
9     int numberOfDays; // Number of days of sales
10    double sales; // Sales amount for a day
11
12    // Get the number of days.
13    cout << "For how many days do you have sales? ";
14    cin >> numberOfDays;
15
16    // Open a file named Sales.txt.
17    outputFile.open("Sales.txt");
18
19    // Get the sales for each day and write it
20    // to the file.
21    for (int count = 1; count <= numberOfDays; count++)
22    {
23        // Get the sales for a day.
24        cout << "Enter the sales for day "
25            << count << ": ";
26        cin >> sales;
27
28        // Write the sales to the file.
29        outputFile << sales << endl;
30    }
31
32    // Close the file.
33    outputFile.close();
34    cout << "Data written to Sales.txt\n";
35
36 }
```

```
Program Output (with Input Shown in Bold)
For how many days do you have sales? 5 [Enter]
Enter the sales for day 1: 1000.00 [Enter]
Enter the sales for day 2: 2000.00 [Enter]
Enter the sales for day 3: 3000.00 [Enter]
Enter the sales for day 4: 4000.00 [Enter]
Enter the sales for day 5: 5000.00 [Enter]
Data written to sales.txt.
```

Figure 5-24 Contents of Sales.txt

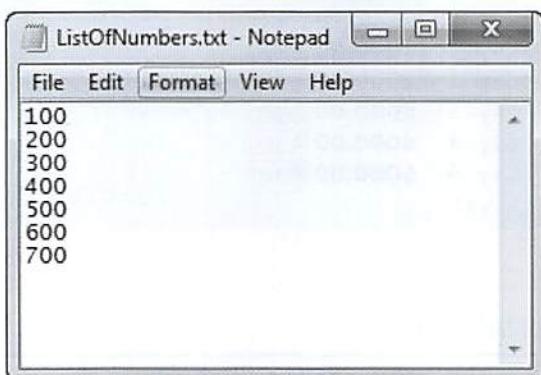


Detecting the End of the File

Quite often a program must read the contents of a file without knowing the number of items that are stored in the file. For example, suppose you need to write a program that displays all of the items in a file, but you do not know how many items the file contains. You can open the file then use a loop to repeatedly read an item from the file and display it. However, an error will occur if the program attempts to read beyond the end of the file. The program needs some way of knowing when the end of the file has been reached so it will not try to read beyond it.

Fortunately, the `>>` operator not only reads data from a file, but also returns a true or false 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.

Let's look at an example. A file named `ListOfNumbers.txt`, which is shown in Figure 5-25, contains a list of numbers. Without knowing how many numbers the file contains, Program 5-22 opens the file, reads all of the values it contains, and displays them.

Figure 5-25 Contents of ListOfNumbers.txt**Program 5-22**

```
1 // This program reads data from a file.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     ifstream inputFile;
9     int number;
10
11    // Open the file.
12    inputFile.open("ListOfNumbers.txt");
13
14    // Read the numbers from the file and
15    // display them.
16    while (inputFile >> number)
17    {
18        cout << number << endl;
19    }
20
21    // Close the file.
22    inputFile.close();
23    return 0;
24 }
```

Program Output

```
100
200
300
400
500
600
700
```

Take a closer look at line 16:

```
while (inputFile >> number)
```

Notice the statement that extracts data from the file is used as the Boolean expression in the while loop. It works like this:

- The expression `inputFile >> number` executes.
- If an item is successfully read from the file, the item is stored in the `number` variable, and the expression returns true to indicate that it succeeded. In that case, the statement in line 18 executes and the loop repeats.
- If there are no more items to read from the file, the expression `inputFile >> number` returns false, indicating that it did not read a value. In that case, the loop terminates.

Because the value returned from the `>>` operator controls the loop, it will read items from the file until the end of the file has been reached.

Testing for File Open Errors

Under certain circumstances, 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");
```

There is a way to determine whether the `open` member function successfully opened the file. After you call the `open` member function, you can test the file stream object as if it were a Boolean expression. Program 5-23 shows an example.

Program 5-23

```
1 // This program tests for file open errors.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     ifstream inputFile;
9     int number;
10
11    // Open the file.
12    inputFile.open("BadListOfNumbers.txt");
13
14    // If the file successfully opened, process it.
15    if (inputFile)
16    {
17        // Read the numbers from the file and
18        // display them.
19        while (inputFile >> number)
20        {
21            cout << number << endl;
22        }
23    }
```

(program continues)

Program 5-23 (continued)

```

24         // Close the file.
25         inputFile.close();
26     }
27     else
28     {
29         // Display an error message.
30         cout << "Error opening the file.\n";
31     }
32     return 0;
33 }
```

Program Output (Assume BadListOfNumbers.txt does not exist)

Error opening the file.

Let's take a closer look at certain parts of the code. Line 12 calls the `inputFile` object's `open` member function to open the file `ListOfNumbers.txt`. Then, the `if` statement in line 15 tests the value of the `inputFile` object as if it were a Boolean expression. When tested this way, the `inputFile` object will give a true value if the file was successfully opened. Otherwise, it will give a false value. The example output shows this program will display an error message if it could not open the file.

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 << "Error opening file.\n";
}
else
{
    // Process the file.
}
```

The `fail` member function returns true when 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.

Letting the User Specify a Filename

11

In each of the previous examples, the name of the file that is opened is hard-coded as a string literal into the program. In many cases, you will want the user to specify the name of a file for the program to open. In C++ 11, you can pass a `string` object as an argument to a file stream object's `open` member function. Program 5-24 shows an example. This is a modified version of Program 5-23. This version prompts the user to enter the name of the file. In line 15, the name the user enters is stored in a `string` object named `filename`. In line 18, the `filename` object is passed as an argument to the `open` function.

Program 5-24

```
1 // This program lets the user enter a filename.
2 #include <iostream>
3 #include <string>
4 #include <fstream>
5 using namespace std;
6
7 int main()
8 {
9     ifstream inputFile;
10    string filename;
11    int number;
12
13    // Get the filename from the user.
14    cout << "Enter the filename: ";
15    cin >> filename;
16
17    // Open the file.
18    inputFile.open(filename);
19
20    // If the file successfully opened, process it.
21    if (inputFile)
22    {
23        // Read the numbers from the file and
24        // display them.
25        while (inputFile >> number)
26        {
27            cout << number << endl;
28        }
29
30        // Close the file.
31        inputFile.close();
32    }
33    else
34    {
35        // Display an error message.
36        cout << "Error opening the file.\n";
37    }
38
39 }
```

Program Output with Example Input Shown in Bold

Enter the filename: **ListOfNumbers.txt**

```
100
200
300
400
500
600
700
```

Using the `c_str` Member Function in Older Versions of C++

In older versions of the C++ language (prior to C++ 11), a file stream object's `open` member function will not accept a `string` object as an argument. The `open` member function requires you pass the name of the file as a null-terminated string, which is also known as a *C-string*. String literals are stored in memory as null-terminated C-strings, but `string` objects are not.

Fortunately, `string` objects have a member function named `c_str` that returns the contents of the object formatted as a null-terminated C-string. Here is the general format of how you call the function:

```
stringObject.c_str()
```

In the general format, `stringObject` is the name of a `string` object. The `c_str` function returns the string that is stored in `stringObject` as a null-terminated C-string.

For example, line 18 in Program 5-24 could be rewritten in the following manner to make the program compatible with an older version of C++:

```
inputFile.open(filename.c_str());
```

In this version of the statement, the value that is returned from `filename.c_str()` is passed as an argument to the `open` function.



Checkpoint

- 5.16 What is an output file? What is an input file?
- 5.17 What three steps must be taken when a file is used by a program?
- 5.18 What is the difference between a text file and a binary file?
- 5.19 What is the difference between sequential access and random access?
- 5.20 What type of file stream object do you create if you want to write data to a file?
- 5.21 What type of file stream object do you create if you want to read data from a file?
- 5.22 Write a short program that uses a `for` loop to write the numbers 1 through 10 to a file.
- 5.23 Write a short program that opens the file created by the program you wrote for Checkpoint 5.22, reads all of the numbers from the file, and displays them.

5.12

Optional Topics: Breaking and Continuing a Loop

CONCEPT: The `break` statement causes a loop to terminate early. The `continue` statement causes a loop to stop its current iteration and begin the next one.



WARNING! Use the `break` and `continue` statements with great caution. Because they bypass the normal condition that controls the loop's iterations, these statements make code difficult to understand and debug. For this reason, you should avoid using `break` and `continue` whenever possible. However, because they are part of the C++ language, we discuss them briefly in this section.

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 = 0;
while (count++ < 10)
{
    cout << count << endl;
    if (count == 5)
        break;
}
```

Program 5-25 uses the `break` statement to interrupt a `for` loop. The program asks the user for a number, then displays the value of that number raised to the powers of 0 through 10. The user can stop the loop at any time by entering Q.

Program 5-25

```
1 // This program raises the user's number to the powers
2 // of 0 through 10.
3 #include <iostream>
4 #include <cmath>
5 using namespace std;
6
7 int main()
8 {
9     double value;
10    char choice;
11
12    cout << "Enter a number: ";
13    cin >> value;
14    cout << "This program will raise " << value;
15    cout << " to the powers of 0 through 10.\n";
16    for (int count = 0; count <= 10; count++)
17    {
18        cout << value << " raised to the power of ";
19        cout << count << " is " << pow(value, count);
20        cout << "\nEnter Q to quit or any other key ";
21        cout << "to continue. ";
22        cin >> choice;
23        if (choice == 'Q' || choice == 'q')
24            break;
25    }
26    return 0;
27 }
```

(program output continues)

Program 5-25 (continued)**Program Output with Example Input Shown in Bold**

```
Enter a number: 2 Enter
This program will raise 2 to the powers of 0 through 10.
2 raised to the power of 0 is 1
Enter Q to quit or any other key to continue. C Enter
2 raised to the power of 1 is 2
Enter Q to quit or any other key to continue. C Enter
2 raised to the power of 2 is 4
Enter Q to quit or any other key to continue. Q Enter
```

Using break in a Nested Loop

In a nested loop, the `break` statement only interrupts the loop in which it is placed. 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 eleventh iteration.

```
for (int row = 0; row < 5; row++)
{
    for (int star = 0; star < 20; star++)
    {
        cout << '*';
        if (star == 10)
            break;
    }
    cout << endl;
}
```

The output of the program segment above is:

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

The continue Statement

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 whether the next iteration will begin. In a `for` loop, `continue` causes the update expression to be executed, 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)
{
    if (testVal == 4)
        continue;
    cout << testVal << " ";
```

This loop looks like it displays the integers 1 through 10. When `testVal` is equal to 4, however, the `continue` statement causes the loop to skip the `cout` statement and begin the next iteration. The output of the loop is

```
1 2 3 5 6 7 8 9 10
```

Program 5-26 demonstrates the `continue` statement. 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 5-26

```
1 // This program calculates the charges for DVD rentals.
2 // Every third DVD is free.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     int dvdCount = 1;      // DVD counter
10    int numDVDs;          // Number of DVDs rented
11    double total = 0.0;    // Accumulator
12    char current;         // Current release, Y or N
13
14    // Get the number of DVDs.
15    cout << "How many DVDs are being rented? ";
16    cin >> numDVDs;
17
18    // Determine the charges.
19    do
20    {
21        if ((dvdCount % 3) == 0)
22        {
23            cout << "DVD #" << dvdCount << " is free!\n";
24            continue; // Immediately start the next iteration
25        }
26        cout << "Is DVD #" << dvdCount;
27        cout << " a current release? (Y/N) ";
28        cin >> current;
29        if (current == 'Y' || current == 'y')
30            total += 3.50;
31        else
32            total += 2.50;
33    } while (dvdCount++ < numDVDs);
34
```

(program continues)

Program 5-26 (continued)

```

35      // Display the total.
36      cout << fixed << showpoint << setprecision(2);
37      cout << "The total is $" << total << endl;
38      return 0;
39  }

```

Program Output with Example Input Shown in Bold

How many DVDs are being rented? **6**
 Is DVD #1 a current release? (Y/N) **y**
 Is DVD #2 a current release? (Y/N) **n**
 DVD #3 is free!
 Is DVD #4 a current release? (Y/N) **n**
 Is DVD #5 a current release? (Y/N) **y**
 DVD #6 is free!
 The total is \$12.00

Case Study: See the Loan Amortization Case Study on the Computer Science Portal at www.pearsonhighered.com/gaddis.

Review Questions and Exercises**Short Answer**

1. Why should you indent the statements in the body of a loop?
2. Describe the difference between pretest loops and posttest loops.
3. Why are the statements in the body of a loop called conditionally executed statements?
4. What is the difference between the `while` loop and the `do-while` loop?
5. Which loop should you use in situations where you wish the loop to repeat until the test expression is false, and the loop should not execute if the test expression is false to begin with?
6. Which loop should you use in situations where you wish the loop to repeat until the test expression is false, but the loop should execute at least one time?
7. Which loop should you use when you know the number of required iterations?
8. Why is it critical that counter variables be properly initialized?
9. Why is it critical that accumulator variables be properly initialized?
10. Why should you be careful not to place a statement in the body of a `for` loop that changes the value of the loop's counter variable?
11. What header file do you need to include in a program that performs file operations?
12. What data type do you use when you want to create a file stream object that can write data to a file?
13. What data type do you use when you want to create a file stream object that can read data from a file?
14. Why should a program close a file when it's finished using it?

15. What is a file's read position? Where is the read position when a file is first opened for reading?

Fill-in-the-Blank

16. To _____ a value means to increase it by one, and to _____ a value means to decrease it by one.
17. When the increment or decrement operator is placed before the operand (or to the operand's left), the operator is being used in _____ mode.
18. When the increment or decrement operator is placed after the operand (or to the operand's right), the operator is being used in _____ mode.
19. The statement or block that is repeated is known as the _____ of the loop.
20. Each repetition of a loop is known as a(n) _____.
21. A loop that evaluates its test expression before each repetition is a(n) _____ loop.
22. A loop that evaluates its test expression after each repetition is a(n) _____ loop.
23. A loop that does not have a way of stopping is a(n) _____ loop.
24. A(n) _____ is a variable that "counts" the number of times a loop repeats.
25. A(n) _____ is a sum of numbers that accumulates with each iteration of a loop.
26. A(n) _____ is a variable that is initialized to some starting value, usually zero, then has numbers added to it in each iteration of a loop.
27. A(n) _____ is a special value that marks the end of a series of values.
28. The _____ loop always iterates at least once.
29. The _____ and _____ loops will not iterate at all if their test expressions are false to start with.
30. The _____ loop is ideal for situations that require a counter.
31. Inside the `for` loop's parentheses, the first expression is the _____, the second expression is the _____, and the third expression is the _____.
32. A loop that is inside another is called a(n) _____ loop.
33. The _____ statement causes a loop to terminate immediately.
34. The _____ statement causes a loop to skip the remaining statements in the current iteration.

Algorithm Workbench

35. Write a `while` loop that lets the user enter a number. The number should be multiplied by 10, and the result stored in the variable `product`. The loop should iterate as long as `product` contains a value less than 100.
36. 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.
37. Write a `for` loop that displays the following set of numbers:
- 0, 10, 20, 30, 40, 50 ... 1000
38. 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.
39. Write a nested loop that displays 10 rows of '#' characters. There should be 15 '#' characters in each row.

40. Convert the following `while` loop to a `do-while` loop:

```
int x = 1;
while (x > 0)
{
    cout << "enter a number: ";
    cin >> x;
}
```

41. Convert the following `do-while` loop to a `while` loop:

```
char sure;
do
{
    cout << "Are you sure you want to quit? ";
    cin >> sure;
} while (sure != 'Y' && sure != 'N');
```

42. Convert the following `while` loop to a `for` loop:

```
int count = 0;
while (count < 50)
{
    cout << "count is " << count << endl;
    count++;
}
```

43. Convert the following `for` loop to a `while` loop:

```
for (int x = 50; x > 0; x--)
{
    cout << x << " seconds to go.\n";
}
```

44. Write code that does the following: Opens an output file with the filename `Numbers.txt`, uses a loop to write the numbers 1 through 100 to the file, then closes the file.

45. Write code that does the following: Opens the `Numbers.txt` file that was created by the code you wrote in Question 44, reads all of the numbers from the file and displays them, then closes the file.

46. Modify the code that you wrote in Question 45 so it adds all of the numbers read from the file and displays their total.

True or False

47. T F The operand of the increment and decrement operators can be any valid mathematical expression.

48. T F The `cout` statement in the following program segment will display 5:

```
int x = 5;
cout << x++;
```

49. T F The `cout` statement in the following program segment will display 5:

```
int x = 5;
cout << ++x;
```

50. T F The `while` loop is a pretest loop.

51. T F The `do-while` loop is a pretest loop.

52. T F The `for` loop is a posttest loop.

53. T F It is not necessary to initialize counter variables.

54. T F All three of the `for` loop's expressions may be omitted.
55. T F One limitation of the `for` loop is that only one variable may be initialized in the initialization expression.
56. T F Variables may be defined inside the body of a loop.
57. T F A variable may be defined in the initialization expression of the `for` loop.
58. T F In a nested loop, the outer loop executes faster than the inner loop.
59. T F In a nested loop, the inner loop goes through all of its iterations for every single iteration of the outer loop.
60. T F To calculate the total number of iterations of a nested loop, add the number of iterations of all the loops.
61. T F The `break` statement causes a loop to stop the current iteration and begin the next one.
62. T F The `continue` statement causes a terminated loop to resume.
63. T F In a nested loop, the `break` statement only interrupts the loop in which it is placed.
64. T F When you call an `ofstream` object's `open` member function, the specified file will be erased if it already exists.

Find the Errors

Each of the following programs has errors. Find as many as you can.

65. // 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;
}
```
66. // 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 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? ";
        cin >> again;
    return 0;
}
```

```
67. // 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;  
}  
  
68. // 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;  
}  
  
69. // 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;
}

70. // 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;
}

```

Programming Challenges

1. 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 a negative starting number.

2. Characters for the ASCII Codes

Write a program that uses a loop to display the characters for the ASCII codes 0 through 127. Display 16 characters on each line.

3. Ocean Levels

Assuming the ocean's level is currently rising at about 1.5 millimeters per year, write a program that displays a table showing the number of millimeters that the ocean will have risen each year for the next 25 years.

4. Calories Burned

Running on a particular treadmill you burn 3.6 calories per minute. Write a program that uses a loop to display the number of calories burned after 5, 10, 15, 20, 25, and 30 minutes.

5. Membership Fees Increase

A country club, which currently charges \$2,500 per year for membership, has announced it will increase its membership fee by 4 percent each year for the next 6 years. Write a program that uses a loop to display the projected rates for the next 6 years.

6. 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. The program should then use a loop to display the distance the vehicle has traveled for 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 1 for time traveled.`

7. Pennies for Pay

Write a program that calculates how much a person would earn over a period of time if his or her salary is one penny the first day and two pennies the second day, and continues to double each day. The program should ask the user for the number of days. Display a table showing how much the salary was for each day, and then show the total pay at the end of the period. The output should be displayed in a dollar amount, not the number of pennies.

`Input Validation: Do not accept a number less than 1 for the number of days worked.`

8. Math Tutor

This program started in Programming Challenge 17, of Chapter 3, and was modified in Programming Challenge 11 of Chapter 4. Modify the program again so it displays a menu allowing the user to select an addition, subtraction, multiplication, or division 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.`

9. Hotel Occupancy

Write a program that calculates the occupancy rate for a hotel. The program should start by asking the user how many floors the hotel has. A loop should then iterate once for each floor. In each iteration, the loop should ask the user for the number of rooms on the floor and how many of them are occupied. After all the iterations, the program should display how many rooms the hotel has, how many of them are occupied, how many are unoccupied, and the percentage of rooms that are occupied. The percentage may be calculated by dividing the number of rooms occupied by the number of rooms.



NOTE: It is traditional that most hotels do not have a thirteenth floor. The loop in this program should skip the entire thirteenth iteration.

Input Validation: Do not accept a value less than 1 for the number of floors. Do not accept a number less than 10 for the number of rooms on a floor.

10. Average Rainfall

Write a program that uses nested loops to collect data and calculate the average rainfall over a period of years. The program should first ask for the number of years. The outer loop will iterate once for each year. The inner loop will iterate 12 times, once for each month. Each iteration of the inner loop will ask the user for the inches of rainfall for that month.

After all iterations, the program should display the number of months, the total inches of rainfall, and the average rainfall per month for the entire period.

Input Validation: Do not accept a number less than 1 for the number of years. Do not accept negative numbers for the monthly rainfall.

11. 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), 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 2 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 1 for the number of days they will multiply.

12. Celsius to Fahrenheit Table

In Programming Challenge 12 of Chapter 3, you were asked to write a program that converts a Celsius temperature to Fahrenheit. Modify that program so that it uses a loop to display a table of the Celsius temperatures 0–20, and the Fahrenheit equivalents.

13. The Greatest and Least of These

Write a program with a loop that lets the user enter a series of integers. The user should enter -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.

14. Student Line Up

A teacher has asked all her students to line up 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 that many names. Once all the names have been read, 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.

15. 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. Do not accept values for state, federal, or FICA withholdings that are greater than the gross pay. If the sum of 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.

16. Savings Account Balance

Write a program that calculates the balance of a savings account at the end of a period of time. It should ask the user for the annual interest rate, the starting balance, and the number of months that have passed since the account was established. A loop should then iterate once for every month, performing the following:

- Ask the user for the amount deposited into the account during the month. (Do not accept negative numbers.) This amount should be added to the balance.
- Ask the user for the amount withdrawn from the account during the month. (Do not accept negative numbers.) This amount should be subtracted from the balance.
- Calculate the monthly interest. The monthly interest rate is the annual interest rate divided by 12. Multiply the monthly interest rate by the balance, and add the result to the balance.

After the last iteration, the program should display the ending balance, the total amount of deposits, the total amount of withdrawals, and the total interest earned.



NOTE: If a negative balance is calculated at any point, a message should be displayed indicating the account has been closed and the loop should terminate.

17. Sales Bar Chart

Write a program that asks the user to enter today's sales for five stores. The program should then display a bar graph comparing each store's sales. Create each bar in the bar graph by displaying a row of asterisks. Each asterisk should represent \$100 of sales.

Here is an example of the program's output:

```
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: 1800 [Enter]
Enter today's sales for store 4: 800 [Enter]
Enter today's sales for store 5: 1900 [Enter]
```

```
SALES BAR CHART
(Each * = $100)
Store 1: *****
Store 2: *****
Store 3: *****
Store 4: *****
Store 5: *****
```

18. 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 1,000 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 1,000 people. The data can be found in the `People.txt` file.

Here is an example of how the chart might begin:

```
PRAIRIEVILLE POPULATION GROWTH
(each * represents 1,000 people)
1900 **
1920 ****
1940 *****
```

19. Budget Analysis

Write a program that asks the user to enter the amount that he or she has budgeted for a month. A loop should then prompt the user to enter each of his or her expenses for the month and keep a running total. When the loop finishes, the program should display the amount that the user is over or under budget.

20. Random Number Guessing Game

Write a program that generates a random number 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.

21. Random Number Guessing Game Enhancement

Enhance the program that you wrote for Programming Challenge 20 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.

22. Square Display

Write a program that asks the user for a positive integer no greater than 15. The program should then display a square on the screen using the character 'X'. The number entered by the user will be the length of each side of the square. For example, if the user enters 5, the program should display the following:

```
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
```

If the user enters 8, the program should display the following:

```
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
```

23. Pattern Displays

Write a program that uses a loop to display Pattern A below, followed by another loop that displays Pattern B.

Pattern A	Pattern B
+	++++++
++	++++++
+++	++++++
++++	++++++
+++++	++++++
++++++	++++
++++++	+++
+++++++	++
++++++	+

24. Using Files—Numeric Processing

If you have downloaded this book's source code from the Computer Science Portal, you will find a file named Random.txt in the Chapter 05 folder. (The Portal can be found at www.pearsonhighered.com/gaddis.) This file contains a long list of random numbers. Copy the file to your system, then write a program that opens the file, reads all the numbers from the file, and calculates the following:

- A) The number of numbers in the file
- B) The sum of all the numbers in the file (a running total)
- C) The average of all the numbers in the file

The program should display the number of numbers found in the file, the sum of the numbers, and the average of the numbers.

25. Using Files—Student Line Up

Modify the Student Line Up program described in Programming Challenge 14 so it gets the names from a file. Names should be read in until there is no more data to read. If you have downloaded this book's source code you will find a file named LineUp.txt in the Chapter 05 folder. You can use this file to test the program.

26. Personal Web Page Generator

Write a program that asks the user for his or her name, then asks the user to enter a sentence that describes himself or herself. Here is an example of the program's screen:

Enter your name: **Julie Taylor**

Describe yourself: **I am a computer science major, a member of the Jazz club, and I hope to work as a mobile app developer after I graduate.**

Once the user has entered the requested input, the program should create an HTML file, containing the input, for a simple webpage. Here is an example of the HTML content, using the sample input previously shown:

```
<html>
<head>
</head>
<body>
    <center>
        <h1>Julie Taylor</h1>
    </center>
    <hr />
    I am a computer science major, a member of the Jazz club,
    and I hope to work as a mobile app developer after I graduate.
    <hr />
</body>
</html>
```

27. Average Steps Taken

A Personal Fitness Tracker is a wearable device that tracks your physical activity, calories burned, heart rate, sleeping patterns, and so on. One common physical activity that most of these devices track is the number of steps you take each day.

If you have downloaded this book's source code, you will find a file named `steps.txt` in the Chapter 05 folder. The `steps.txt` file contains the number of steps a person has taken each day for a year. There are 365 lines in the file, and each line contains the number of steps taken during a day. (The first line is the number of steps taken on January 1, the second line is the number of steps taken on January 2, and so forth.) Write a program that reads the file, then displays the average number of steps taken for each month. (The data is from a year that was not a leap year, so February has 28 days.)

TOPICS

- | | |
|--|---|
| 6.1 Focus on Software Engineering:
Modular Programming | 6.9 Returning a Boolean Value |
| 6.2 Defining and Calling Functions | 6.10 Local and Global Variables |
| 6.3 Function Prototypes | 6.11 Static Local Variables |
| 6.4 Sending Data into a Function | 6.12 Default Arguments |
| 6.5 Passing Data by Value | 6.13 Using Reference Variables as
Parameters |
| 6.6 Focus on Software Engineering: Using
Functions in a Menu-Driven Program | 6.14 Overloading Functions |
| 6.7 The <code>return</code> Statement | 6.15 The <code>exit()</code> Function |
| 6.8 Returning a Value from a Function | 6.16 Stubs and Drivers |

6.1**Focus on Software Engineering:
Modular Programming**

CONCEPT: A program may be broken up into manageable functions.

A function is a collection of statements that performs a specific task. So far, you have experienced functions in two ways: (1) you have created a function named `main` in every program you've written, and (2) you have used library functions such as `pow` and `strcmp`. 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. Instead of writing one long function that contains all of the statements necessary to solve a problem, several small functions that each solve a specific part of the problem can be written. 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 easily solved. Figure 6-1 illustrates this idea by comparing two programs: one that uses a long, complex function containing all of the statements necessary to solve a problem, and another that divides a problem into smaller problems, each of which is handled by a separate function.

Figure 6-1 Modular programming

This program has one long, complex function containing all of the statements necessary to solve a problem.

```
int main()
{
    statement;
    statement;
}
```

In this program, the problem has been divided into smaller problems, each of which is handled by a separate function.

```
int main()
{
    statement;
    statement;
    statement;
}

void function2()
{
    statement;
    statement;
    statement;
}

void function3()
{
    statement;
    statement;
    statement;
}

void function4()
{
    statement;
    statement;
    statement;
}
```

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, 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, 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.

When creating a function, you must write its *definition*. All function definitions have the following parts:

Return type: A function can send a value to the part of the program that executed it. The return type is the data type of the value that is sent from the function.

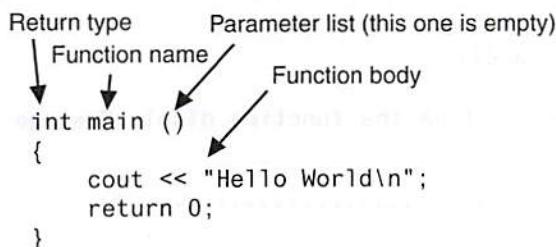
Name: You should give each function a descriptive name. In general, the same rules that apply to variable names also apply to function names.

Parameter list: The program can send data into a function. The parameter list is a list of variables that hold the values being passed to the function.

Body: The body of a function is the set of statements that perform the function's operation. They are enclosed in a set of braces.

Figure 6-2 shows the definition of a simple function with the various parts labeled.

Figure 6-2 Function definition



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 terminate. These are called *void functions*. The `displayMessage` function, which follows, is an example.

```
void displayMessage() {    cout << "Hello from the function displayMessage.\n"; }
```

The function's name is `displayMessage`. This name gives an indication of what the function does: It displays a message. You should always give functions names that reflect their purpose. Notice the function's return type is `void`. This means the function does not return a value to the part of the program that executed it. Also notice the function has no `return` statement. It simply displays a message on the screen and exits.

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 // Definition of function displayMessage *
7 // This function displays a greeting. *
8 //*****
9
10 void displayMessage()
11 {
12     cout << "Hello from the function displayMessage.\n";
13 }
14
15 //*****
16 // Function main *
17 //*****
18
19 int main()
20 {
21     cout << "Hello from main.\n";
22     displayMessage();
23     cout << "Back in function main again.\n";
24     return 0;
25 }
```

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 22:

```
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 is not 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. The return type is not listed in the function call, 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.

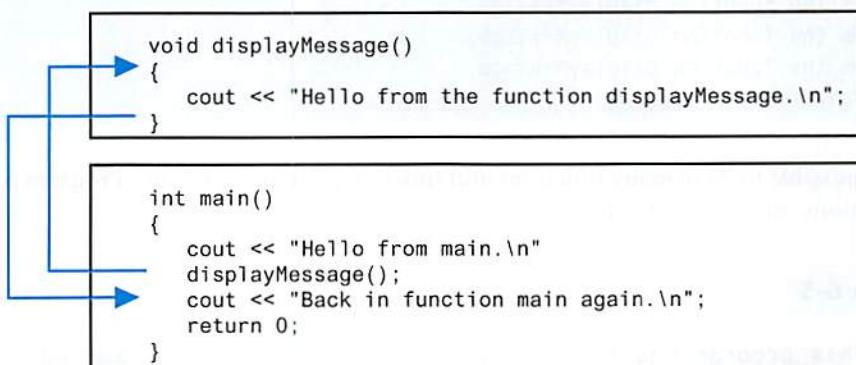
Even though the program starts executing at `main`, the function `displayMessage` is defined first. This is because the compiler must know the function's return type, the number of parameters, and the type of each parameter before the function is called. One way to ensure the compiler will know this information is to place the function definition before all calls to that function. (Later, you will see an alternative, 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 A function call



Function call statements may be used in control structures like 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 a loop.
2 #include <iostream>
3 using namespace std;
4
5 //*****
6 // Definition of function displayMessage *
7 // This function displays a greeting. *
8 //*****
9
10 void displayMessage()
11 {
12     cout << "Hello from the function displayMessage.\n";
13 }
14
15 //*****
16 // Function main *
17 //*****
18
19 int main()
20 {
21     cout << "Hello from main.\n";
22     for (int count = 0; count < 5; count++)
23         displayMessage(); // Call displayMessage
24     cout << "Back in function main again.\n";
25     return 0;
26 }
```

Program Output

Hello from main.
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 // Definition of function first      *
7 // This function displays a message.  *
8 //*****
9
10 void first()
11 {
12     cout << "I am now inside the function first.\n";
13 }
14
15 //*****
16 // Definition of function second      *
17 // This function displays a message.  *
18 //*****
19
20 void second()
21 {
22     cout << "I am now inside the function second.\n";
23 }
24
25 //*****
26 // Function main                  *
27 //*****
28
29 int main()
30 {
31     cout << "I am starting in function main.\n";
32     first();    // Call function first
33     second();   // Call function second
34     cout << "Back in function main again. \n";
35     return 0;
36 }
```

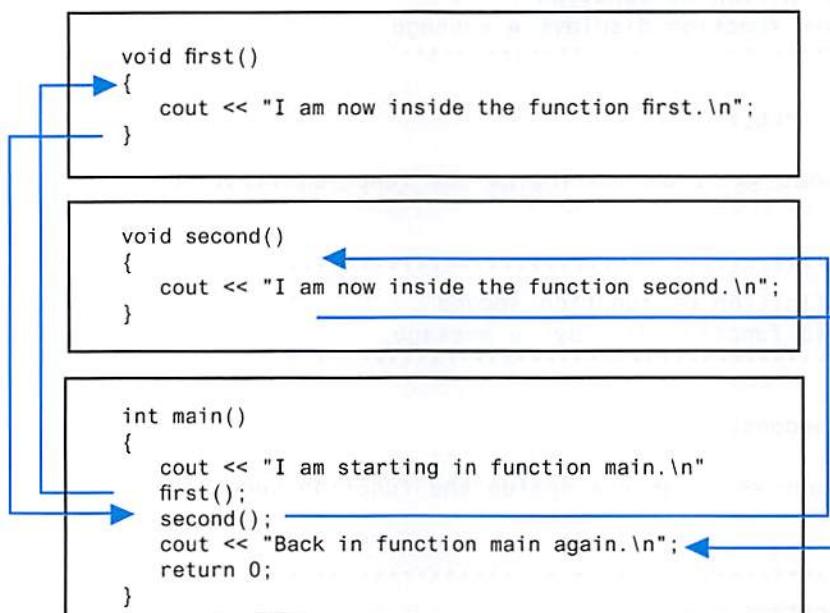
Program Output

```
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 32 and 33 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 then back to `main` when the function is finished. Figure 6-4 illustrates the paths taken by the program.

Figure 6-4 Paths taken by the program

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 // Definition of function deeper      *
7 // This function displays a message.  *
8 //*****
9
10 void deeper()
11 {
12     cout << "I am now inside the function deeper.\n";
13 }
14
15 //*****
16 // Definition of function deep       *
17 // This function displays a message.  *
18 //*****
19
  
```

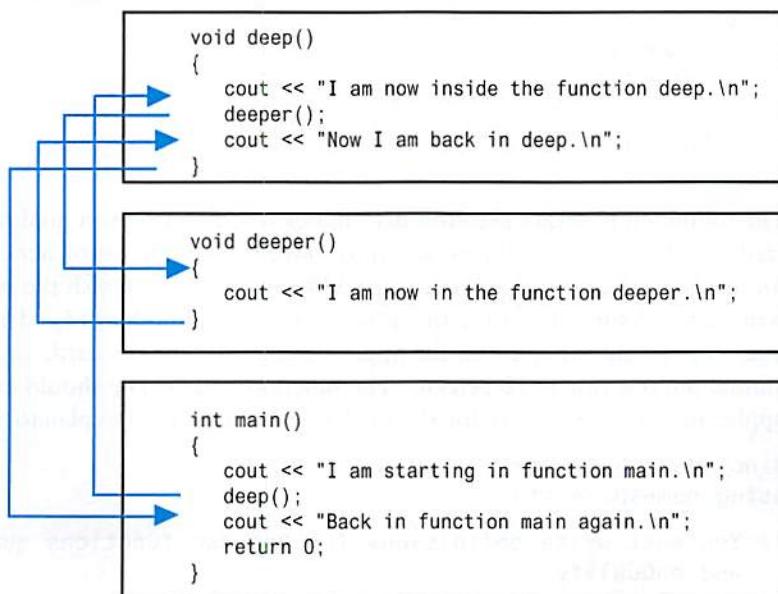
```
20 void deep()
21 {
22     cout << "I am now inside the function deep.\n";
23     deeper();      // Call function deeper
24     cout << "Now I am back in deep.\n";
25 }
26
27 //*****
28 // Function main
29 //*****
30
31 int main()
32 {
33     cout << "I am starting in function main.\n";
34     deep();        // Call function deep
35     cout << "Back in function main again.\n";
36     return 0;
37 }
```

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 Layered function calls





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 2 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 the function's return type, the number of parameters it uses, and the type of each parameter. (You will learn how to use parameters in the next section.)

One way of ensuring that the compiler has this information is to place the function definition before all calls to that function. This was the approach taken in Programs 6-1, 6-2, 6-3, and 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();
```

The prototype looks similar to the function header, except there is a semicolon at the end. The statement above tells the compiler that the function `displayMessage` has a `void` return type (it doesn't return a value) and uses no parameters.



NOTE: Function prototypes are also known as *function declarations*.



WARNING! You must place either 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 a function prototype has been placed after the `using namespace std` statement.

Program 6-5

```

1 // This program has three functions: main, first, and second.
2 #include <iostream>
3 using namespace std;
4
5 // Function Prototypes
6 void first();
7 void second();
8
9 int main()
10 {
11     cout << "I am starting in function main.\n";
12     first();    // Call function first
13     second();   // Call function second
14     cout << "Back in function main again.\n";
15     return 0;
16 }
17
18 //*****
19 // Definition of function first.      *
20 // This function displays a message.  *
21 //*****
22
23 void first()
24 {
25     cout << "I am now inside the function first.\n";
26 }
27
28 //*****
29 // Definition of function second.      *
30 // This function displays a message.  *
31 //*****
32
33 void second()
34 {
35     cout << "I am now inside the function second.\n";
36 }
```

Program Output

(The program's 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 12 and 13 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`.



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.

Values 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 and two arguments, `2.0` and `4.0`, are passed to it:

```
result = pow(2.0, 4.0);
```

By using *parameters*, you can design your own functions that accept data this way. A parameter is a special variable that holds a value being passed into a function. 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 using 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 argument* and *formal argument*. 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);
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
```

(program continues)

Program 6-6

(continued)

```

16 //*****
17 // Definition of function displayValue. *
18 // It uses an integer parameter whose value is displayed.*  

19 //*****
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.

First, notice the function prototype for `displayValue` in line 6:

```
void displayValue(int);
```

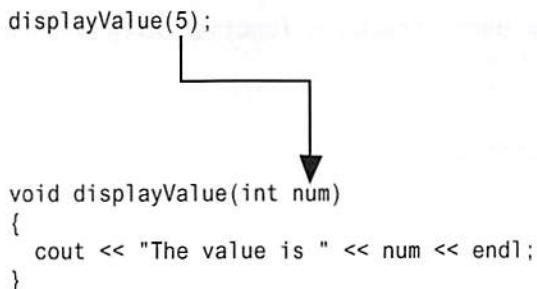
It is not necessary to list the name of the parameter variable inside the parentheses. Only its data type is required. The function prototype shown above could optionally have been written as

```
void displayValue(int num);
```

However, the compiler ignores the name of the parameter variable in the function prototype.

In `main`, the `displayValue` function is called 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 An argument passed to a function



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 their corresponding arguments. Program 6-7 shows the function `displayValue` being called several times with a different argument being 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);
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 // Definition of function displayValue.
21 // It uses an integer parameter whose value is displayed.
22 //*****
23
24 void displayValue(int num)
25 {
26     cout << "The value is " << num << endl;
27 }
```

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.
```



WARNING! When passing a variable as an argument, simply write the variable name inside the parentheses of the function call. Do not write the data type of the argument variable in the function call. For example, the following function call will cause an error:

```
displayValue(int x); // Error!
```

The function call should appear as

```
displayValue(x); // Correct
```

Each time the function is called in Program 6-7, 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);
```

If you pass an argument whose type is not the same as the parameter's type, the argument will be promoted or demoted automatically. For instance, the argument in the following function call would be truncated, causing the value 4 to be passed to num:

```
displayValue(4.7);
```

Often, it's useful to pass several arguments into a function. Program 6-8 shows the definition of a function with three parameters.

Program 6-8

```

1 // This program demonstrates a function with three parameters.
2 #include <iostream>
3 using namespace std;
4
5 // Function Prototype
6 void showSum(int, int, int);
7
8 int main()
9 {
10     int value1, value2, value3;
11
12     // Get three integers.
13     cout << "Enter three integers and I will display ";
14     cout << "their sum: ";
15     cin >> value1 >> value2 >> value3;
16
17     // Call showSum passing three arguments.
18     showSum(value1, value2, value3);
19     return 0;
20 }
21
22 //*****
23 // Definition of function showSum.
24 // It uses three integer parameters. Their sum is displayed.* 
25 //*****
26
27 void showSum(int num1, int num2, int num3)
28 {
29     cout << (num1 + num2 + num3) << endl;
30 }
```

Program Output with Example Input Shown in Bold

Enter three integers and I will display their sum: **4 8 7**

19

In the function header for showSum, the parameter list contains three variable definitions separated by commas:

```
void showSum(int num1, int num2, int num3)
```

WARNING! Each parameter variable in a parameter list must have a data type listed before its name. For example, a compiler error would occur if the parameter list for the `showSum` function were defined as shown in the following header:

```
void showSum(int num1, num2, num3) // Error!
```

A data type for all three of the parameter variables must be listed, as shown here:

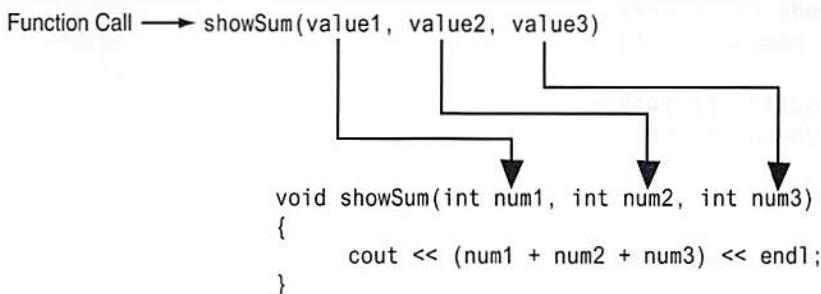
```
void showSum(int num1, int num2, int num3) // Correct
```

In the function call in line 18, the variables `value1`, `value2`, and `value3` are passed as arguments:

```
showSum(value1, value2, value3);
```

When a function with multiple parameters is called, the arguments are passed to the parameters in order. This is illustrated in Figure 6-7.

Figure 6-7 Multiple arguments



The following function call will cause 5 to be passed into the `num1` parameter, 10 to be passed into `num2`, and 15 to be passed into `num3`:

```
showSum(5, 10, 15);
```

However, the following function call will cause 15 to be passed into the `num1` parameter, 5 to be passed into `num2`, and 10 to be passed into `num3`:

```
showSum(15, 5, 10);
```



NOTE: The function prototype must list the data type of each parameter.



NOTE: Like all variables, parameters have a scope. The scope of a parameter is limited to the body of the function that uses it.

6.5

Passing Data by Value

CONCEPT: When an argument is passed into a parameter, only a copy of the argument's value is passed. Changes to the parameter do not affect the original argument.

As you've seen in this chapter, parameters are special-purpose variables that are defined inside the parentheses of a function definition. They are separate and distinct from the arguments that are listed inside the parentheses of a function call. The values that are stored in the parameter variables are copies of the arguments. Normally, when a parameter's value is changed inside a function, it has no effect on the original argument. Program 6-9 demonstrates this concept.

Program 6-9

```
1 // This program demonstrates that changes to a function parameter
2 // have no effect on the original argument.
3 #include <iostream>
4 using namespace std;
5
6 // Function Prototype
7 void changeMe(int);
8
9 int main()
10 {
11     int number = 12;
12
13     // Display the value in number.
14     cout << "number is " << number << endl;
15
16     // Call changeMe, passing the value in number
17     // as an argument.
18     changeMe(number);
19
20     // Display the value in number again.
21     cout << "Now back in main again, the value of ";
22     cout << "number is " << number << endl;
23     return 0;
24 }
25
26 //*****
27 // Definition of function changeMe.
28 // This function changes the value of the parameter myValue. *
29 //*****
```

```

30
31 void changeMe(int myValue)
32 {
33     // Change the value of myValue to 0.
34     myValue = 0;
35
36     // Display the value in myValue.
37     cout << "Now the value is " << myValue << endl;
38 }
```

Program Output

```

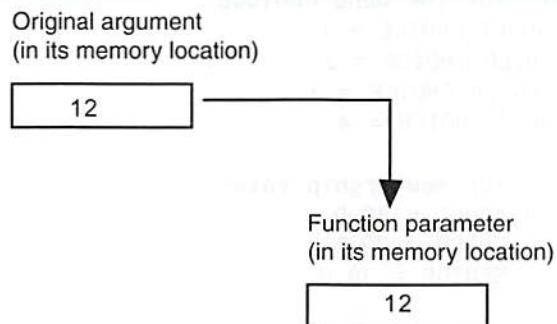
number is 12
Now the value is 0
Now back in main again, the value of number is 12
```

Even though the parameter variable `myValue` is changed in the `changeMe` function, the argument `number` is not modified. The `myValue` variable contains only a copy of the `number` variable.

The `changeMe` function does not have access to the original argument. When only a copy of an argument is passed to a function, it is said to be *passed by value*. This is because the function receives a copy of the argument's value and 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 The argument and the parameter are separate



NOTE: Later in this chapter, you will learn ways to give a function access to its original arguments.

6.6

Focus on Software Engineering: 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 6-10 shows the program redesigned as a modular program. A *modular* program is broken up into functions that perform specific tasks.

Program 6-10

```
1 // This is a menu-driven program that makes a function call
2 // for each selection the user makes.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 // Function prototypes
8 void showMenu();
9 void showFees(double, int);
10
11 int main()
12 {
13     int choice;      // To hold a menu choice
14     int months;      // To hold a number of months
15
16     // Constants for the menu choices
17     const int ADULT_CHOICE = 1,
18             CHILD_CHOICE = 2,
19             SENIOR_CHOICE = 3,
20             QUIT_CHOICE = 4;
21
22     // Constants for membership rates
23     const double ADULT = 40.0,
24                 CHILD = 20.0;
25                 SENIOR = 30.0,
26
27     // Set up numeric output formatting.
28     cout << fixed << showpoint << setprecision(2);
29
30     do
31     {
32         // Display the menu and get the user's choice.
33         showMenu();
34         cin >> choice;
35
36         // Validate the menu selection.
37         while (choice < ADULT_CHOICE || choice > QUIT_CHOICE)
38     {
```

```
39         cout << "Please enter a valid menu choice: ";
40         cin >> choice;
41     }
42
43     // If the user does not want to quit, proceed.
44     if (choice != QUIT_CHOICE)
45     {
46         // Get the number of months.
47         cout << "For how many months? ";
48         cin >> months;
49
50         // Display the membership fees.
51         switch (choice)
52         {
53             case ADULT_CHOICE:
54                 showFees(ADULT, months);
55                 break;
56             case CHILD_CHOICE:
57                 showFees(CHILD, months);
58                 break;
59             case SENIOR_CHOICE:
60                 showFees(SENIOR, months);
61         }
62     }
63 } while (choice != QUIT_CHOICE);
64 return 0;
65 }
66
67 //*****
68 // Definition of function showMenu which displays the menu.
69 //*****
70
71 void showMenu()
72 {
73     cout << "\n\t\tHealth Club Membership Menu\n\n"
74         << "1. Standard Adult Membership\n"
75         << "2. Child Membership\n"
76         << "3. Senior Citizen Membership\n"
77         << "4. Quit the Program\n\n"
78         << "Enter your choice: ";
79 }
80
81 //*****
82 // Definition of function showFees. The memberRate parameter holds *
83 // the monthly membership rate and the months parameter holds the *
84 // number of months. The function displays the total charges. *
85 //*****
86
87 void showFees(double memberRate, int months)
88 {
89     cout << "The total charges are $"
90         << (memberRate * months) << endl;
91 }
```

(program output continues)

Program 6-10

(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**

For how many months? **12**

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**

Let's take a closer look at this program. First notice the `showMenu` function in lines 71 through 79. This function displays the menu and is called from the `main` function in line 33.

The `showFees` function appears in lines 87 through 91. Its purpose is to display the total fees for a membership lasting a specified number of months. The function accepts two arguments: the monthly membership fee (a `double`) and the number of months of membership (an `int`). The function uses these values to calculate and display the total charges. For example, if we wanted the function to display the fees for an adult membership lasting 6 months, we would pass the `ADULT` constant as the first argument, and 6 as the second argument.

The `showFees` function is called from three different locations in the `switch` statement, which is in the `main` function. The first location is line 54. This statement is executed when the user has selected item 1, standard adult membership, from the menu. The `showFees` function is called with the `ADULT` constant and the `months` variable passed as arguments. The second location is line 57. This statement is executed when the user has selected item 2, child membership, from the menu. The `showFees` function is called in this line with the `CHILD` constant and the `months` variable passed as arguments. The third location is line 60. This statement is executed when the user has selected item 3, senior citizen membership, from the menu. The `showFees` function is called with the `SENIOR` constant and the `months` variable passed as arguments. Each time the `showFees` function is called, it displays the total membership fees for the specified type of membership, for the specified number of months.



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 ten. (*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); // 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 << "\t" << (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>
using namespace std;

void showDollars(double); // 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.

```

6.7

The return Statement

CONCEPT: The `return` statement causes a function to end immediately.

When the last statement in a `void` function has finished executing, the function terminates and the program returns to the statement following the function call. It's possible, however, to force a function to return before the last statement has been executed. When the `return` statement is encountered, the function immediately terminates and control of the program returns to the statement that called the function. This is demonstrated in Program 6-11. The function `divide` shows the quotient of `arg1` divided by `arg2`. If `arg2` is set to zero, the function returns.

Program 6-11

```

1 // This program uses a function to perform division. If division
2 // by zero is detected, the function returns.
3 #include <iostream>
4 using namespace std;
5
6 // Function prototype.
7 void divide(double, double);
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 // Definition of function divide.
22 // Uses two parameters: arg1 and arg2. The function divides arg1
23 // by arg2 and shows the result. If arg2 is zero, however, the
24 // function returns.
25 //*****
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 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. In line 16, the `divide` function was called, passing 12 into the `arg1` parameter and 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 executes. 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.

6.8

Returning a Value from a Function

CONCEPT: A function may send a value back to the part of the program that called the function.

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 appropriately known as *value-returning functions*.

The `pow` function, which you have already seen, is an example of a value-returning function. Here is an example:

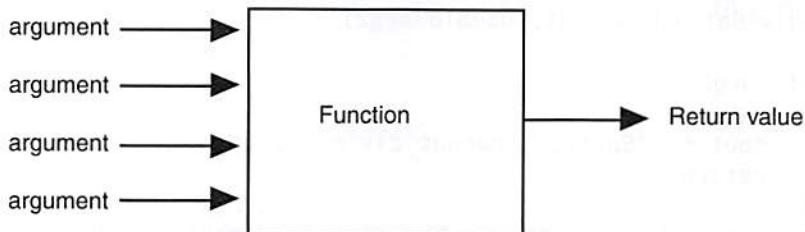
```
double x;
x = pow(4.0, 2.0);
```



The second line in 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 may be passed into a function, only one value may 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 A function returns one value



NOTE: It is possible to return multiple values from a function, but they must be “packaged” in such a way that they are treated as a single value. This will be a topic in Chapter 11.

Defining a Value-Returning Function

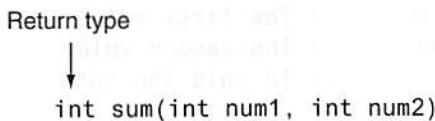
When you are writing a value-returning function, you must decide what type of value the function will return. This is because you must specify the data type of the return value in the function header and in the function prototype. Recall that a `void` function, which does not return a value, uses the key word `void` as its return type in the function header.

A value-returning function will use `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 the return type is `int`, as illustrated in Figure 6-10.

Figure 6-10 Function return type



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, a variable, `result`, is defined. Variables that are defined inside a function are called *local variables*. After the variable definition, 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 and returned the expression `num1 + num2`, as shown in the following code:

```
int sum(int num1, int num2)
{
    return num1 + num2;
}
```

When writing the prototype for a value-returning function, follow the same conventions we have covered earlier. Here is the prototype for the `sum` function:

```
int sum(int, int);
```

Calling a Value-Returning Function

Program 6-12 shows an example of how to call the `sum` function.

Program 6-12

```

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, int);
7
8 int main()
9 {
10     int value1 = 20,      // The first value
11         value2 = 40,      // The second value
12         total;          // To hold the 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 // Definition of function sum. This function returns *
27 // the sum of its two parameters. *
28 //*****
29
30 int sum(int num1, int num2)
31 {
32     return num1 + num2;
33 }
```

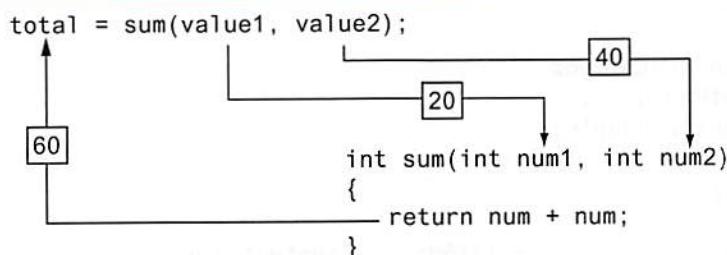
Program Output

The sum of 20 and 40 is 60

Here is the statement in line 17 that 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 A function's arguments and return value

When you call a value-returning function, you usually want to do something meaningful with the value it returns. Program 6-12 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. For example, the following code shows a mathematical 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 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 6-13, 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 6-13

```

1 // This program demonstrates two value-returning functions.
2 // The square function is called in a mathematical statement.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
  
```

(program continues)

Program 6-13

(continued)

```

6
7 //Function prototypes
8 double getRadius();
9 double square(double);
10
11 int main()
12 {
13     const double PI = 3.14159; // Constant for pi
14     double radius;           // To hold the circle's radius
15     double area;            // To hold the circle's area
16
17     // Set the numeric output formatting.
18     cout << fixed << showpoint << setprecision(2);
19
20     // Get the radius of the circle.
21     cout << "This program calculates the area of ";
22     cout << "a circle.\n";
23     radius = getRadius();
24
25     // Calculate the area of the circle.
26     area = PI * square(radius);
27
28     // Display the area.
29     cout << "The area is " << area << endl;
30     return 0;
31 }
32
33 //*****
34 // Definition of function getRadius.
35 // This function asks the user to enter the radius of *
36 // the circle and then returns that number as a double.* *
37 //*****
38
39 double getRadius()
40 {
41     double rad;
42
43     cout << "Enter the radius of the circle: ";
44     cin >> rad;
45     return rad;
46 }
47
48 //*****
49 // Definition of function square.
50 // This function accepts a double argument and returns *
51 // the square of the argument as a double. *
52 //*****
53
54 double square(double number)

```

```

55  {
56      return number * number;
57  }

```

Program Output with Example Input Shown in Bold

This program calculates the area of a circle.

Enter the radius of the circle: **10**

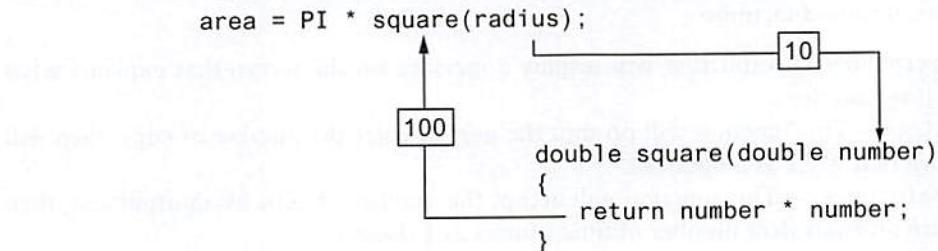
The area is 314.16

First, look at the `getRadius` function defined in lines 39 through 46. The purpose of the function is to prompt the user to enter the radius of a circle. In line 41, the function defines a local variable, `rad`. Lines 43 and 44 prompt the user to enter the circle's radius, which is stored in the `rad` variable. In line 45, the value of the `rad` value is returned. The `getRadius` function is called in the `main` function, in line 23. The value that is returned from the function is assigned to the `radius` variable.

Next, look at the `square` function, which is defined in lines 54 through 57. When the function is called, a `double` argument is passed to it. The function stores the argument in the `number` parameter. The `return` statement in line 56 returns the value of the expression `number * number`, which is the square of the `number` parameter. The `square` function is called in the `main` function, in line 26, with the value of `radius` passed as an argument. The function will return the square of the `radius` variable, and that value will be used in the mathematical expression.

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 the mathematical expression in line 26. The value 100 will then be used in the mathematical expression.

Figure 6-12 The `square` function



Functions can return values of any type. Both the `getRadius` and `square` functions in Program 6-13 return a `double`. The `sum` function you saw in Program 6-12 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.

In the Spotlight: Using Functions



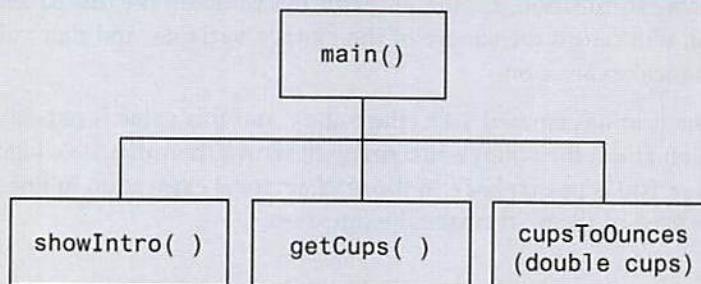
Your friend Michael runs a catering company. Some of the ingredients that his recipes require are measured in cups. When he goes to the grocery store to buy those ingredients, however, they are sold only by the fluid ounce. He has asked you to write a simple program that converts cups to fluid ounces.

You design the following algorithm:

1. *Display an introductory screen that explains what the program does.*
2. *Get the number of cups.*
3. *Convert the number of cups to fluid ounces and display the result.*

This algorithm lists the top level of tasks that the program needs to perform and becomes the basis of the program's `main` function. The hierarchy chart shown in Figure 6-13 shows how the program will be broken down into functions.

Figure 6-13 Hierarchy chart for the program



As shown in the hierarchy chart, the `main` function will call three other functions. Here are summaries of those functions:

- `showIntro`—This function will display a message on the screen that explains what the program does.
- `getCups`—This function will prompt the user to enter the number of cups, then will return that value as a `double`.
- `cupsToOunces`—This function will accept the number of cups as an argument, then return an equivalent number of fluid ounces as a `double`.

Program 6-14 shows the code for the program.

Program 6-14

```

1 // This program converts cups to fluid ounces.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
  
```

```
5 // Function prototypes
6 void showIntro();
7 double getCups();
8 double cupsToOunces(double);
9
10 int main()
11 {
12     // Variables for the cups and ounces.
13     double cups, ounces;
14
15     // Set up numeric output formatting.
16     cout << fixed << showpoint << setprecision(1);
17
18     // Display an intro screen.
19     showIntro();
20
21     // Get the number of cups.
22     cups = getCups();
23
24     // Convert cups to fluid ounces.
25     ounces = cupsToOunces(cups);
26
27     // Display the number of ounces.
28     cout << cups << " cups equals "
29         << ounces << " ounces.\n";
30
31     return 0;
32 }
33
34 //*****
35 // The showIntro function displays an *
36 // introductory screen. *
37 //***** *
38
39
40 void showIntro()
41 {
42     cout << "This program converts measurements\n"
43         << "in cups to fluid ounces. For your\n"
44         << "reference the formula is:\n"
45         << " 1 cup = 8 fluid ounces\n\n";
46 }
47
48 //*****
49 // The getCups function prompts the user *
50 // to enter the number of cups and then *
51 // returns that value as a double. *
52 //***** *
53
54 double getCups()
```

(program continues)

Program 6-14 (continued)

```
55  {
56      double numCups;
57
58      cout << "Enter the number of cups: ";
59      cin >> numCups;
60      return numCups;
61  }
62
63 //*****
64 // The cupsToOunces function accepts a *
65 // number of cups as an argument and *
66 // returns the equivalent number of fluid *
67 // ounces as a double.
68 //*****
69
70 double cupsToOunces(double numCups)
71 {
72     return numCups * 8.0;
73 }
```

Program Output with Example Input Shown in Bold

This program converts measurements in cups to fluid ounces. For your reference the formula is:

1 cup = 8 fluid ounces

Enter the number of cups: **2**
2.0 cups equals 16.0 ounces.

6.9

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 exists. Such a function would return a bool value. For example, the following function 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 uses 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";
```

When this code executes, the message “The value is within range.” will be displayed.

Program 6-15 shows another example. This program has a function named `isEven` which returns `true` if its argument is an even number. Otherwise, the function returns `false`.

Program 6-15

```
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;
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
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 // Definition of function isEven. This function accepts an
27 // integer argument and tests it to be even or odd. The function *
28 // returns true if the argument is even or false if the argument *
29 // is odd. The return value is a bool.
30 //*****
31
32 bool isEven(int number)
33 {
34     bool status;
35
36     if (number % 2 == 0)
37         status = true; // The number is even if there is no remainder.
38     else
```

(program continues)

Program 6-15

(continued)

```

39     status = false; // Otherwise, the number is odd.
40     return status;
41 }
```

Program Output with Example Input Shown in Bold

Enter an integer and I will tell you if it is even or odd: **5** 5 is odd.

The `isEven` function is called in line 18, in the following statement:

```
if (isEven(val))
```

When the `if` statement executes, `isEven` is called with `val` as its argument. If `val` is even, `isEven` returns `true`; otherwise, it returns `false`.

**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`.

6.10**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 6-16 shows that because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.

Program 6-16

```

1 // This program shows that variables defined in a function
2 // are hidden from other functions.
3 #include <iostream>
4 using namespace std;
```

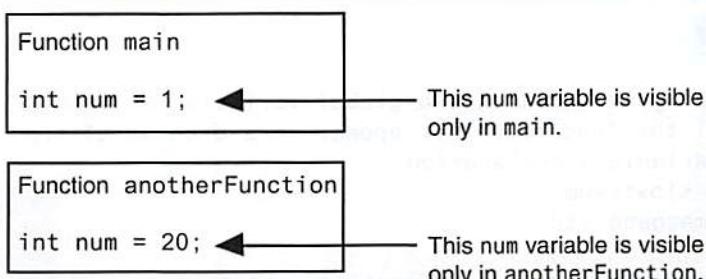
```
5 void anotherFunction(); // Function prototype
6
7 int main()
8 {
9     int num = 1; // Local variable
10
11     cout << "In main, num is " << num << endl;
12     anotherFunction();
13     cout << "Back in main, num is " << num << endl;
14     return 0;
15 }
16
17 //*****
18 // Definition of anotherFunction
19 // It has a local variable, num, whose initial value
20 // is displayed.
21 //*****
22
23
24 void anotherFunction()
25 {
26     int num = 20; // Local variable
27
28     cout << "In anotherFunction, num is " << num << endl;
29 }
```

Program Output

```
In main, num is 1
In anotherFunction, num is 20
Back in main, num is 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 6-14 illustrates the closed nature of the two functions. The boxes represent the scope of the variables.

Figure 6-14 Visibility of local variables



Local Variable Lifetime

A function's local variables exist only while the function is executing. This is known as the *lifetime* of a local variable. When the function begins, its local variables and its parameter variables are created in memory, and when the function ends, the local variables and parameter variables are destroyed. This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.

Initializing Local Variables with Parameter Values

It is possible to use a parameter variable to initialize a local variable. Sometimes this simplifies the code in a function. For example, recall the first version of the `sum` function we discussed earlier:

```
int sum(int num1, int num2)
{
    int result;
    result = num1 + num2;
    return result;
}
```

In the body of the function, the `result` variable is defined and then a separate assignment statement assigns `num1 + num2` to `result`. We can combine these statements into one, as shown in the following modified version of the function.

```
int sum(int num1, int num2)
{
    int result = num1 + num2;
    return result;
}
```

Because the scope of a parameter variable is the entire function in which it is declared, we can use parameter variables to initialize local variables.

Global Variables

A global variable is any variable defined outside all the functions in a program. The scope of a global variable is the portion of the program from the variable definition to the end. This means a global variable can be accessed by all functions that are defined after the global variable is defined. Program 6-17 shows two functions, `main` and `anotherFunction`, that access the same global variable, `num`.

Program 6-17

```
1 // This program shows that a global variable is visible
2 // to all the functions that appear in a program after
3 // the variable's declaration.
4 #include <iostream>
5 using namespace std;
6
7 void anotherFunction(); // Function prototype
8 int num = 2;           // Global variable
9
10 int main()
```

```
11 {
12     cout << "In main, num is " << num << endl;
13     anotherFunction();
14     cout << "Back in main, num is " << num << endl;
15     return 0;
16 }
17
18 //*****
19 // Definition of anotherFunction
20 // This function changes the value of the
21 // global variable num.
22 //*****
23
24 void anotherFunction()
25 {
26     cout << "In anotherFunction, num is " << num << endl;
27     num = 50;
28     cout << "But, it is now changed to " << num << endl;
29 }
```

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 6-17, 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.

Unless you explicitly initialize numeric global variables, they are automatically initialized to zero. Global character variables are initialized to NULL.* The variable globalNum in Program 6-18 is never set to any value by a statement, but because it is global, it is automatically set to zero.

Program 6-18

```
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
```

*The NULL character is stored as ASCII code 0.

Now that you've had a basic introduction to global variables, I must warn you to 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 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. If you are 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 the global variable.

Because of this, you should not use global variables for the conventional purposes of storing, manipulating, and retrieving data. In most cases, you should declare variables locally and pass them as arguments to the functions that need to access them.

Global Constants

Although you should try to avoid the use of global variables, it is generally permissible to use global constants in a program. A *global constant* is a named constant that is available to every function in a program. Because a global constant's value cannot be changed during the program's execution, you do not have to worry about the potential hazards that are associated with the use of global variables.

Global constants are typically used to represent unchanging values that are needed throughout a program. For example, suppose a banking program uses a named constant to represent an interest rate. If the interest rate is used in several functions, it is easier to create a global constant, rather than a local named constant in each function. This also simplifies maintenance. If the interest rate changes, only the declaration of the global constant has to be changed, instead of several local declarations.

Program 6-19 shows an example of how global constants might be used. The program calculates an employee's gross pay, including overtime. 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 6-19

```
1 // This program calculates gross pay.  
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
15 int main()
16 {
17     double hours,           // Hours worked
18         basePay,          // Base pay
19         overtime = 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         overtime = getOvertimePay(hours);
32
33     // Calculate the total pay.
34     totalPay = basePay + overtime;
35
36     // Set up numeric output formatting.
37     cout << setprecision(2) << fixed << showpoint;
38
39     // Display the pay.
40     cout << "Base pay: $" << basePay << endl
41         << "Overtime pay $" << overtime << endl
42         << "Total pay $" << totalPay << endl;
43     return 0;
44 }
45
46 //*****
47 // The getBasePay function accepts the number of *
48 // hours worked as an argument and returns the   *
49 // employee's pay for non-overtime hours.        *
50 //*****
51
52 double getBasePay(double hoursWorked)
53 {
54     double basePay; // To hold base pay
55 }
```

(program continues)

Program 6-19

(continued)

```

56     // Determine base pay.
57     if (hoursWorked > BASE_HOURS)
58         basePay = BASE_HOURS * PAY_RATE;
59     else
60         basePay = hoursWorked * PAY_RATE;
61
62     return basePay;
63 }
64
65 //*****
66 // The getOvertimePay function accepts the number *
67 // of hours worked as an argument and returns the *
68 // employee's overtime pay.
69 //*****
70
71 double getOvertimePay(double hoursWorked)
72 {
73     double overtimePay; // To hold overtime pay
74
75     // Determine overtime pay.
76     if (hoursWorked > BASE_HOURS)
77     {
78         overtimePay = (hoursWorked - BASE_HOURS) *
79                         PAY_RATE * OT_MULTIPLIER;
80     }
81     else
82         overtimePay = 0.0;
83
84     return overtimePay;
85 }
```

Program Output with Example Input Shown in BoldHow many hours did you work? **48**

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, 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 57 and 58 and accesses the PAY_RATE constant in lines 58 and 60. The getOvertimePay function accesses the BASE_HOURS constant in lines 76 and 78, the PAY_RATE constant in line 79, and the OT_MULTIPLIER constant in line 79.

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 local variable or a parameter variable with the same name as a global variable, or a global constant. When you do, the name of the local or parameter variable *shadows* the name of the global variable or global constant. This means that the global variable or constant's name is hidden by the name of the local or parameter variable. For example, look at Program 6-20. This program has a global constant named `BIRDS`, set to 500. The `california` function has a local constant named `BIRDS`, set to 10000.

Program 6-20

```
1 // This program demonstrates how a local variable
2 // can shadow the name of a global constant.
3 #include <iostream>
4 using namespace std;
5
6 // Global constant.
7 const int BIRDS = 500;
8
9 // Function prototype
10 void california();
11
12 int main()
13 {
14     cout << "In main there are " << BIRDS
15         << " birds.\n";
16     california();
17     return 0;
18 }
19
20 //*****
21 // california function
22 //*****
23
24 void california()
25 {
26     const int BIRDS = 10000;
27     cout << "In california there are " << BIRDS
28         << " birds.\n";
29 }
```

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 lines 14 and 15 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 lines 27 and 28 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 the local variables are destroyed when the function terminates, and are then re-created when the function starts again. This is shown in Program 6-21.

Program 6-21

```

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 // Function prototype
7 void showLocal();
8
9 int main()
10 {
11     showLocal();
12     showLocal();
13     return 0;
14 }
15
16 //*****
17 // Definition of function showLocal.
18 // The initial value of localNum, which is 5, is displayed. *
19 // The value of localNum is then changed to 99 before the   *
20 // function returns.                                     *
21 //*****
22
23 void showLocal()
24 {
25     int localNum = 5; // Local variable
26
27     cout << "localNum is " << localNum << endl;
28     localNum = 99;
29 }
```

Program Output

```
localNum is 5
localNum is 5
```

Even though in line 28, the last statement in the `showLocal` function stores 99 in `localNum`, the variable is destroyed when the function returns. The next time the function is called, `localNum` is re-created and initialized to 5 again.

Sometimes 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 lifetime of the program, even though their scope is only the function in which they are defined. Program 6-22 demonstrates some characteristics of **static** local variables:

Program 6-22

```
1 // This program uses a static local variable.
2 #include <iostream>
3 using namespace std;
4
5 void showStatic(); // Function prototype
6
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 // Definition of function showStatic.
17 // statNum is a static local variable. Its value is displayed *
18 // and then incremented just before the function returns. *
19 //*****
20
21 void showStatic()
22 {
23     static int statNum;
24
25     cout << "statNum is " << statNum << endl;
26     statNum++;
27 }
```

Program Output

```
statNum is 0
statNum is 1
statNum is 2
statNum is 3
statNum is 4
```

In line 26 of Program 6-22, **statNum** is incremented in the **showStatic** function, and it retains its value between each function call. Notice even though **statNum** is not explicitly initialized, it starts at zero. Like global variables, all **static** local variables are initialized to zero by default. (Of course, you can provide your own initialization value, if necessary.)

If you do provide an initialization value for a **static** local variable, the initialization only occurs once. This is because initialization normally happens when the variable is created, and **static** local variables are only created once during the running of a program. Program 6-23, which is a slight modification of Program 6-22, illustrates this point.

Program 6-23

```

1 // This program shows that a static local variable is only
2 // initialized once.
3 #include <iostream>
4 using namespace std;
5
6 void showStatic(); // Function prototype
7
8 int main()
9 {
10     // Call the showStatic function five times.
11     for (int count = 0; count < 5; count++)
12         showStatic();
13     return 0;
14 }
15
16 //***** Definition of function showStatic. *****
17 // statNum is a static local variable. Its value is displayed *
18 // and then incremented just before the function returns. *
19 //***** *
20 //*****
21
22 void showStatic()
23 {
24     static int statNum = 5;
25
26     cout << "statNum is " << statNum << endl;
27     statNum++;
28 }
```

Program Output

```

statNum is 5
statNum is 6
statNum is 7
statNum is 8
statNum is 9
```

Even though the statement that defines `statNum` in line 24 initializes it to 5, the initialization does not happen each time the function is called. If it did, the variable would not be able to retain its value between function calls.

**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 = 20.0, double = 10.0);
```

Default arguments are literal values or constants with an = operator in front of them, appearing after the data types listed in a function prototype. Since parameter names are optional in function prototypes, the example prototype could also be declared as

```
void showArea(double length = 20.0, double width = 10.0);
```

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. The parameter `length` will take the value 20.0 and `width` will take 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 call below, 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 function call below, arguments are supplied for both parameters:

```
showArea(12.0, 5.5);
```

The output of the function call above will be

```
The area is 66
```



NOTE: 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;
}
```



WARNING! A function's default arguments should be assigned in the earliest occurrence of the function name. This will usually be the function prototype.

Program 6-24 uses a function that displays asterisks on the screen. Arguments are passed to the function specifying how many columns and rows of asterisks to display. Default arguments are provided to display one row of 10 asterisks.

Program 6-24

```

1 // This program demonstrates default function arguments.
2 #include <iostream>
3 using namespace std;
4
5 // Function prototype with default arguments
6 void displayStars(int = 10, int = 1);
7
8 int main()
9 {
10     displayStars();          // Use default values for cols and rows.
11     cout << endl;
12     displayStars(5);        // Use default value for rows.
13     cout << endl;
14     displayStars(7, 3);     // Use 7 for cols and 3 for rows.
15     return 0;
16 }
17
18 //*****
19 // Definition of function displayStars.
20 // The default argument for cols is 10 and for rows is 1.
21 // This function displays a square made of asterisks.
22 //*****
23
24 void displayStars(int cols, int rows)
25 {
26     // Nested loop. The outer loop controls the rows
27     // and the inner loop controls the columns.
28     for (int down = 0; down < rows; down++)
29     {
30         for (int across = 0; across < cols; across++)
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 6-24, it is not possible to omit the argument for `cols` without also omitting the argument for `rows`. For example, the following function call would be illegal:

```
displayStars(, 3); // Illegal function call.
```

It's possible for a function to have some parameters with default arguments, and some without. For example, in the following function (which displays an employee's gross pay), 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 << fixed << showpoint << setprecision(2);
    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`) since they have no default arguments. Here are examples of valid calls:

```
calcPay(769, 15.75);      // Use default arg for 40 hours
calcPay(142, 12.00, 20);  // Specify number of hours
```

When a function uses a mixture of parameters with and without default arguments, the parameters with default arguments must be defined last. In the `calcPay` function, `hours` could not have been defined 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 be left out too.
- When a function has a mixture of parameters both with and without default arguments, the parameters with default arguments must be declared last.

6.13

Using Reference Variables as Parameters

CONCEPT: When used as parameters, reference variables allow a function to access the parameter's original argument. Changes to the parameter are also made to the argument.

Earlier you saw that arguments are normally passed to a function by value, and that the function cannot change the source of the argument. C++ provides a special type of variable

called a *reference variable* that, when used as a function parameter, allows access to the original argument.

A reference variable is an alias for another variable. Any changes made to the reference variable are actually performed on the variable for which it is an alias. By using a reference variable as a parameter, a function may change a variable that is defined in another function.

Reference variables are defined like regular variables, except you place an ampersand (&) in front of the name. For example, the following function definition makes the parameter `refVar` a reference variable:

```
void doubleNum(int &refVar)
{
    refVar *= 2;
}
```



NOTE: The variable `refVar` is called “a reference to an int.”

This function doubles `refVar` by multiplying it by 2. Since `refVar` is a reference variable, this action is actually performed on the variable that was passed to the function as an argument. When prototyping a function with a reference variable, be sure to include the ampersand after the data type. Here is the prototype for the `doubleNum` function:

```
void doubleNum(int &);
```



NOTE: Some programmers prefer not to put a space between the data type and the ampersand. The following prototype is equivalent to the one above:

```
void doubleNum(int &);
```



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 6-25 demonstrates how the `doubleNum` function works.

Program 6-25

```
1 // This program uses a reference variable as a function
2 // parameter.
3 #include <iostream>
4 using namespace std;
5
6 // Function prototype. The parameter is a reference variable.
7 void doubleNum(int &);
```

(program continues)

Program 6-25

(continued)

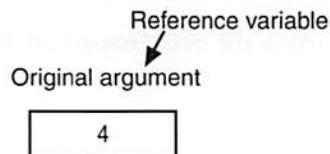
```

8
9 int main()
10 {
11     int value = 4;
12
13     cout << "In main, value is " << value << endl;
14     cout << "Now calling doubleNum..." << endl;
15     doubleNum(value);
16     cout << "Now back in main. value is " << value << endl;
17     return 0;
18 }
19
20 //*****
21 // Definition of doubleNum.
22 // The parameter refVar is a reference variable. The value *
23 // in refVar is doubled.
24 //*****
25
26 void doubleNum (int &refVar)
27 {
28     refVar *= 2;
29 }
```

Program Output

In main, value is 4
 Now calling doubleNum...
 Now back in main. value is 8

The parameter `refVar` in Program 6-25 “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 to which it points. This is illustrated in Figure 6-15.

Figure 6-15 Reference variable

Recall that function arguments are normally passed by value, which means a copy of the argument’s value is passed into the parameter variable. When a reference parameter is used, it is said that the argument is *passed by reference*.

Program 6-26 is a modification of Program 6-25. The function `getNum` has been added. The function asks the user to enter a number, which is stored in `userNum`. `userNum` is a reference to `main`’s variable `value`.

Program 6-26

```
1 // This program uses reference variables as function parameters.
2 #include <iostream>
3 using namespace std;
4
5 // Function prototypes. Both functions use reference variables
6 // as parameters.
7 void doubleNum(int &);
8 void getNum(int &);
9
10 int main()
11 {
12     int value;
13
14     // Get a number and store it in value.
15     getNum(value);
16
17     // Double the number stored in value.
18     doubleNum(value);
19
20     // Display the resulting number.
21     cout << "That value doubled is " << value << endl;
22     return 0;
23 }
24
25 //*****
26 // Definition of getNum.
27 // The parameter userNum is a reference variable. The user is *
28 // asked to enter a number, which is stored in userNum. *
29 //*****
30
31 void getNum(int &userNum)
32 {
33     cout << "Enter a number: ";
34     cin >> userNum;
35 }
36
37 //*****
38 // Definition of doubleNum.
39 // The parameter refVar is a reference variable. The value * *
40 // in refVar is doubled. *
41 //*****
42
43 void doubleNum (int &refVar)
44 {
45     refVar *= 2;
46 }
```

Program Output with Example Input Shown in BoldEnter a number: **12**

That value doubled is 24



NOTE: Only variables may be passed by reference. If you attempt to pass a nonvariable argument, such as a literal, a constant, or an expression, into a reference parameter, an error will result. Using the `doubleNum` function as an example, the following statements will generate an error:

```
doubleNum(5);           // Error
doubleNum(userNum + 10); // Error
```

If a function uses more than one reference variable as a parameter, be sure to place the ampersand before each reference variable name. Here is the prototype and definition for a function that uses four reference variable parameters:

```
// Function prototype with four reference variables
// as parameters.
void addThree(int &, int &, int &, int &);

// Definition of addThree.
// All four parameters are reference variables.
void addThree(int &sum, int &num1, int &num2, int &num3)
{
    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 situation requires them.



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 6-13 there is a `square` function that uses a `double` parameter. But, suppose you also wanted a `square` function that works exclusively with integers, accepting 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* functions. 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 userFloat;
14
15 // Get an int and a double.
16 cout << fixed << showpoint << setprecision(2);
17 cout << "Enter an integer and a floating-point value: ";
18 cin >> userInt >> userFloat;
19
20 // Display their squares.
21 cout << "Here are their squares: ";
22 cout << square(userInt) << " and " << square(userFloat);
23 return 0;
24 }
25
26 //*****
27 // Definition of overloaded function square.
28 // This function uses an int parameter, number. It returns the
29 // square of number as an int.
30 //*****
```

```
31
32 int square(int number)
33 {
34 return number * number;
35 }
36
37 //***** Definition of overloaded function square. ****
38 // This function uses a double parameter, number. It returns *
39 // the square of number as a double. *
40 //*****
41
42 double square(double number)
43 {
44 return number * number;
45 }
```

**Program Output with Example Input Shown in Bold**

Enter an integer and a floating-point value: **12 4.2**  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 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 all may be used in the same program. Program 6-28 is an example that 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 paid or salaried employees.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 // Function prototypes
8 void getChoice(char &);
9 double calcWeeklyPay(int, double);
10 double calcWeeklyPay(double);
11
12 int main()
13 {
14 char selection; // Menu selection
15 int worked; // Hours worked
16 double rate; // Hourly pay rate
17 double yearly; // Yearly 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 paid employee, or \n";
25 cout << "(S) a salaried employee?\n";
26 getChoice(selection);
27
28 // Process the menu selection.
29 switch (selection)
30 {
31 // Hourly paid 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
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 break;
48 }
49 return 0;
50 }
51
52 //***** Definition of function getChoice. *****
53 // The parameter letter is a reference to a char. *
54 // This function asks the user for an H or an S and returns *
55 // the validated input. *
56 //***** Definition of overloaded function calcWeeklyPay. *****
57
58 void getChoice(char & letter)
59 {
60 // Get the user's selection.
61 cout << "Enter your choice (H or S): ";
62 cin >> letter;
63
64 // Validate the selection.
65 while (letter != 'H' && letter != 'h' &&
66 letter != 'S' && letter != 's')
67 {
68 cout << "Please enter H or S: ";
69 cin >> letter;
70 }
71 }
72
73
74 //***** Definition of overloaded function calcWeeklyPay. *****
75 // This function calculates the gross weekly pay of *
76 // an hourly paid employee. The parameter hours holds the *
77 // number of hours worked. The parameter payRate holds the *
78 // hourly pay rate. The function returns the weekly salary. *
79 //***** Definition of overloaded function calcWeeklyPay. *****
80
81 double calcWeeklyPay(int hours, double payRate)
82 {

```

(program continues)

**Program 6-28**

(continued)

```

84 return hours * payRate;
85 }
86
87 //*****
88 // Definition of overloaded function calcWeeklyPay.
89 // This function calculates the gross weekly pay of
90 // a salaried employee. The parameter holds the employee's
91 // annual salary. The function returns the weekly salary.
92 //*****
93
94 double calcWeeklyPay(double annSalary)
95 {
96 return annSalary / 52;
97 }
```

**Program Output with Example Input Shown in Bold**

Do you want to calculate the weekly pay of  
 (H) an hourly paid employee, or  
 (S) a salaried employee?

Enter your choice (H or S): **H**

How many hours were worked? **40**

What is the hourly pay rate? **18.50**

The gross weekly pay is \$740.00

**Program Output with Example Input Shown in Bold**

Do you want to calculate the weekly pay of  
 (H) an hourly paid employee, or  
 (S) a salaried employee?

Enter your choice (H or S): **S**

What is the annual salary? **68000.00**

The gross weekly pay is \$1307.69

## 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 the `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 rare circumstances make it necessary 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 its use.

**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 for the exit function
5 using namespace std;
6
7 void function(); // Function prototype
8
9 int main()
10 {
11 function();
12 return 0;
13 }
14
15 //*****
16 // This function simply demonstrates that exit can be used *
17 // to terminate a program from a function other than main. *
18 //*****
19
20 void function()
21 {
22 cout << "This program terminates with the exit function.\n";
23 cout << "Bye!\n";
24 exit(0);
25 cout << "This message will never be displayed\n";
26 cout << "because the program has already terminated.\n";
27 }
```

**Program Output**

This program terminates with the exit function.  
Bye!

To use the `exit` function, you must 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);
```

**NOTE:** Generally, the exit code is important only if you know it will be tested outside the program. If it is not used, just pass zero, or `EXIT_SUCCESS`.



**WARNING!** The `exit()` function unconditionally shuts down your program. Because it bypasses a program's normal logical flow, you should use it with caution.



## Checkpoint

- 6.24 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;
}
```

- 6.25 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 6-10 (the modular health club membership program), it might look like this:

```
void showFees(double memberRate, int months)
{
 cout << "The showFees function was called with "
 << "the following arguments:\n"
 << "memberRate: " << memberRate << endl
 << "months: " << months << endl;
}
```

The following is an example output of the program if it were run with the stub instead of the actual `showFees` function. (A version of the health club program using this stub function is available from Computer Science Portal at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis). The program is named `HealthClubWithStub.cpp`.)

```
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? 4 [Enter]
The showFees function was called with the following arguments:
memberRate: 40.00
months: 4
```

```
Health Club Membership Menu
1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program
Enter your choice: 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 to 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 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 shows a driver for testing the showFees function in the health club membership program.

### Program 6-30

```
1 // This program is a driver for testing the showFees function.
2 #include <iostream>
3 using namespace std;
4
5 // Prototype
6 void showFees(double, int);
7
8 int main()
9 {
10 // Constants for membership rates
11 const double ADULT = 40.0;
12 const double SENIOR = 30.0;
13 const double CHILD = 20.0;
14
15 // Perform a test for adult membership.
16 cout << "Testing an adult membership...\n"
17 << "Calling the showFees function with arguments "
18 << ADULT << " and 10.\n";
19 showFees(ADULT, 10);
20
21 // Perform a test for senior citizen membership.
22 cout << "\nTesting a senior citizen membership...\n"
23 << "Calling the showFees function with arguments "
24 << SENIOR << " and 10.\n";
25 showFees(SENIOR, 10);
26
27 // Perform a test for child membership.
28 cout << "\nTesting a child membership...\n"
29 << "Calling the showFees function with arguments "
30 << CHILD << " and 10.\n";
31 showFees(CHILD, 10);
32
33 }
34
35 //*****
36 // Definition of function showFees. The memberRate parameter holds *
37 // the monthly membership rate and the months parameter holds the *
38 // number of months. The function displays the total charges. *
39 //*****
40
41 void showFees(double memberRate, int months)
42 {
43 cout << "The total charges are $"
44 << (memberRate * months) << endl;
45 }
```

**Program Output**

```
Testing an adult membership...
Calling the showFees function with arguments 40 and 10.
The total charges are $400

Testing a senior citizen membership...
Calling the showFees function with arguments 30 and 10.
The total charges are $300

Testing a child membership...
Calling the showFees function with arguments 20 and 10.
The total charges are $200
```

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.

Case Study: See High Adventure Travel Agency Part 1 Case Study on the Computer Science Portal at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).

**Review Questions and Exercises****Short Answer**

1. Why do local variables lose their values between calls to the function in which they are defined?
2. What is the difference between an argument and a parameter variable?
3. Where do you define parameter variables?
4. 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 do you do?
5. When a function accepts multiple arguments, does it matter in what order the arguments are passed?
6. How do you return a value from a function?
7. What is the advantage of breaking your application's code into several small procedures?
8. How would a `static` local variable be useful?
9. Give an example where passing an argument by reference would be useful.

**Fill-in-the-Blank**

10. The \_\_\_\_\_ is the part of a function definition that shows the function name, return type, and parameter list.
11. If a function doesn't return a value, the word \_\_\_\_\_ will appear as its return type.
12. Either a function's \_\_\_\_\_ or its \_\_\_\_\_ must precede all calls to the function.
13. Values that are sent into a function are called \_\_\_\_\_.

14. Special variables that hold copies of function arguments are called \_\_\_\_\_.
15. When only a copy of an argument is passed to a function, it is said to be passed by \_\_\_\_\_.
16. A(n) \_\_\_\_\_ eliminates the need to place a function definition before all calls to the function.
17. A(n) \_\_\_\_\_ variable is defined inside a function and is not accessible outside the function.
18. \_\_\_\_\_ variables are defined outside all functions and are accessible to any function within their scope.
19. \_\_\_\_\_ variables provide an easy way to share large amounts of data among all the functions in a program.
20. Unless you explicitly initialize global variables, they are automatically initialized to \_\_\_\_\_.
21. If a function has a local variable with the same name as a global variable, only the \_\_\_\_\_ variable can be seen by the function.
22. \_\_\_\_\_ local variables retain their value between function calls.
23. The \_\_\_\_\_ statement causes a function to end immediately.
24. \_\_\_\_\_ arguments are passed to parameters automatically if no argument is provided in the function call.
25. When a function uses a mixture of parameters with and without default arguments, the parameters with default arguments must be defined \_\_\_\_\_.
26. The value of a default argument must be a(n) \_\_\_\_\_.
27. When used as parameters, \_\_\_\_\_ variables allow a function to access the parameter's original argument.
28. Reference variables are defined like regular variables, except there is a(n) \_\_\_\_\_ in front of the name.
29. Reference variables allow arguments to be passed by \_\_\_\_\_.
30. The \_\_\_\_\_ function causes a program to terminate.
31. Two or more functions may have the same name, as long as their \_\_\_\_\_ are different.

### Algorithm Workbench

32. Examine the following function header, then write an example call to the function:  

```
void showValue(int quantity)
```
33. The following statement calls a function named `half`. The `half` function returns a value that is half that of the argument. Write the 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 argument. When the function is called, it should display the product of its argument multiplied by 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 procedure and passes the following variables to it:

```
int age;
double income;
char initial;
```

37. Write a function named `getNumber` that uses a reference parameter variable 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.

### True or False

38. T F Functions should be given names that reflect their purpose.
39. T F Function headers are terminated with a semicolon.
40. T F Function prototypes are terminated with a semicolon.
41. T F If other functions are defined before `main`, the program still starts executing at function `main`.
42. T F When a function terminates, it always branches back to `main`, regardless of where it was called from.
43. T F Arguments are passed to the function parameters in the order they appear in the function call.
44. T F The scope of a parameter is limited to the function that uses it.
45. T F Changes to a function parameter always affect the original argument as well.
46. T F In a function prototype, the names of the parameter variables may be left out.
47. T F Many functions may have local variables with the same name.
48. T F Overuse of global variables can lead to problems.
49. T F Static local variables are not destroyed when a function returns.
50. T F All static local variables are initialized to -1 by default.
51. T F Initialization of static local variables only happens once, regardless of how many times the function in which they are defined is called.
52. T F When a function with default arguments is called and an argument is left out, all arguments that come after it must be left out as well.

53. T F It is not possible for a function to have some parameters with default arguments and some without.
54. T F The `exit` function can only be called from `main`.
55. T F A stub is a dummy function that is called instead of the actual function it represents.

### Find the Errors

Each of the following functions has errors. Locate as many errors as you can.

- ```
56. void total(int value1, value2, value3)
{
    return value1 + value2 + value3;
}

57. double average(int value1, int value2, int value3)
{
    double average;
    average = value1 + value2 + value3 / 3;
}

58. void area(int length = 30, int width)
{
    return length * width;
}

59. void getValue(int value&)
{
    cout << "Enter a value: ";
    cin >> value&;
}

60. (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;
}
```

Programming Challenges

1. Markup

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 percent, then the item's retail price is 10.00.

- If an item's wholesale cost is 5.00 and its markup percentage is 50 percent, 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 markup percentage.

2. Rectangle Area—Complete the Program

If you have downloaded this book's source code, you will find a partially written program named `AreaRectangle.cpp` in the Chapter 06 folder. Your job is to complete the program. When it is complete, the program will ask the user to enter the width and length of a rectangle, then display the rectangle's area. The program calls the following functions, which have not been written:

- `getLength`—This function should ask the user to enter the rectangle's length then return that value as a `double`.
- `getWidth`—This function should ask the user to enter the rectangle's width then return that value as a `double`.
- `getArea`—This function should accept the rectangle's length and width as arguments and return the rectangle's area. The area is calculated by multiplying the length by the width.
- `displayData`—This function should accept the rectangle's length, width, and area as arguments and display them in an appropriate message on the screen.

3. 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.

4. Safest Driving Area

Write a program that determines which of five geographic regions within a major city (north, south, east, west, and central) had the fewest reported automobile 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 automobile 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.

5. Falling Distance

When an object is falling because of gravity, the following formula can be used to determine the distance the object falls 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 amount of 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.

6. 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, 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.

7. 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.

8. Coin Toss

Write a function named `coinToss` that simulates the tossing of a coin. When you call the function, it should generate a random number in the range of 1 through 2. If the random number is 1, the function should display "heads." If the random number is 2, the function should display "tails." Demonstrate the function in a program that asks the user how many times the coin should be tossed, then simulates the tossing of the coin that number of times.

9. 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? You can use the following formula, which is known as the present value formula, to find out:

$$P = \frac{F}{(1 + r)^n}$$

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, F is \$10,000.)
- r is the **annual interest rate**.
- n is the **number of years** that you plan to let the money sit in the account.

Write a program that has 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.

10. Future Value

Suppose you have a certain amount of money in a savings account that earns compound monthly interest, and you want to calculate the amount that you will have after a specific number of months. The formula, which is known as the future value formula, is

$$F = P \times (1 + i)^t$$

The terms in the formula are as follows:

- F is the **future value** of the account after the specified time period.
- P is the **present value** of the account.
- i is the **monthly interest rate**.
- t is the **number of months**.

Write a program that prompts the user to enter the account's present value, monthly interest rate, and the number of months that the money will be left in the account. The program should pass these values to a function named `futureValue` that returns the future value of the account, after the specified number of months. The program should display the account's future value.

11. 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 which of the five scores to drop.

Input Validation: Do not accept test scores lower than 0 or higher than 100.

12. Star Search

A particular talent competition has five 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 three remaining scores. Write a program that uses this method to calculate 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 five judges.
- `void calcScore()` should calculate and display the average of the three 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 five 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 five scores passed to it.
- `int findHighest()` should find and return the highest of the five scores passed to it.

Input Validation: Do not accept judge scores lower than 0 or higher than 10.

13. Days Out

Write a program that calculates the average number of days a company's employees are absent. The program should have the following functions:

- A function called by `main` that asks the user for the number of employees in the company. This value should be returned as an `int`. (The function accepts no arguments.)
- A function called by `main` that accepts one argument: the number of employees in the company. The function should ask the user to enter the number of days each employee missed during the past year. The total of these days should be returned as an `int`.
- A function called by `main` that takes two arguments: the number of employees in the company and the total number of days absent for all employees during the year. The function should return, as a `double`, the average number of days absent. (This function does not perform screen output and does not ask the user for input.)

Input Validation: Do not accept a number less than 1 for the number of employees. Do not accept a negative number for the days any employee missed.

14. Order Status

The Middletown Wholesale Copper Wire Company sells spools of copper wiring for \$100 each. Write a program that displays the status of an order. The program should have a function that asks for the following data:

- The number of spools ordered
- The number of spools in stock
- Whether there are special shipping and handling charges

(Shipping and handling is normally \$10 per spool.) If there are special charges, the program should ask for the special charges per spool.

The gathered data should be passed as arguments to another function that displays:

- The number of spools ready to ship from current stock
- The number of spools on backorder (if the number ordered is greater than what is in stock)
- Subtotal 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 shipping and handling charges.

15. 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 inpatient or an outpatient. If the patient was an inpatient, the following data should be entered:

- The number of days spent in the hospital
- The daily rate
- Hospital medication charges
- Charges for hospital services (lab tests, etc.)

The program should ask for the following data if the patient was an outpatient:

- 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 inpatient data, while the other function accepts arguments for outpatient information. Both functions should return the total charges.

Input Validation: Do not accept negative numbers for any data.

16. 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 displays the size of a population for any number of years. The program should ask for the following data:

- The starting size of a population
- The annual birth rate
- The annual death rate
- The number of years to display

Write a function that calculates the size of the population for a year. The formula is

$$N = P + BP - DP$$

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.

17. Transient Population

Modify Programming Challenge 16 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 who typically move into the area each year, and the number of individuals who 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.

18. Paint Job Estimator

A painting company has determined that for every 110 square feet of wall space, 1 gallon of paint and 8 hours of labor will be required. The company charges \$25.00 per hour for labor. Write a modular program that allows the user to enter the number of rooms that are to be painted and the price of the paint per gallon. It should also ask for the square feet of wall space in each room. It should then display the following data:

- The number of gallons of paint required
- The hours of labor required
- The cost of the paint
- The labor charges
- The total cost of the paint job

Input validation: Do not accept a value less than 1 for the number of rooms. Do not accept a value less than \$10.00 for the price of paint. Do not accept a negative value for square footage of wall space.

19. Using Files—Hospital Report

Modify Programming Challenge 15 (Overloaded Hospital) to write the report it creates to a file.

20. 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.

21. Multiple Stock Sales

Use the function that you wrote for Programming Challenge 20 (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, then display the total.

22. `isPrime` Function

A prime number is a number that is only evenly divisible by itself and 1. 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 evenly by 1, 2, 3, and 6.

Write a function name `isPrime`, which takes an integer as an argument and returns `true` if the argument is a prime number, or `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`.

23. Prime Number List

Use the `isPrime` function that you wrote in Programming Challenge 22 (`isPrime` function) in a program that stores a list of all the prime numbers from 1 through 100 in a file.

24. Rock, Paper, Scissors Game

Write a program that lets the user play the game of Rock, Paper, Scissors against the computer. The program should work as follows:

- When the program begins, a random number in the range of 1 through 3 is generated. If the number is 1, then the computer has chosen rock. If the number is 2, then the computer has chosen paper. If the number is 3, then the computer has chosen scissors. (Don't display the computer's choice yet.)
- The user enters his or her choice of "rock", "paper", or "scissors" at the keyboard. (You can use a menu if you prefer.)
- The computer's choice is displayed.
- A winner is selected according to the following rules:
 - If one player chooses rock and the other player chooses scissors, then rock wins. (The rock smashes the scissors.)
 - If one player chooses scissors and the other player chooses paper, then scissors wins. (Scissors cuts paper.)
 - If one player chooses paper and the other player chooses rock, then paper wins. (Paper wraps rock.)
 - If both players make the same choice, the game must be played again to determine the winner.

Be sure to divide the program into functions that perform each major task.

Group Project**25. 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 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.
- The parameters and return types of each function should be decided in advance.
- Stubs and drivers should be used to test and debug the program.
- The program can be implemented 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 businessperson 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. Calculate the vehicle expense as \$0.27 per mile driven
- Parking fees (The company allows up to \$6 per day. Anything in excess of this must be paid by the employee.)
- Taxi fees, if a taxi was used anytime during the trip (The company allows up to \$10 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 amount 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 12 noon. Dinner is allowed on the first day 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 on the last day if the time of arrival is after 7 p.m. The program should only ask for the amounts of allowable meals. (The company allows up to \$9 for breakfast, \$12 for lunch, and \$16 for dinner. Anything in excess of this must be paid by the employee.)

The program should calculate and display the total expenses incurred by the businessperson, the total allowable expenses for the trip, the excess that must be reimbursed by the businessperson, if any, and the amount saved by the businessperson if the expenses were under the total allowed.

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.

TOPICS

- | | |
|--|--|
| 7.1 Arrays Hold Multiple Values | 7.7 Arrays as Function Arguments |
| 7.2 Accessing Array Elements | 7.8 Two-Dimensional Arrays |
| 7.3 No Bounds Checking in C++ | 7.9 Arrays with Three or More Dimensions |
| 7.4 The Range-Based for Loop | 7.10 Focus on Problem Solving and |
| 7.5 Processing Array Contents | Program Design: A Case Study |
| 7.6 Focus on Software Engineering: Using Parallel Arrays | 7.11 Introduction to the STL vector |

7.1**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 7-1 causes only enough memory to be reserved to hold one value of the specified data type.

Figure 7-1 Variables can hold one value at a time

```
int count;    Enough memory for 1 int
```

```
12314
```

```
float price;  Enough memory for 1 float
```

```
56.981
```

```
char letter;  Enough memory for 1 char
```

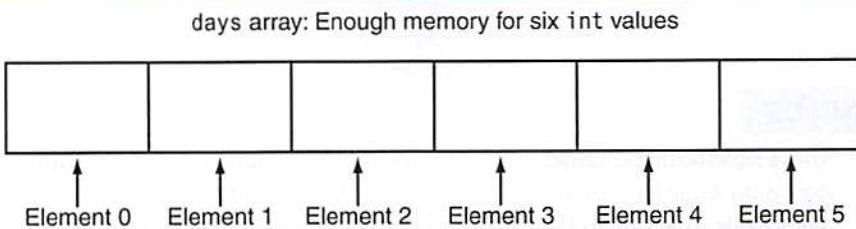
```
A
```

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 days[6];
```

The name of this array is `days`. The number inside the brackets is the array's *size declarator*. It indicates the number of *elements*, or values, the array can hold. The `days` array can store six elements, each one an integer. This is depicted in Figure 7-2.

Figure 7-2 An array with six elements



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 in the following:

```
const int NUM_DAYS = 6;
int days[NUM_DAYS];
```

Arrays of any data type can be defined. The following are all valid array definitions:

```
float temperatures[100]; // Array of 100 floats
string names[10]; // Array of 10 string objects
long units[50]; // Array of 50 long integers
double sizes[1200]; // Array of 1200 doubles
```

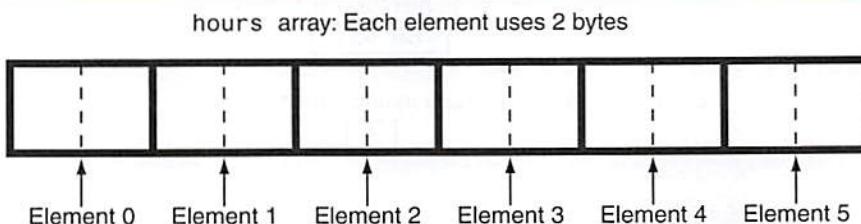
Memory Requirements of Arrays

The amount of memory used by an array depends on the array's data type and the number of elements. The `hours` array, defined here, is an array of six shorts.

```
short hours[6];
```

On a typical system, a `short` uses 2 bytes of memory, so the `hours` array would occupy 12 bytes. This is shown in Figure 7-3.

Figure 7-3 Array memory usage



The size of an array can be calculated by multiplying the size of an individual element by the number of elements in the array. Table 7-1 shows the typical sizes of various arrays.

Table 7-1 Arrays and Their sizes

Array Definition	Number of Elements	Size of Each Element	Size of the Array
char letters[25];	25	1 byte	25 bytes
short rings[100];	100	2 bytes	200 bytes
int miles[84];	84	4 bytes	336 bytes
float temp[12];	12	4 bytes	48 bytes
double distance[1000];	1000	8 bytes	8000 bytes

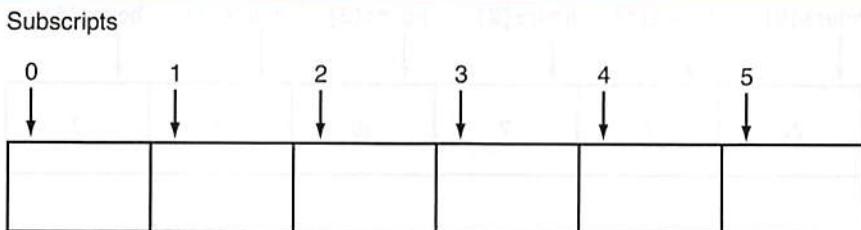
7.2

Accessing Array Elements

CONCEPT: The individual elements of an array are assigned unique subscripts. These subscripts are used to access the 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 array hours would have the subscripts 0 through 5. This is shown in Figure 7-4.

Figure 7-4 Subscripts



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 7-4, the element `hours[6]` does not exist. `hours[5]` is the last element in the array.

Each element in the `hours` array, when accessed by its subscript, can be used as a `short` variable. Here is an example of a statement that stores the number 20 in the first element of the array:

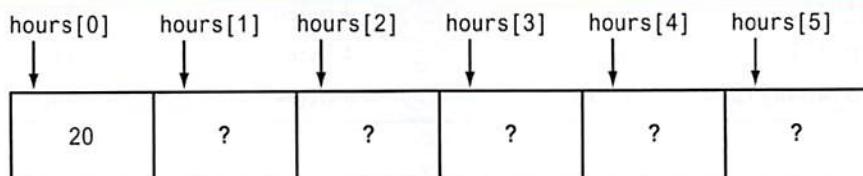
```
hours[0] = 20;
```



NOTE: The expression `hours[0]` is pronounced “hours sub zero.” You would read this assignment statement as “hours sub zero is assigned twenty.”

Figure 7-5 shows the contents of the `hours` array after the statement assigns 20 to `hours[0]`.

Figure 7-5 Contents of the `hours` array



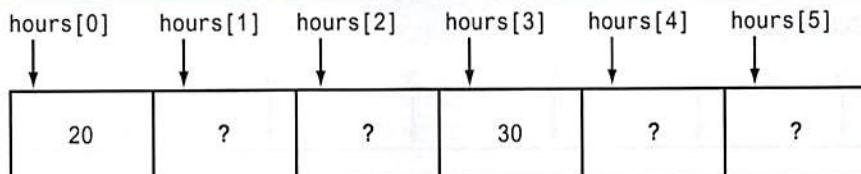
NOTE: Because values have not been assigned to the other elements of the array, question marks will be used to indicate that the contents of those elements are unknown. If an array is defined globally, all of its elements are initialized to zero by default. Local arrays, however, have no default initialization value.

The following statement stores the integer 30 in `hours[3]`:

```
hours[3] = 30;
```

Figure 7-6 shows the contents of the array after the previous statement executes.

Figure 7-6 Contents of the `hours` array



NOTE: Understand the difference between the array size declarator and a subscript. The number inside the brackets of an array definition is the size declarator. The number inside the brackets of an assignment statement or any statement that works with the contents of an array is a subscript.

Inputting and Outputting Array Contents

Array elements may be used with the `cin` and `cout` objects like any other variable. Program 7-1 shows the array `hours` being used to store and display values entered by the user.

Program 7-1

```
1 // This program asks for the number of hours worked
2 // by six employees. It stores the values in an array.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     const int NUM_EMPLOYEES = 6;
9     int hours[NUM_EMPLOYEES];
10
11    // Get the hours worked by each employee.
12    cout << "Enter the hours worked by "
13        << NUM_EMPLOYEES << " employees: ";
14    cin >> hours[0];
15    cin >> hours[1];
16    cin >> hours[2];
17    cin >> hours[3];
18    cin >> hours[4];
19    cin >> hours[5];
20
21    // Display the values in the array.
22    cout << "The hours you entered are:";
23    cout << " " << hours[0];
24    cout << " " << hours[1];
25    cout << " " << hours[2];
26    cout << " " << hours[3];
27    cout << " " << hours[4];
28    cout << " " << hours[5] << endl;
29
30 }
```

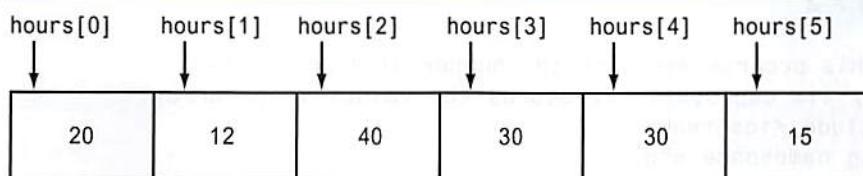
Program Output with Example Input Shown in Bold

Enter the hours worked by 6 employees: **20 12 40 30 30 15**

The hours you entered are: 20 12 40 30 30 15

Figure 7-7 shows the contents of the array hours with the values entered by the user in the example output above.

Figure 7-7 Contents of the hours array



Even though the size declarator of an array definition must be a constant or a literal, subscript numbers can be stored in variables. This makes it possible to use a loop to “step through” an entire array, performing the same operation on each element. For example, look at the following code:

```
const int ARRAY_SIZE = 5;
int numbers[ARRAY_SIZE];

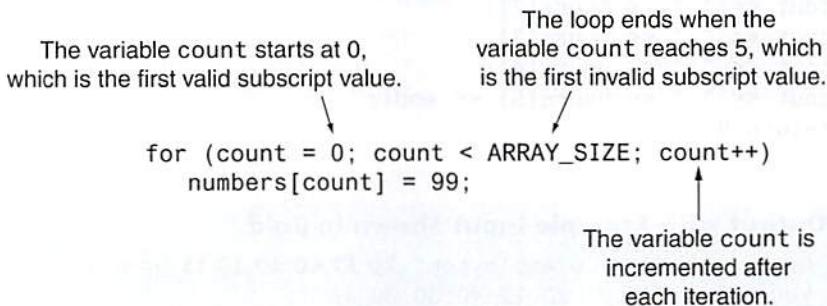
for (int count = 0; count < ARRAY_SIZE; count++)
    numbers[count] = 99;
```



This code first defines a constant `int` named `ARRAY_SIZE` and initializes it with the value 5. Then, it defines an `int` array named `numbers`, using `ARRAY_SIZE` as the size declarator. As a result, the `numbers` array will have five elements. The `for` loop uses a counter variable named `count`. This loop will iterate five times, and during the loop iterations the `count` variable will take on the values 0 through 4.

Notice the statement inside the loop uses the `count` variable as a subscript. It assigns 99 to `numbers[count]`. During the first iteration, 99 is assigned to `numbers[0]`. During the next iteration, 99 is assigned to `numbers[1]`. This continues until 99 has been assigned to all of the array’s elements. Figure 7-8 illustrates that the loop’s initialization, test, and update expressions have been written so that the loop starts and ends the counter variable with valid subscript values (0 through 4). This ensures that only valid subscripts are used in the body of the loop.

Figure 7-8 Initialization, test, and update expressions



Program 7-1 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 7-2.

Program 7-2

```
1 // This program asks for the number of hours worked
2 // by six employees. It stores the values in an array.
3 #include <iostream>
4 using namespace std;
5
```

```

6 int main()
7 {
8     const int NUM_EMPLOYEES = 6; // Number of employees
9     int hours[NUM_EMPLOYEES]; // Each employee's hours
10    int count; // Loop counter
11
12    // Input the hours worked.
13    for (count = 0; count < NUM_EMPLOYEES; count++)
14    {
15        cout << "Enter the hours worked by employee "
16            << (count + 1) << ": ";
17        cin >> hours[count];
18    }
19
20    // Display the contents of the array.
21    cout << "The hours you entered are:";
22    for (count = 0; count < NUM_EMPLOYEES; count++)
23        cout << " " << hours[count];
24    cout << endl;
25
26    return 0;
}

```

Program Output with Example Input Shown in Bold

```

Enter the hours worked by employee 1: 20 Enter
Enter the hours worked by employee 2: 12 Enter
Enter the hours worked by employee 3: 40 Enter
Enter the hours worked by employee 4: 30 Enter
Enter the hours worked by employee 5: 30 Enter
Enter the hours worked by employee 6: 15 Enter
The hours you entered are: 20 12 40 30 30 15

```

The first for loop, in lines 13 through 18, prompts the user for each employee's hours. Take a closer look at lines 15 through 17:

```

cout << "Enter the hours worked by employee "
    << (count + 1) << ": ";
cin >> hours[count];

```

Notice the cout statement uses the expression count + 1 to display the employee number, but the cin statement uses count as the array subscript. This is because the hours for employee number 1 are stored in hours[0], the hours for employee number 2 are stored in hours[1], and so forth.

The loop in lines 22 and 23 also uses the count variable to step through the array, displaying each element.



NOTE: You can use any integer expression as an array subscript. For example, the first loop in Program 7-2 could have been written like this:

```
for (count = 1; count <= NUM_EMPLOYEES; count++)
{
    cout << "Enter the hours worked by employee "
        << count << ": ";
    cin >> hours[count - 1];
}
```

In this code, the `cin` statement uses the expression `count - 1` as a subscript.

Inputting data into an array must normally be done one element at a time. For example, the following `cin` statement will not input data into the `hours` array:

```
cin >> hours; // Wrong! This will NOT work.
```

Instead, you must use multiple `cin` statements to read data into each array element, or use a loop to step through the array, reading data into its elements. Also, outputting an array's contents must normally be done one element at a time. For example, the following `cout` statement will not display the contents of the `hours` array:

```
cout << hours; // Wrong! This will NOT work.
```

Instead, you must output each element of the array separately.

Array Initialization

Like regular variables, C++ allows you to initialize an array's elements when you create the array. Here is an example:

```
const int MONTHS = 12;
int days[MONTHS] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

The series of values inside the braces and separated with commas is called an *initialization list*. 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 7-9 shows the contents of the array after the initialization.

Figure 7-9 The array after initialization

Subscripts											
0	1	2	3	4	5	6	7	8	9	10	11
31	28	31	30	31	30	31	31	30	31	30	31

Program 7-3 demonstrates how an array may be initialized.

Program 7-3

```
1 // This program displays the number of days in each month.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     const int MONTHS = 12;
8     int days[MONTHS] = { 31, 28, 31, 30,
9                         31, 30, 31, 31,
10                        30, 31, 30, 31 };
11
12     for (int count = 0; count < MONTHS; count++)
13     {
14         cout << "Month " << (count + 1) << " has ";
15         cout << days[count] << " days.\n";
16     }
17     return 0;
18 }
```

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.
```



NOTE: C++ allows you to spread the initialization list across multiple lines. Both of the following array definitions are equivalent:

```
double coins[5] = {0.05, 0.1, 0.25, 0.5, 1.0};
double coins[5] = {0.05,
                  0.1,
                  0.25,
                  0.5,
                  1.0};
```

Program 7-4 shows an example with a **string** array that is initialized with strings.

Program 7-4

```
1 // This program initializes a string array.
2 #include<iostream>
3 #include<string>
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 9;
9     string planets[SIZE] = { "Mercury", "Venus", "Earth", "Mars",
10                           "Jupiter", "Saturn", "Uranus",
11                           "Neptune", "Pluto (a dwarf planet)" };
12
13     cout << "Here are the planets:\n";
14
15     for (int count = 0; count < SIZE; count++)
16         cout << planets[count] << endl;
17     return 0;
18 }
```

Program Output

Here are the planets:
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto (a dwarf planet)

Program 7-5 shows a character array being initialized with the first ten letters of the alphabet. The array is then used to display those characters' ASCII codes.

Program 7-5

```

13     cout << "Character" << "\t" << "ASCII Code\n";
14     cout << "-----" << "\t" << "-----\n";
15     for (int count = 0; count < NUM LETTERS; count++)
16     {
17         cout << letters[count] << "\t\t";
18         cout << static_cast<int>(letters[count]) << endl;
19     }
20     return 0;
21 }
```

Program Output

Character	ASCII Code
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74

 **NOTE:** An array's initialization list cannot have more values than the array has elements.

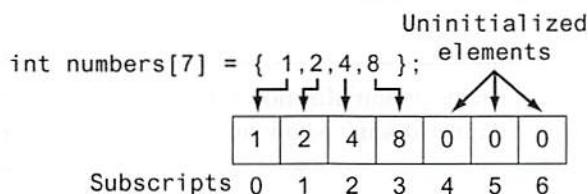
Partial Array Initialization

When an array is being initialized, C++ does not require a value for every element. It's possible to only initialize part of an array, such as:

```
int numbers[7] = {1, 2, 4, 8};
```

This definition initializes only the first four elements of a 7-element array, as illustrated in Figure 7-10.

Figure 7-10 Partial array initialization



It's important to note that if an array is partially initialized, the uninitialized elements will be set to zero. The uninitialized elements of a `string` array will contain empty strings. This is true even if the array is defined locally. (If a local array is completely uninitialized, its elements will contain “garbage,” like all other local variables.) Program 7-6 shows the contents of the `numbers` array after it is partially initialized.

Program 7-6

```

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 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
14    cout << endl;
15    return 0;
16 }
```

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 numbers[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. 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* provide an initialization list if you leave out an array's size declarator. Otherwise, the compiler doesn't know how large to make the array.

Reading Data from a File into an Array

Reading the contents of a file into an array is straightforward: Open the file and use a loop to read each item from the file, storing each item in an array element. The loop should iterate until either the array is filled or the end of the file is reached. Program 7-7 demonstrates by opening a file that has 10 numbers stored in it then reading the file's contents into an array.

Program 7-7

```
1 // This program reads data from a file into an array.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     const int ARRAY_SIZE = 10; // Array size
9     int numbers[ARRAY_SIZE]; // Array with 10 elements
10    int count = 0; // Loop counter variable
11    ifstream inputFile; // Input file stream object
12
13    // Open the file.
14    inputFile.open("TenNumbers.txt");
15
16    // Read the numbers from the file into the array.
17    while (count < ARRAY_SIZE && inputFile >> numbers[count])
18        count++;
19
20    // Close the file.
21    inputFile.close();
22
23    // Display the numbers read:
24    cout << "The numbers are: ";
25    for (count = 0; count < ARRAY_SIZE; count++)
26        cout << numbers[count] << " ";
27    cout << endl;
28    return 0;
29 }
```

Program Output

The numbers are: 101 102 103 104 105 106 107 108 109 110

The while loop in lines 17 and 18 reads items from the file and assigns them to elements of the numbers array. Notice the loop tests two Boolean expressions, connected by the && operator:

- The first expression is count < ARRAY_SIZE. The purpose of this expression is to prevent the loop from writing beyond the end of the array. If the expression is true, the second Boolean expression is tested. If this expression is false, however, the loop stops.
- The second expression is inputFile >> numbers[count]. This expression reads a value from the file and stores it in the numbers[count] array element. If a value is successfully read from the file, the expression is true and the loop continues. If no value can be read from the file, however, the expression is false and the loop stops.

Each time the loop iterates, it increments count in line 18.

Writing the Contents of an Array to a File

Writing the contents of an array to a file is also a straightforward matter. Use a loop to step through each element of the array, writing its contents to a file. Program 7-8 demonstrates this.

Program 7-8

```

1 // This program writes the contents of an array to a file.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     const int ARRAY_SIZE = 10;    // Array size
9     int numbers[ARRAY_SIZE];    // Array with 10 elements
10    int count;                 // Loop counter variable
11    ofstream outputFile;        // Output file stream object
12
13    // Store values in the array.
14    for (count = 0; count < ARRAY_SIZE; count++)
15        numbers[count] = count;
16
17    // Open a file for output.
18    outputFile.open("SavedNumbers.txt");
19
20    // Write the array contents to the file.
21    for (count = 0; count < ARRAY_SIZE; count++)
22        outputFile << numbers[count] << endl;
23
24    // Close the file.
25    outputFile.close();
26
27    // That's it!
28    cout << "The numbers were saved to the file.\n";
29    return 0;
30 }
```

Program Output

The numbers were saved to the file.

Contents of the File SavedNumbers.txt

```

0
1
2
3
4
5
6
7
8
9
```

7.3

No Bounds Checking in C++

CONCEPT: C++ does not prevent you from overwriting an array's bounds.

C++ is a popular language for software developers who have to write fast, efficient code. To increase runtime efficiency, C++ does not provide many of the common safeguards to prevent unsafe memory access found in other languages. For example, C++ does not perform array bounds checking. This means you can write programs with subscripts that go beyond the boundaries of a particular array. Program 7-9 demonstrates this capability.

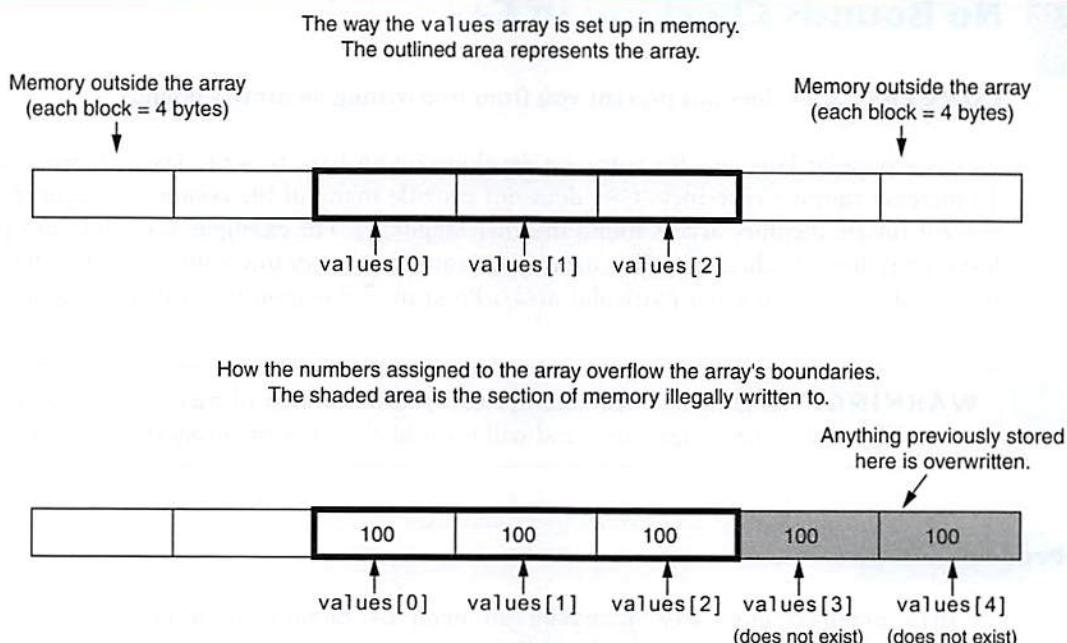


WARNING! Program 7-9 will attempt to write to an area of memory outside the array. This is an invalid operation and will most likely cause the program to crash.

Program 7-9

```
1 // This program unsafely accesses an area of memory by writing
2 // values beyond an array's boundary.
3 // WARNING: If you compile and run this program, it could crash.
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     const int SIZE = 3;    // Constant for the array size
10    int values[SIZE];    // An array of 3 integers
11    int count;           // Loop counter variable
12
13    // Attempt to store five numbers in the 3-element array.
14    cout << "I will store 5 numbers in a 3-element array!\n";
15    for (count = 0; count < 5; count++)
16        values[count] = 100;
17
18    // If the program is still running, display the numbers.
19    cout << "If you see this message, it means the program\n";
20    cout << "has not crashed! Here are the numbers:\n";
21    for (count = 0; count < 5; count++)
22        cout << values[count] << endl;
23
24 }
```

The `values` array has three integer elements, with the subscripts 0, 1, and 2. The loop, however, stores the number 100 in elements 0, 1, 2, 3, and 4. The elements with subscripts 3 and 4 do not exist, but C++ allows the program to write beyond the boundary of the array, as if those elements were there. Figure 7-11 depicts the way the array is set up in memory when the program first starts to execute, and what happens when the loop writes data beyond the boundary of the array.

Figure 7-11 Writing data beyond the bounds of an array

Although C++ programs are fast and efficient, the absence of safeguards such as array bounds checking usually proves to be a bad thing. It's easy for C++ programmers to make careless mistakes that allow programs to access areas of memory that are supposed to be off-limits. You must always 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

In 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

- 7.1 Define the following arrays:
- A) `empNums`, a 100-element array of `ints`
 - B) `payRates`, a 25-element array of `floats`
 - C) `miles`, a 14-element array of `longs`
 - D) `cityName`, a 26-element array of `string` objects
 - E) `lightYears`, a 1,000-element array of `doubles`
- 7.2 What's wrong with the following array definitions?
- ```
int readings[-1];
float measurements[4.5];
int size;
string names[size];
```
- 7.3 What would the valid subscript values be in a 4-element array of `doubles`?
- 7.4 What is the difference between an array's size declarator and a subscript?
- 7.5 What is "array bounds checking"? Does C++ perform it?
- 7.6 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;
```
- 7.7 The following program skeleton contains a 20-element array of `ints` called `fish`. When completed, the program should ask how many fish were caught by fishermen 1 through 20, and store this data in the array. Complete the program.
- ```
#include <iostream>
using namespace std;
int main()
{
 const int NUM_FISH = 20;
 int fish[NUM_FISH];
 // You must finish this program. It should ask how
 // many fish were caught by fishermen 1-20, and
 // store this data in the array fish.
 return 0;
}
```
- 7.8 Define the following arrays:
- A) `ages`, a 10-element array of `ints` initialized with the values 5, 7, 9, 14, 15, 17, 18, 19, 21, and 23.
  - B) `temps`, a 7-element array of `floats` 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 `chars` initialized with the values 'J', 'B', 'L', 'A', '\*', '\$', 'H', and 'M'.

- 7.9 Is each of the following a valid or invalid array definition? (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[];`

11

7.4

## The Range-Based for Loop

**CONCEPT:** The range-based `for` loop is a loop that iterates once for each element in an array. Each time the loop iterates, it copies an element from the array to a variable. The range-based `for` loop was introduced in C++ 11.

C++ 11 provides a specialized version of the `for` loop that, in many circumstances, simplifies array processing. It is known as the *range-based for loop*. When you use the range-based `for` loop with an array, the loop automatically iterates once for each element in the array. For example, if you use the range-based `for` loop with an 8-element array, the loop will iterate 8 times. Because the range-based `for` loop automatically knows the number of elements in an array, you do not have to use a counter variable to control its iterations, as with a regular `for` loop. Additionally, you do not have to worry about stepping outside the bounds of an array when you use the range-based `for` loop.

The range-based `for` loop is designed to work with a built-in variable known as the *range variable*. Each time the range-based `for` loop iterates, it copies an array element to the range variable. For example, the first time the loop iterates, the range variable will contain the value of element 0, the second time the loop iterates, the range variable will contain the value of element 1, and so forth.

Here is the general format of the range-based `for` loop:

```
for (dataType rangeVariable : array)
 statement;
```

Let's look at the syntax more closely as follows:

- *dataType* is the data type of the range variable. It must be the same as the data type of the array elements, or a type to which the elements can automatically be converted.
- *rangeVariable* is the name of the range variable. This variable will receive the value of a different array element during each loop iteration. During the first loop iteration, it receives the value of the first element; during the second iteration, it receives the value of the second element, and so forth.
- *array* is the name of an array on which you wish the loop to operate. The loop will iterate once for every element in the array.
- *statement* is a statement that executes during a loop iteration. If you need to execute more than one statement in the loop, enclose the statements in a set of braces.

For example, assume that you have the following array definition:

```
int numbers[] = { 3, 6, 9 };
```

You can use the following range-based for loop to display the contents of the `numbers` array:

```
for (int val : numbers)
 cout << val << endl;
```

Because the `numbers` array has three elements, this loop will iterate three times. The first time it iterates, the `val` variable will receive the value in `numbers[0]`. During the second iteration, `val` will receive the value in `numbers[1]`. During the third iteration, `val` will receive the value in `numbers[2]`. The code's output will be as follows:

```
3
6
9
```

Here is an example of a range-based for loop that executes more than one statement in the body of the loop:

```
int[] numbers = { 3, 6, 9 };
for (int val : numbers)
{
 cout << "The next value is ";
 cout << val << endl;
}
```

This code will produce the following output:

```
The next value is 3
The next value is 6
The next value is 9
```

If you wish, you can use the `auto` key word to specify the range variable's data type. Here is an example:

```
int[] numbers = { 3, 6, 9 };
for (auto val : numbers)
 cout << val << endl;
```

Program 7-10 demonstrates the range-based for loop by displaying the elements of an `int` array.

### Program 7-10

```
1 // This program demonstrates the range-based for loop.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7 // Define an array of integers.
8 int numbers[] = { 10, 20, 30, 40, 50 };
9
10 // Display the values in the array.
11 for (int val : numbers)
12 cout << val << endl;
13
14 return 0;
15 }
```

(program output continues)

**Program 7-10**

(continued)

**Program Output**

```
10
20
30
40
50
```

Program 7-11 shows another example of the range-based for loop. This program displays the elements of a string array.

**Program 7-11**

```
1 // This program demonstrates the range-based for loop.
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main()
7 {
8 string planets[] = { "Mercury", "Venus", "Earth", "Mars",
9 "Jupiter", "Saturn", "Uranus",
10 "Neptune", "Pluto (a dwarf planet)" };
11
12 cout << "Here are the planets:\n";
13
14 // Display the values in the array.
15 for (string val : planets)
16 cout << val << endl;
17
18 return 0;
19 }
```

**Program Output**

```
Here are the planets:
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto (a dwarf planet)
```

## Modifying an Array with a Range-Based for Loop

As the range-based for loop executes, its range variable contains only a copy of an array element. As a consequence, you cannot use a range-based for loop to modify the contents

of an array unless you declare the range variable as a reference. Recall from Chapter 6 that a reference variable is an alias for another value. Any changes made to the reference variable are actually made to the value for which it is an alias.

To declare the range variable as a reference variable, simply write an ampersand (&) in front of its name in the loop header. Program 7-12 shows an example.

### Program 7-12

```
1 // This program uses a range-based for loop to
2 // modify the contents of an array.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8 const int SIZE = 5;
9 int numbers[5];
10
11 // Get values for the array.
12 for (int &val : numbers)
13 {
14 cout << "Enter an integer value: ";
15 cin >> val;
16 }
17
18 // Display the values in the array.
19 cout << "Here are the values you entered:\n";
20 for (int val : numbers)
21 cout << val << endl;
22
23 return 0;
24 }
```

### Program Output with Example Input Shown in Bold

```
Enter an integer value: 1 Enter
Enter an integer value: 2 Enter
Enter an integer value: 3 Enter
Enter an integer value: 4 Enter
Enter an integer value: 5 Enter

Here are the values you entered:
1
2
3
4
5
```

Notice in line 12 the range variable, `val`, has an ampersand (&) written in front of its name. This declares `val` as a reference variable. As the loop executes, the `val` variable will not merely contain a copy of an array element, but it will be an alias for the element. Any changes made to the `val` variable will actually be made to the array element it references.

Also notice in line 20, we did not declare `val` as a reference variable (there is no ampersand written in front of the variable's name). Because the loop is simply displaying the array elements, and does not need to change the array's contents, there is no need to make `val` a reference variable.

## The Range-Based for Loop versus the Regular for Loop

The range-based `for` loop can be used in any situation where you need to step through the elements of an array, and you do not need to use the element subscripts. It will not work, however, in situations where you need the element subscript for some purpose. In those situations, you need to use the regular `for` loop.



**NOTE:** You can use the `auto` key word with a reference range variable. For example, the code in lines 12 through 16 in Program 7-12 could have been written like this:

```
for (auto &val : numbers)
{
 cout << "Enter an integer value: ";
 cin >> val;
}
```

### 7.5

## 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. For example, the following statement decrements the variable `count`, but does nothing to the value in `amount[count]`.

```
amount[count--];
```

To decrement the value stored in `amount[count]`, use the following statement:

```
amount[count]--;
```

Program 7-13 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 7-13

```
1 // This program stores, in an array, the hours worked by
2 // 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_EMPLOYEES = 5; // Number of employees
10 int hours[NUM_EMPLOYEES]; // Array to hold hours
11 double payrate; // Hourly pay rate
12 double grossPay; // To hold the gross pay
13
14 // Input the hours worked.
15 cout << "Enter the hours worked by ";
16 cout << NUM_EMPLOYEES << " employees who all\n";
17 cout << "earn the same hourly rate.\n";
18 for (int index = 0; index < NUM_EMPLOYEES; index++)
19 {
20 cout << "Employee #" << (index + 1) << ": ";
21 cin >> hours[index];
22 }
23
24 // Input the hourly rate for all employees.
25 cout << "Enter the hourly pay rate for all the employees: ";
26 cin >> payrate;
27
28 // Display each employee's gross pay.
29 cout << "Here is the gross pay for each employee:\n";
30 cout << fixed << showpoint << setprecision(2);
31 for (int index = 0; index < NUM_EMPLOYEES; index++)
32 {
33 grossPay = hours[index] * payrate;
34 cout << "Employee #" << (index + 1);
35 cout << ": $" << grossPay << endl;
36 }
37 return 0;
38 }
```

#### Program Output with Example Input Shown in Bold

Enter the hours worked by 5 employees who all  
earn the same hourly rate.

Employee #1: **5**

Employee #2: **10**

(program output continues)

**Program 7-13**

(continued)

```

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

```

The following statement in line 33 assigns the value of `hours[index]` times `payRate` to the `grossPay` variable:

```
grossPay = hours[index] * payRate;
```

Array elements may 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 the following statement sets up a `while` loop to iterate as long as `value[place]` does not equal 0:

```
while (value[place] != 0)
```

## Thou Shall Not Assign

The following code defines two integer arrays: `newValues` and `oldValues`. `newValues` is uninitialized, and `oldValues` is initialized with 10, 100, 200, and 300.

```

const int SIZE = 4;
int oldValues[SIZE] = {10, 100, 200, 300};
int newValues[SIZE];

```

At first glance, it might appear that the following statement assigns the contents of the array `oldValues` to `newValues`:

```
newValues = oldValues; // Wrong!
```

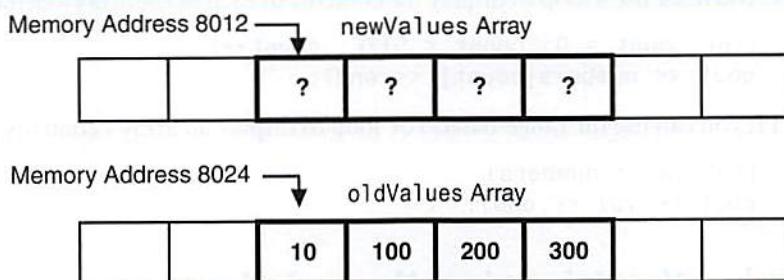
Unfortunately, this statement will not work. The only way to assign one array to another is to assign the individual elements in the arrays. Usually, this is best done with a loop, such as:

```

for (int count = 0; count < SIZE; count++)
 newValues[count] = oldValues[count];

```

The reason the assignment operator will not work with an entire array at once is complex, but important to understand. Anytime the name of an array is used without brackets and a subscript, *it is seen as the array's beginning memory address*. To illustrate this, consider the definition of the arrays `newValues` and `oldValues` above. Figure 7-12 depicts the two arrays in memory.

**Figure 7-12** The newValues and oldValues arrays

In the figure, `newValues` is shown starting at memory address 8012 and `oldValues` is shown starting at 8024. (Of course, these are just arbitrary addresses, picked for illustration purposes. In reality, the addresses would probably be different.) Table 7-2 shows various expressions that use the names of these arrays and their values.

**Table 7-2** Array Expressions

| Expression                | Value                                                  |
|---------------------------|--------------------------------------------------------|
| <code>oldValues[0]</code> | 10 (Contents of Element 0 of <code>oldValues</code> )  |
| <code>oldValues[1]</code> | 100 (Contents of Element 1 of <code>oldValues</code> ) |
| <code>oldValues[2]</code> | 200 (Contents of Element 2 of <code>oldValues</code> ) |
| <code>oldValues[3]</code> | 300 (Contents of Element 3 of <code>oldValues</code> ) |
| <code>newValues</code>    | 8012 (Memory Address of <code>newValues</code> )       |
| <code>oldValues</code>    | 8024 (Memory Address of <code>oldValues</code> )       |

Because the name of an array without the brackets and subscript stands for the array's starting memory address, the statement

```
newValues = oldValues;
```

is interpreted by C++ as

```
8012 = 8024;
```

The statement will not work, because you cannot change the starting memory address of an array.

## Printing the Contents of an Array

Suppose we have the following array definition:

```
const int SIZE = 5;
int numbers [SIZE] = {10, 20, 30, 40, 50};
```

You now know that an array's name is seen as the array's beginning memory address. This explains why the following statement cannot be used to display the contents of the `numbers` array.

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

When this statement executes, cout will display the array's memory address, not the array's contents. You must use a loop to display the contents of each of the array's elements, as follows:

```
for (int count = 0; count < SIZE; count++)
 cout << numbers[count] << endl;
```

11

In C++ 11, you can use the range-based for loop to display an array's contents, as shown here:

```
for (int val : numbers)
 cout << val << endl;
```

## 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 the following statements appear in a program, and values have been stored in the units array:

```
const int NUM_UNITS = 24;
int units[NUM_UNITS];
```

The following code uses a regular for loop to add 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.

```
int total = 0; // Initialize accumulator
for (int count = 0; count < NUM_UNITS; count++)
 total += units[count];
```

11

In C++ 11, you can use the range-based for loop, as shown below. When the code is finished, total will contain the sum of the units array's elements.

```
int total = 0; // Initialize accumulator
for (int val : units)
 total += val;
```



**NOTE:** The first statement in both of these code segments sets total 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.

## Getting 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 the following statements appear in a program, and values have been stored in the scores array:

```
const int NUM_SCORES = 10;
double scores[NUM_SCORES];
```

The following code calculates the average of the values in the scores array. When the code completes, the average will be stored 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 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 its iterations.

11

In C++ 11, you can use the range-based for loop, as shown below. When the code completes, the average will be stored in the average variable.

```
double total = 0; // Initialize accumulator
double average; // Will hold the average
for (int val : scores)
 total += val;
average = total / NUM_SCORES;
```

## 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 the following code exists in a program, and values have already been stored in the numbers array:

```
const int SIZE = 50;
int numbers[SIZE];
```

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 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.

## 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 `count` that will hold the number of items stored in the array.

```
const int SIZE = 100;
int numbers[SIZE];
int count = 0;
```

Each time we add an item to the array, we must increment `count`. The following code demonstrates this:

```
int num;
cout << "Enter a number or -1 to quit: ";
cin >> num;
while (num != -1 && count < SIZE)
{
 count++;
 numbers[count - 1] = num;
 cout << "Enter a number or -1 to quit: ";
 cin >> num;
}
```

Each iteration of this sentinel-controlled loop allows the user to enter a number to be stored in the array, or `-1` to quit. The `count` variable is incremented then used to calculate the subscript of the next available element in the array. When the user enters `-1`, or `count` exceeds 99, the loop stops. The following code displays all of the valid items in the array:

```
for (int index = 0; index < count; index++)
{
 cout << numbers[index] << endl;
}
```

Notice this code uses `count` to determine the maximum array subscript to use.

Program 7-14 shows how this technique can be used to read an unknown number of items from a file into an array. The program reads values from the file `numbers.txt`.

### Program 7-14

```
1 //This program reads data from a file into an array.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8 const int ARRAY_SIZE = 100; // Array size
9 int numbers[ARRAY_SIZE]; // Array with 100 elements
```

```
10 int count = 0; // Loop counter variable
11 ifstream inputFile; // Input file stream object
12
13 inputFile.open("numbers.txt"); // Open the file.
14
15 // Read the numbers from the file into the array.
16 // After this loop executes, the count variable will hold
17 // the number of values that were stored in the array.
18 while (count < ARRAY_SIZE && inputFile >> numbers[count])
19 count++;
20
21 // Close the file.
22 inputFile.close();
23
24 // Display the numbers read.
25 cout << "The numbers are: ";
26 for (int index = 0; index < count; index++)
27 cout << numbers[index] << " ";
28 cout << endl;
29 return 0;
30 }
```

### Program Output

The numbers are: 47 89 65 36 12 25 17 8 62 10 87 62

Look closer at the `while` loop that begins in line 18. It repeats as long as `count` is less than `ARRAY_SIZE` and the end of the file has not been encountered. The first part of the `while` loop's test expression, `count < ARRAY_SIZE`, prevents the loop from writing outside the array boundaries. Recall from Chapter 4 that the `&&` operator performs short-circuit evaluation, so the second part of the `while` loop's test expression, `inputFile >> values[count]`, will be executed only if `count` is less than `ARRAY_SIZE`.

## Comparing Arrays

We have already noted that you cannot simply assign one array to another array. You must assign each element of the first array to an element of the second array. In addition, you cannot use the `==` operator with the names of two arrays to determine whether the arrays are equal. For example, the following code appears to compare two arrays, but in reality it does not:

```
int firstArray[] = { 5, 10, 15, 20, 25 };
int secondArray[] = { 5, 10, 15, 20, 25 };
if (firstArray == secondArray) // 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 array names in this code will obviously have different memory addresses. Therefore, the result of the expression `firstArray == secondArray` is false, and the code reports that the arrays are not the same.

To compare the contents of two arrays, you must compare the elements of the two arrays. For example, look at the following code:

```

const int SIZE = 5;
int firstArray[SIZE] = { 5, 10, 15, 20, 25 };
int secondArray[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 (firstArray[count] != secondArray[count])
 arraysEqual = false;
 count++;
}

if (arraysEqual)
 cout << "The arrays are equal.\n";
else
 cout << "The arrays are not equal.\n";

```

This code determines whether `firstArray` and `secondArray` 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 set of corresponding elements in the arrays. When it finds two corresponding elements that have different values, the `arraysEqual` variable is set to `false`. After the loop finishes, an `if` statement examines the `arraysEqual` variable. If the variable is `true`, then the arrays are equal and a message indicating so is displayed. Otherwise, they are not equal, so a different message is displayed.

## 7.6

## Focus on Software Engineering: Using Parallel Arrays

**CONCEPT:** By using the same subscript, you can build relationships between data stored in two or more arrays.

Sometimes it's useful to store related data in two or more arrays. It's especially useful when the related data is of unlike types. For example, Program 7-15 is another variation of the payroll program. It uses two arrays: one to store the hours worked by each employee (as `ints`), and another to store each employee's hourly pay rate (as `doubles`).

### Program 7-15

```

1 // This program uses two parallel arrays: one for hours
2 // worked and one for pay rate.
3 #include <iostream>

```

```
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9 const int NUM_EMPLOYEES = 5; // Number of employees
10 int hours[NUM_EMPLOYEES]; // Holds hours worked
11 double payRate[NUM_EMPLOYEES]; // Holds pay rates
12
13 // Input the hours worked and the hourly pay rate.
14 cout << "Enter the hours worked by " << NUM_EMPLOYEES
15 << " employees and their\n"
16 << "hourly pay rates.\n";
17 for (int index = 0; index < NUM_EMPLOYEES; index++)
18 {
19 cout << "Hours worked by employee #" << (index+1) << ": ";
20 cin >> hours[index];
21 cout << "Hourly pay rate for employee #" << (index+1) << ": ";
22 cin >> payRate[index];
23 }
24
25 // Display each employee's gross pay.
26 cout << "Here is the gross pay for each employee:\n";
27 cout << fixed << showpoint << setprecision(2);
28 for (int index = 0; index < NUM_EMPLOYEES; index++)
29 {
30 double grossPay = hours[index] * payRate[index];
31 cout << "Employee #" << (index + 1)
32 << ": $" << grossPay << endl;
33 }
34 return 0;
35 }
```

### Program Output with Example Input Shown in Bold

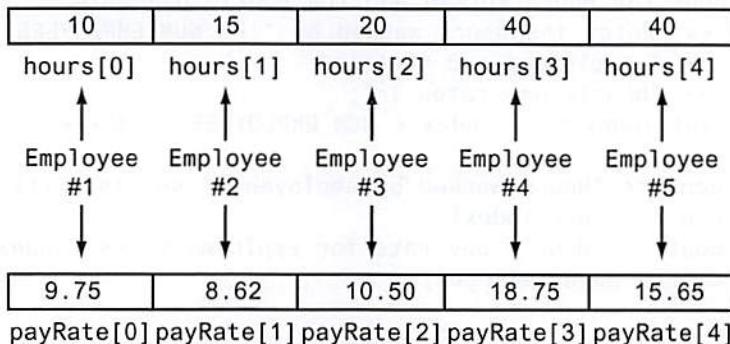
Enter the hours worked by 5 employees and their hourly pay rates.

```
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 the same subscript is used to access both arrays. That's because the data for one 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 7-13.

**Figure 7-13** Parallel arrays



### Checkpoint

- 7.10 Given the following array definition:

```
int values[] = {2, 6, 10, 14};
```

What does each of the following display?

- A) `cout << values[2];`
- B) `cout << ++values[0];`
- C) `cout << values[1]++;`
- D) `x = 2;  
cout << values[++x];`

- 7.11 Given the following array definition:

```
int nums[5] = {1, 2, 3};
```

What will the following statement display?

```
cout << nums[3];
```

- 7.12 What is the output of the following code? (You may need to use a calculator.)

```
double balance[5] = {100.0, 250.0, 325.0, 500.0, 1100.0};
const double INTRATE = 0.1;
```

```
cout << fixed << showpoint << setprecision(2);
for (int count = 0; count < 5; count++)
 cout << (balance[count] * INTRATE) << endl;
```

- 7.13 What is the output of the following code? (You may need to use a calculator.)

```
const int SIZE = 5;
int time[SIZE] = {1, 2, 3, 4, 5},
 speed[SIZE] = {18, 4, 27, 52, 100},
 dist[SIZE];
for (int count = 0; count < SIZE; count++)
 dist[count] = time[count] * speed[count];
for (int count = 0; count < SIZE; count++)
{
 cout << time[count] << " ";
 cout << speed[count] << " ";
 cout << dist[count] << endl;
}
```

## 7.7

## Arrays as Function Arguments

**CONCEPT:** To pass an array as an argument to a function, pass the name of the array.



Quite often, you'll want to write functions that process the data in arrays. For example, functions could 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 7-16 shows a loop that passes one element of the array numbers to the function showValue each time the loop iterates.

### Program 7-16

```
1 // This program demonstrates that an array element is 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 SIZE = 8;
11 int numbers[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
12
13 for (int index = 0; index < SIZE; index++)
14 showValue(numbers[index]);
15 return 0;
16 }
```

(program continues)

**Program 7-16** (continued)

```

17 //*****
18 // Definition of function showValue. *
19 // This function accepts an integer argument. *
20 // The value of the argument is displayed. *
21 //*****
22 //*****
23
24 void showValue(int num)
25 {
26 cout << num << " ";
27 }
```

**Program Output**

5 10 15 20 25 30 35 40

Each time `showValue` is called in line 14, a copy of an array element is passed into the parameter variable `num`. The `showValue` function simply displays the contents of `num` and doesn't work directly with the array element itself. (In other words, the array element is passed by value.)

If the function were written to accept the entire array as an argument, however, 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 array, not a single value.

```

void showValues(int nums[], int size)
{
 for (int index = 0; index < size; index++)
 cout << nums[index] << " ";
 cout << endl;
}
```

The reason there is no `size` declarator inside the brackets of `nums` is because `nums` is not actually an array. It's a special variable that can accept the address of an array. When an entire array is passed to a function, it is not passed by value, but passed by reference. 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. Program 7-17 shows the function `showValues` in use.



**NOTE:** Notice in the function prototype, empty brackets appear after the data type of the array parameter. This indicates that `showValues` accepts the address of an array of integers.

**Program 7-17**

```

1 // This program demonstrates an array being passed to a function.
2 #include <iostream>
3 using namespace std;
4
5 void showValues(int [], int); // Function prototype
6
7 int main()
8 {
9 const int ARRAY_SIZE = 8;
10 int numbers[ARRAY_SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
11
12 showValues(numbers, ARRAY_SIZE);
13 return 0;
14 }
15
16 //*****
17 // Definition of function showValue.
18 // This function accepts an array of integers and
19 // the array's size as its arguments. The contents
20 // of the array are displayed.
21 //*****
22
23 void showValues(int nums[], int size)
24 {
25 for (int index = 0; index < size; index++)
26 cout << nums[index] << " ";
27 cout << endl;
28 }
```

**Program Output**

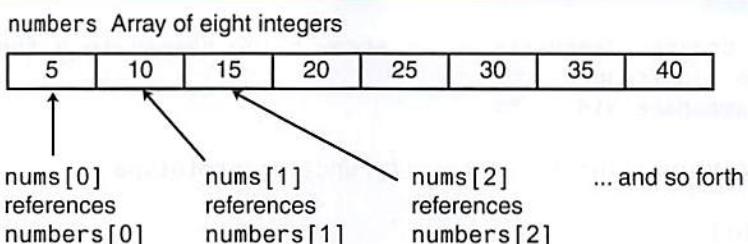
5 10 15 20 25 30 35 40

In Program 7-17, the function `showValues` is called in the following statement that appears in line 12:

```
showValues(numbers, ARRAY_SIZE);
```

The first argument is the name of the array. 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 `numbers` array is being passed as the first argument to the function. The second argument is the size of the array.

In the `showValues` function, the beginning address of the `numbers` array is copied into the `nums` parameter variable. The `nums` variable is then used to reference the `numbers` array. Figure 7-14 illustrates the relationship between the `numbers` array and the `nums` parameter variable. When the contents of `nums[0]` are displayed, it is actually the contents of `numbers[0]` that appear on the screen.

**Figure 7-14** Relationship between the numbers array and the nums parameter

**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 be used 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 7-18 uses the function to display the contents of two different arrays.

### Program 7-18

```

1 // This program demonstrates the showValues function being
2 // used to display the contents of two arrays.
3 #include <iostream>
4 using namespace std;
5
6 void showValues(int [], int); // Function prototype
7
8 int main()
9 {
10 const int SIZE1 = 8; // Size of set1 array
11 const int SIZE2 = 5; // Size of set2 array
12 int set1[SIZE1] = {5, 10, 15, 20, 25, 30, 35, 40};
13 int set2[SIZE2] = {2, 4, 6, 8, 10};
14
15 // Pass set1 to showValues.
16 showValues(set1, SIZE1);
17
18 // Pass set2 to showValues.
19 showValues(set2, SIZE2);
20 return 0;
21 }
22
23 //***** Definition of function showValues. *****
24 // This function accepts an array of integers and *
25 // the array's size as its arguments. The contents *
26 // of the array are displayed. *
27 //*****
```

```
30 void showValues(int nums[], int size)
31 {
32 for (int index = 0; index < size; index++)
33 cout << nums[index] << " ";
34 cout << endl;
35 }
```

### Program Output

```
5 10 15 20 25 30 35 40
2 4 6 8 10
```

Recall 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 with the array parameter are actually made on the original array used as the argument. The function `doubleArray` in Program 7-19 uses this capability to double the contents of each element in the array.

### Program 7-19

```
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 // Function prototypes
7 void doubleArray(int [], int);
8 void showValues(int [], int);
9
10 int main()
11 {
12 const int ARRAY_SIZE = 7;
13 int set[ARRAY_SIZE] = {1, 2, 3, 4, 5, 6, 7};
14
15 // Display the initial values.
16 cout << "The array's values are:\n";
17 showValues(set, ARRAY_SIZE);
18
19 // Double the values in the array.
20 doubleArray(set, ARRAY_SIZE);
21
22 // Display the resulting values.
23 cout << "After calling doubleArray the values are:\n";
24 showValues(set, ARRAY_SIZE);
25
26 return 0;
27 }
28
```

(program continues)

**Program 7-19**

(continued)

```

29 //*****
30 // Definition of function doubleArray *
31 // This function doubles the value of each element *
32 // in the array passed into nums. The value passed *
33 // into size is the number of elements in the array. *
34 //*****
35
36 void doubleArray(int nums[], int size)
37 {
38 for (int index = 0; index < size; index++)
39 nums[index] *= 2;
40 }
41
42 //*****
43 // Definition of function showValues. *
44 // This function accepts an array of integers and *
45 // the array's size as its arguments. The contents *
46 // of the array are displayed. *
47 //*****
48
49 void showValues(int nums[], int size)
50 {
51 for (int index = 0; index < size; index++)
52 cout << nums[index] << " ";
53 cout << endl;
54 }
```

**Program Output**

The array's values are:

1 2 3 4 5 6 7

After calling doubleArray the values are:

2 4 6 8 10 12 14

**Using const Array Parameters**

Sometimes you want a function to be able to modify the contents of an array that is passed to it as an argument, and sometimes you don't. You can prevent a function from making changes to an array argument by using the `const` key word in the parameter declaration. Here is an example of the `showValues` function, shown previously, rewritten with a `const` array parameter:

```

void showValues(const int nums[], int size)
{
 for (int index = 0; index < size; index++)
 cout << nums[index] << " ";
 cout << endl;
}
```

When an array parameter is declared as `const`, the function is not allowed to make changes to the array's contents. If a statement in the function attempts to modify the array, an error

will occur at compile time. As a precaution, you should always use `const` array parameters in any function that is not intended to modify its array argument. That way, the function will fail to compile if you inadvertently write code in it that modifies the array.

## Some Useful Array Functions

Section 7.5 introduced you to algorithms such as summing an array and finding the highest and lowest values in an array. Now that you know how to pass an array as an argument to a function, you can write general purpose functions that perform those operations. The following *In the Spotlight* section shows an example.

### In the Spotlight:

#### Processing an Array



Dr. LaClaire gives four exams during the semester in her chemistry class. At the end of the semester, she drops each student's lowest test score before averaging the scores. She has asked you to write a program that will read a student's four test scores as input, and calculate the average with the lowest score dropped. Here is the pseudocode algorithm that you developed:

*Read the student's four test scores.*

*Calculate the total of the scores.*

*Find the lowest score.*

*Subtract the lowest score from the total. This gives the adjusted total.*

*Divide the adjusted total by 3. This is the average.*

*Display the average.*

Program 7-20 shows the program, which is modularized. Rather than presenting the entire program at once, let's first examine the `main` function, then each additional function separately. Here is the first part of the program, including the `main` function:

#### Program 7-20 (main function)

```
1 // This program gets a series of test scores and
2 // calculates the average of the scores with the
3 // lowest score dropped.
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;
7
8 // Function prototypes
9 void getTestScores(double[], int);
10 double getTotal(const double[], int);
11 double getLowest(const double[], int);
12
```

(program continues)

**Program 7-20** (continued)

```

13 int main()
14 {
15 const int SIZE = 4; // Array size
16 double testScores[SIZE], // Array of test scores
17 total, // Total of the scores
18 lowestScore, // Lowest test score
19 average; // Average test score
20
21 // Set up numeric output formatting.
22 cout << fixed << showpoint << setprecision(1);
23
24 // Get the test scores from the user.
25 getTestScores(testScores, SIZE);
26
27 // Get the total of the test scores.
28 total = getTotal(testScores, SIZE);
29
30 // Get the lowest test score.
31 lowestScore = getLowest(testScores, SIZE);
32
33 // Subtract the lowest score from the total.
34 total -= lowestScore;
35
36 // Calculate the average. Divide by 3 because
37 // the lowest test score was dropped.
38 average = total / (SIZE - 1);
39
40 // Display the average.
41 cout << "The average with the lowest score "
42 << "dropped is " << average << ".\n";
43
44 return 0;
45 }
46

```

Lines 15 through 19 define the following items:

- `SIZE`—an `int` constant that is used as an array size declarator
- `testScores`—a `double` array to hold the test scores
- `total`—a `double` variable that will hold the test score totals
- `lowestScore`—a `double` variable that will hold the lowest test score
- `average`—a `double` variable that will hold the average of the test scores

Line 25 calls the `getTestScores` function, passing the `testScores` array and the value of the `SIZE` constant as arguments. The function gets the test scores from the user and stores them in the array.

Line 28 calls the `getTotal` function, passing the `testScores` array and the value of the `SIZE` constant as arguments. The function returns the total of the values in the array. This value is assigned to the `total` variable.

Line 31 calls the `getLowest` function, passing the `testScores` array and the value of the `SIZE` constant as arguments. The function returns the lowest value in the array. This value is assigned to the `lowestScore` variable.

Line 34 subtracts the lowest test score from the `total` variable. Then, line 38 calculates the average by dividing `total` by `SIZE - 1`. (The program divides by `SIZE - 1` because the lowest test score was dropped.) Lines 41 and 42 display the average.

The `getTestScores` function appears next, as shown here:

**Program 7-20 (getTestScores function)**

```
47 //*****
48 // The getTestScores function accepts an array and its size *
49 // as arguments. It prompts the user to enter test scores, *
50 // which are stored in the array. *
51 //*****
52
53 void getTestScores(double scores[], int size)
54 {
55 // Loop counter
56 int index;
57
58 // Get each test score.
59 for(index = 0; index <= size - 1; index++)
60 {
61 cout << "Enter test score number "
62 << (index + 1) << ": ";
63 cin >> scores[index];
64 }
65 }
66
```

The `getTestScores` function has two parameters:

- `scores[]`—A `double` array
- `size`—An `int` specifying the size of the array that is passed into the `scores[]` parameter

The purpose of this function is to get a student's test scores from the user and store them in the array that is passed as an argument into the `scores[]` parameter.

The `getTotal` function appears next, as shown here:

**Program 7-20 (getTotal function)**

```
67 //*****
68 // The getTotal function accepts a double array *
69 // and its size as arguments. The sum of the array's *
70 // elements is returned as a double. *
71 //*****
72
```

(program continues)

**Program 7-20** (continued)

```

73 double getTotal(const double numbers[], int size)
74 {
75 double total = 0; // Accumulator
76
77 // Add each element to total.
78 for (int count = 0; count < size; count++)
79 total += numbers[count];
80
81 // Return the total.
82 return total;
83 }
84

```

The `getTotal` function has two parameters:

- `numbers[]`—A `const double` array
- `size`—An `int` specifying the size of the array that is passed into the `numbers[]` parameter

This function returns the total of the values in the array that is passed as an argument into the `numbers[]` parameter.

The `getLowest` function appears next, as shown here:

**Program 7-20** (getLowest function)

```

85 //*****
86 // The getLowest function accepts a double array and *
87 // its size as arguments. The lowest value in the *
88 // array is returned as a double. *
89 //*****
90
91 double getLowest(const double numbers[], int size)
92 {
93 double lowest; // To hold the lowest value
94
95 // Get the first array's first element.
96 lowest = numbers[0];
97
98 // Step through the rest of the array. When a
99 // value less than lowest is found, assign it
100 // to lowest.
101 for (int count = 1; count < size; count++)
102 {
103 if (numbers[count] < lowest)
104 lowest = numbers[count];
105 }
106
107 // Return the lowest value.
108 return lowest;
109 }

```

The `getLowest` function has two parameters:

- `numbers[]`—A `const double` array
- `size`—An `int` specifying the size of the array that is passed into the `numbers[]` parameter

This function returns the lowest value in the array that is passed as an argument into the `numbers[]` parameter. Here is an example of the program's output:

### Program 7-20

#### Program Output with Example Input Shown in Bold

```
Enter test score number 1: 92 Enter
Enter test score number 2: 67 Enter
Enter test score number 3: 75 Enter
Enter test score number 4: 88 Enter
The average with the lowest score dropped is 85.0.
```



### Checkpoint

7.14 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;
```

7.15 When an array name is passed to a function, what is actually being passed?

7.16 When used as function arguments, are arrays passed by value?

7.17 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(const char [], int);

int main ()
{
 const int SIZE = 8;
 char prodCode[SIZE] = {'0', '0', '0', '0', '0', '0', '0', '0'};
 fillArray(prodCode, SIZE);
 showArray(prodCode, SIZE);
 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(const char codes[], int size)
{
 for (int k = 0; k < size; k++)
 cout << codes[k];
 cout << endl;
}

```

- 7.18 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, is to calculate and return the average of the numbers entered.

```

#include <iostream>
using namespace std;

// Write your 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 function avgArray here.
//

```

**7.8****Two-Dimensional Arrays**

**CONCEPT:** A two-dimensional array is like several identical arrays put together. It is useful for storing multiple sets of data.

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