GLOBAL EDITION

Starting Out With C++

From Control Structures through Objects

EIGHTH EDITION

Tony Gaddis

ALWAYS LEARNING

PEARSON

ONLINE ACCESS

Thank you for purchasing a new copy of ***Starting Out with C++ from Control Structures through Objects,*** **Eighth Edition**. Your textbook includes one year of prepaid access to the book's Companion Website. This prepaid subscription provides you with full access to the following student support areas:

- VideoNotes
- Online Appendices
- Source Code
- Case Studies
- Software Development Project: Serendipity Booksellers

Use a coin to scratch off the coating and reveal your student access code.
Do not use a knife or other sharp object as it may damage the code.

To access the ***Starting Out with C++ from Control Structures through Objects,*** **Eighth Edition*,*** Companion Website for the first time, you will need to register online using a computer with an Internet connection and a web browser. The process takes just a couple of minutes and only needs to be completed once.

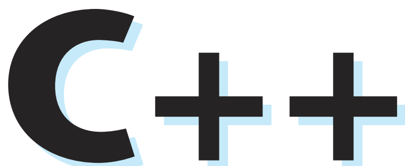
1. Go to **www.pearsonglobaleditions.com/gaddis**
2. Click on **Companion Website**.
3. Click on the **Register** button.
4. On the registration page, enter your student access code* found beneath the scratch-off panel. Do not type the dashes. You can use lower- or uppercase.
5. Follow the on-screen instructions. If you need help at any time during the online registration process, simply click the **Need Help?** icon.
6. Once your personal Login Name and Password are confirmed, you can begin using the ***Starting Out with C++ from Control Structures through Objects*** Companion Website!

To log in after you have registered:

You only need to register for this Companion Website once. After that, you can log in any time at **www.pearsonglobaleditions.com/gaddis** by providing your Login Name and Password when prompted.

*Important: The access code can only be used once. This subscription is valid for one year upon activation and is not transferable. If this access code has already been revealed, it may no longer be valid.

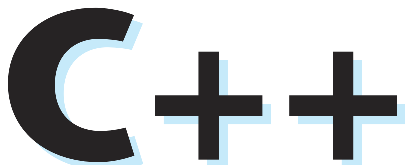
STARTING OUT WITH



From Control Structures through Objects

EIGHTH EDITION

STARTING OUT WITH



From Control Structures through Objects

EIGHTH EDITION
GLOBAL EDITION

Tony Gaddis
Haywood Community College

Global Edition contributions by
Moumita Mitra Manna
Bangabasi College

PEARSON

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

Editorial Director: Marcia Horton
Acquisitions Editor: Matt Goldstein
Program Manager: Kayla Smith-Tarbox
Director of Marketing: Christy Lesko
Marketing Coordinator: Kathryn Ferranti
Marketing Assistant: Jon Bryant
Senior Managing Editor: Scott Disanno
Senior Project Manager: Marilyn Lloyd
Head, Learning Asset Acquisitions, Global Edition: Laura Dent
Acquisition Editor, Global Edition: Aditee Agarwal
Project Editor, Global Edition: Anuprova Dey Chowdhuri
Operations Supervisor: Vincent Scelta
Operations Specialist: Linda Sager
Art Director, Cover: Jayne Conte
Text Designer: Joyce Cosentino Wells
Cover Designer: Lumina Datamatics Ltd.
Cover Image: Anthony Ricci/Shutterstock
Manager, Visual Research: Karen Sanatar
Permissions Supervisor: Michael Joyce
Permission Administrator: Jenell Forschler
Media Project Manager: Renata Butera
Full-Service Project Management: Jogender Taneja/iEnergizer Aptara®, Inc.
Composition: iEnergizer Aptara®, Inc.
Cover Printer/Binder: Courier Kendallville

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com

© Pearson Education Limited 2015

The rights of Tony Gaddis to be identified as the author of this work have been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

Authorized adaptation from the United States edition, entitled Starting Out With C++: From Control Structures through Objects, 8th edition, ISBN 978-0-13-376939-5, by Tony Gaddis, published by Pearson Education © 2015.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmittedin any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a license permitting restricted copying in the
United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS.

All trademarks used herein are the property of their respective owners.The use of any trademark in this text does not vest in the author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation with or endorsement of this book by such owners.

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

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

ISBN 10: 129206997X
ISBN 13: 978-1-29-206997-5

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

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

Typeset in 9 Sabon LT Std by Aptara®, Inc.

Printed and bound by Courier Kendallville.

The publisher's policy is to use paper manufactured from sustainable forests.

Contents at a Glance

Preface 15

CHAPTER 1 Introduction to Computers and Programming 31
CHAPTER 2 Introduction to C++ 57
CHAPTER 3 Expressions and Interactivity 113
CHAPTER 4 Making Decisions 179
CHAPTER 5 Loops and Files 257
CHAPTER 6 Functions 329
CHAPTER 7 Arrays 405
CHAPTER 8 Searching and Sorting Arrays 487
CHAPTER 9 Pointers 525
CHAPTER 10 Characters, C-Strings, and More About the `string` Class 577
CHAPTER 11 Structured Data 629
CHAPTER 12 Advanced File Operations 687
CHAPTER 13 Introduction to Classes 741
CHAPTER 14 More About Classes 841
CHAPTER 15 Inheritance, Polymorphism, and Virtual Functions 921
CHAPTER 16 Exceptions, Templates, and the Standard Template Library (STL) 1001
CHAPTER 17 Linked Lists 1055
CHAPTER 18 Stacks and Queues 1093
CHAPTER 19 Recursion 1151
CHAPTER 20 Binary Trees 1185

Appendix A: Getting Started with Alice 2 1215
Appendix B: The ASCII Character Set 1241
Appendix C: Operator Precedence and Associativity 1243
C++ Quick Reference 1245

Index 1247

Credit 1267

Online The following appendices are available at www.pearsonglobaleditions.com/gaddis.

Appendix D: Introduction to Flowcharting

Appendix E: Using UML in Class Design

Appendix F: Namespaces

Appendix G: Passing Command Line Arguments

Appendix H: Header File and Library Function Reference

Appendix I: Binary Numbers and Bitwise Operations

Appendix J: Multi-Source File Programs

Appendix K: Stream Member Functions for Formatting

Appendix L: Answers to Checkpoints

Appendix M: Solutions to Odd-Numbered Review Questions

Contents

Preface 15

CHAPTER 1 Introduction to Computers and Programming 31

1.1 Why Program? 31
1.2 Computer Systems: Hardware and Software 32
1.3 Programs and Programming Languages 38
1.4 What Is a Program Made of? 44
1.5 Input, Processing, and Output 47
1.6 The Programming Process 48
1.7 Procedural and Object-Oriented Programming 52

CHAPTER 2 Introduction to C++ 57

2.1 The Parts of a C++ Program 57
2.2 The `cout` Object 61
2.3 The `#include` Directive 66
2.4 Variables and Literals 67
2.5 Identifiers 71
2.6 Integer Data Types 72
2.7 The `char` Data Type 78
2.8 The C++ `string` Class 82
2.9 Floating-Point Data Types 84
2.10 The `bool` Data Type 87
2.11 Determining the Size of a Data Type 88
2.12 Variable Assignments and Initialization 89
2.13 Scope 91
2.14 Arithmetic Operators 91
2.15 Comments 99
2.16 Named Constants 101
2.17 Programming Style 103

CHAPTER 3 Expressions and Interactivity 113

3.1 The `cin` Object 113
3.2 Mathematical Expressions 119
3.3 When You Mix Apples and Oranges: Type Conversion 128
3.4 Overflow and Underflow 130
3.5 Type Casting 131
3.6 Multiple Assignment and Combined Assignment 134
3.7 Formatting Output 138
3.8 Working with Characters and `string` Objects 148
3.9 More Mathematical Library Functions 154
3.10 Focus on Debugging: Hand Tracing a Program 160
3.11 Focus on Problem Solving: A Case Study 162

CHAPTER 4 Making Decisions 179

4.1 Relational Operators 179
4.2 The `if` Statement 184
4.3 Expanding the `if` Statement 192
4.4 The `if/else` Statement 196
4.5 Nested `if` Statements 199
4.6 The `if/else if` Statement 206
4.7 Flags 211
4.8 Logical Operators 212
4.9 Checking Numeric Ranges with Logical Operators 219
4.10 Menus 220
4.11 Focus on Software Engineering: Validating User Input 223
4.12 Comparing Characters and Strings 225
4.13 The Conditional Operator 229
4.14 The `switch` Statement 232
4.15 More About Blocks and Variable Scope 241

CHAPTER 5 Loops and Files 257

5.1 The Increment and Decrement Operators 257
5.2 Introduction to Loops: The `while` Loop 262
5.3 Using the `while` Loop for Input Validation 269
5.4 Counters 271
5.5 The `do-while` Loop 272
5.6 The `for` Loop 277
5.7 Keeping a Running Total 287
5.8 Sentinels 290
5.9 Focus on Software Engineering: Deciding Which Loop to Use 291
5.10 Nested Loops 292
5.11 Using Files for Data Storage 295
5.12 Optional Topics: Breaking and Continuing a Loop 314

CHAPTER 6 Functions 329

6.1 Focus on Software Engineering: Modular Programming 329
6.2 Defining and Calling Functions 330
6.3 Function Prototypes 339
6.4 Sending Data into a Function 341

6.5 Passing Data by Value 346
6.6 Focus on Software Engineering: Using Functions in a Menu-Driven Program 348
6.7 The `return` Statement 352
6.8 Returning a Value from a Function 354
6.9 Returning a Boolean Value 362
6.10 Local and Global Variables 364
6.11 Static Local Variables 372
6.12 Default Arguments 375
6.13 Using Reference Variables as Parameters 378
6.14 Overloading Functions 384
6.15 The `exit()` Function 388
6.16 Stubs and Drivers 391

CHAPTER 7 Arrays 405

7.1 Arrays Hold Multiple Values 405
7.2 Accessing Array Elements 407
7.3 No Bounds Checking in C++ 414
7.4 Array Initialization 417
7.5 The Range-Based `for` Loop 422
7.6 Processing Array Contents 426
7.7 Focus on Software Engineering: Using Parallel Arrays 434
7.8 Arrays as Function Arguments 437
7.9 Two-Dimensional Arrays 448
7.10 Arrays with Three or More Dimensions 455
7.11 Focus on Problem Solving and Program Design: A Case Study 457
7.12 If You Plan to Continue in Computer Science: Introduction to the STL `vector` 459

CHAPTER 8 Searching and Sorting Arrays 487

8.1 Focus on Software Engineering: Introduction to Search Algorithms 487
8.2 Focus on Problem Solving and Program Design: A Case Study 493
8.3 Focus on Software Engineering: Introduction to Sorting Algorithms 500
8.4 Focus on Problem Solving and Program Design: A Case Study 507
8.5 If You Plan to Continue in Computer Science: Sorting and Searching `vector`s 515

CHAPTER 9 Pointers 525

9.1 Getting the Address of a Variable 525
9.2 Pointer Variables 527
9.3 The Relationship Between Arrays and Pointers 534
9.4 Pointer Arithmetic 538
9.5 Initializing Pointers 540
9.6 Comparing Pointers 541
9.7 Pointers as Function Parameters 543
9.8 Focus on Software Engineering: Dynamic Memory Allocation 552
9.9 Focus on Software Engineering: Returning Pointers from Functions 556
9.10 Using Smart Pointers to Avoid Memory Leaks 563
9.11 Focus on Problem Solving and Program Design: A Case Study 566

CHAPTER 10 Characters, C-Strings, and More About the `string` Class 577

10.1 Character Testing 577
10.2 Character Case Conversion 581
10.3 C-Strings 584
10.4 Library Functions for Working with C-Strings 588
10.5 C-String/Numeric Conversion Functions 599
10.6 Focus on Software Engineering: Writing Your Own C-String-Handling Functions 605
10.7 More About the C++ `string` Class 611
10.8 Focus on Problem Solving and Program Design: A Case Study 620

CHAPTER 11 Structured Data 629

11.1 Abstract Data Types 629
11.2 Focus on Software Engineering: Combining Data into Structures 631
11.3 Accessing Structure Members 634
11.4 Initializing a Structure 638
11.5 Arrays of Structures 641
11.6 Focus on Software Engineering: Nested Structures 643
11.7 Structures as Function Arguments 647
11.8 Returning a Structure from a Function 650
11.9 Pointers to Structures 653
11.10 Focus on Software Engineering: When to Use `.`, When to Use `->`, and When to Use `*` 656
11.11 Unions 658
11.12 Enumerated Data Types 662

CHAPTER 12 Advanced File Operations 687

12.1 File Operations 687
12.2 File Output Formatting 693
12.3 Passing File Stream Objects to Functions 695
12.4 More Detailed Error Testing 697
12.5 Member Functions for Reading and Writing Files 700
12.6 Focus on Software Engineering: Working with Multiple Files 708
12.7 Binary Files 710
12.8 Creating Records with Structures 715
12.9 Random-Access Files 719
12.10 Opening a File for Both Input and Output 727

CHAPTER 13 Introduction to Classes 741

13.1 Procedural and Object-Oriented Programming 741
13.2 Introduction to Classes 748
13.3 Defining an Instance of a Class 753
13.4 Why Have Private Members? 766
13.5 Focus on Software Engineering: Separating Class Specification from Implementation 767
13.6 Inline Member Functions 773
13.7 Constructors 776
13.8 Passing Arguments to Constructors 780

13.9 Destructors 788
13.10 Overloading Constructors 792
13.11 Private Member Functions 795
13.12 Arrays of Objects 797
13.13 Focus on Problem Solving and Program Design: An OOP Case Study 801
13.14 Focus on Object-Oriented Programming: Simulating Dice with Objects 808
13.15 Focus on Object-Oriented Programming: Creating an Abstract Array Data Type 812
13.16 Focus on Object-Oriented Design: The Unified Modeling Language (UML) 815
13.17 Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities 818

CHAPTER 14 More About Classes 841

14.1 Instance and Static Members 841
14.2 Friends of Classes 849
14.3 Memberwise Assignment 854
14.4 Copy Constructors 855
14.5 Operator Overloading 861
14.6 Object Conversion 888
14.7 Aggregation 890
14.8 Focus on Object-Oriented Design: Class Collaborations 895
14.9 Focus on Object-Oriented Programming: Simulating the Game of Cho-Han 899

CHAPTER 15 Inheritance, Polymorphism, and Virtual Functions 921

15.1 What Is Inheritance? 921
15.2 Protected Members and Class Access 930
15.3 Constructors and Destructors in Base and Derived Classes 936
15.4 Redefining Base Class Functions 948
15.5 Class Hierarchies 953
15.6 Polymorphism and Virtual Member Functions 959
15.7 Abstract Base Classes and Pure Virtual Functions 975
15.8 Multiple Inheritance 982

CHAPTER 16 Exceptions, Templates, and the Standard Template Library (STL) 1001

16.1 Exceptions 1001
16.2 Function Templates 1020
16.3 Focus on Software Engineering: Where to Start When Defining Templates 1026
16.4 Class Templates 1026
16.5 Introduction to the Standard Template Library (STL) 1035

CHAPTER 17 Linked Lists 1055

17.1 Introduction to the Linked List ADT 1055
17.2 Linked List Operations 1057
17.3 A Linked List Template 1073
17.4 Variations of the Linked List 1085
17.5 The STL `list` Container 1086

CHAPTER 18 Stacks and Queues 1093

18.1 Introduction to the Stack ADT 1093
18.2 Dynamic Stacks 1110
18.3 The STL `stack` Container 1121
18.4 Introduction to the Queue ADT 1123
18.5 Dynamic Queues 1135
18.6 The STL `deque` and `queue` Containers 1142

CHAPTER 19 Recursion 1151

19.1 Introduction to Recursion 1151
19.2 Solving Problems with Recursion 1155
19.3 Focus on Problem Solving and Program Design: The Recursive `gcd` Function 1163
19.4 Focus on Problem Solving and Program Design: Solving Recursively Defined Problems 1164
19.5 Focus on Problem Solving and Program Design: Recursive Linked List Operations 1165
19.6 Focus on Problem Solving and Program Design: A Recursive Binary Search Function 1169
19.7 The Towers of Hanoi 1171
19.8 Focus on Problem Solving and Program Design: The QuickSort Algorithm 1174
19.9 Exhaustive Algorithms 1178
19.10 Focus on Software Engineering: Recursion vs. Iteration 1181

CHAPTER 20 Binary Trees 1185

20.1 Definition and Applications of Binary Trees 1185
20.2 Binary Search Tree Operations 1188
20.3 Template Considerations for Binary Search Trees 1205

Appendix A: Getting Started with Alice 2 1215

Appendix B: The ASCII Character Set 1241

Appendix C: Operator Precedence and Associativity 1243

C++ Quick Reference 1245

Index 1247

Credit 1267

Online The following appendices are available at www.pearsonglobaleditions.com/gaddis.

Appendix D: Introduction to Flowcharting

Appendix E: Using UML in Class Design

Appendix F: Namespaces

Appendix G: Passing Command Line Arguments

Appendix H: Header File and Library Function Reference

Appendix I: Binary Numbers and Bitwise Operations

Appendix J: Multi-Source File Programs

Appendix K: Stream Member Functions for Formatting

Appendix L: Answers to Checkpoints

Appendix M: Solutions to Odd-Numbered Review Questions

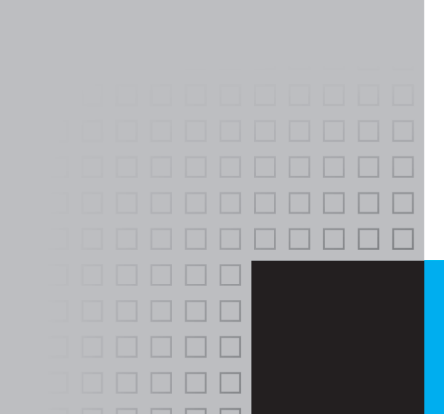
LOCATION OF VIDEONOTES IN THE TEXT

Chapter 1	Introduction to Flowcharting, p. 50 Designing a Program with Pseudocode, p. 50 Designing the Account Balance Program, p. 55 Predicting the Result of Problem 33, p. 56
Chapter 2	Using `cout`, p. 61 Variabe Definitions, p. 67 Assignment Statements and Simple Math Expressions, p. 92 Solving the Restaurant Bill Problem, p. 110
Chapter 3	Reading Input with `cin`, p. 113 Formatting Numbers with `setprecision`, p. 141 Solving the Stadium Seating Problem, p. 172
Chapter 4	The `if` Statement, p. 184 The `if/else` statement, p. 196 The `if/else if` Statement, p. 206 Solving the Time Calculator Problem, p. 251
Chapter 5	The `while` Loop, p. 262 The `for` Loop, p. 277 Reading Data from a File, p. 304 Solving the Calories Burned Problem, p. 323
Chapter 6	Functions and Arguments, p. 341 Value-Returnlng Functions, p. 354 Solving the Markup Problem, p. 396
Chapter 7	Accessing Array Elements With a Loop, p. 410 Passing an Array to a Function, p. 437 Solving the Chips and Salsa Problem, p. 478
Chapter 8	The Binary Search, p. 490 The Selection Sort, p. 504 Solving the Charge Account Validation Modification Problem, p. 522
Chapter 9	Dynamically Allocating an Array, p. 553 Solving the Pointer Rewrite Problem, p. 575
Chapter 10	Writing a C-String-Handling Function, p. 605 More About the `string` Class, p. 611 Solving the Backward String Problem, p. 624

(continued on the next page)

LOCATION OF VIDEONOTES IN THE TEXT *(continued)*

Chapter	VideoNotes
Chapter 11	Creating a Structure, p. 631 Passing a Structure to a Function, p. 647 Solving the Weather Statistics Problem, p. 682
Chapter 12	Passing File Stream Objects to Functions, p. 695 Working with Multiple Files, p. 708 Solving the File Encryption Filter Problem, p. 738
Chapter 13	Writing a Class, p. 748 Defining an Instance of a Class, p. 753 Solving the `Employee` Class Problem, p. 832
Chapter 14	Operator Overloading, p. 861 Class Aggregation, p. 890 Solving the `NumDays` Problem, p. 915
Chapter 15	Redefining a Base Class Function in a Derived Class, p. 948 Polymorphism, p. 959 Solving the `Employee` and `Production-Worker` Classes Problem, p. 993
Chapter 16	Throwing an Exception, p. 1002 Handling an Exception, p. 1002 Writing a Function Template, p. 1020 Storing Objects in a `vector`, p. 1040 Solving the Exception Project Problem, p. 1054
Chapter 17	Appending a Node to a Linked List, p. 1058 Inserting a Node in a Linked List, p. 1065 Deleting a Node from a Linked List, p. 1069 Solving the Member Insertion by Position Problem, p. 1091
Chapter 18	Storing Objects in an STL `stack`, p. 1121 Storing Objects in an STL `queue`, p. 1144 Solving the File Compare Problem, p. 1149
Chapter 19	Reducing a Problem with Recursion, p. 1156 Solving the Recursive Multiplication Problem, p. 1183
Chapter 20	Inserting a Node in a Binary Tree, p. 1190 Deleting a Node from a Binary Tree, p. 1196 Solving the Node Counter Problem, p. 1212



Preface

Welcome to *Starting Out with C++: From Control Structures through Objects, 8th 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 Eighth Edition

C++11 is the latest standard version of the C++ language. In previous years, while the standard was being developed, it was known as C++0x. In August 2011, it was approved by the International Standards Organization (ISO), and the name of the standard was officially changed to C++11. Most of the popular compilers now support the C++11 standard.

The new C++11 standard was the primary motivation behind this edition. Although this edition introduces many of the new language features, a C++11 compiler is not strictly required to use the book. As you progress through the book, you will see C++11 icons in the margins, next to the new features that are introduced. Programs appearing in sections that are not marked with this icon will still compile using an older compiler.

Here is a summary of the new C++11 topics that are introduced in this edition:

- The `auto` key word is introduced as a way to simplify complex variable definitions. The `auto` key word causes the compiler to infer a variable's data type from its initialization value.
- The `long long int` and `unsigned long long int` data types, and the `LL` literal suffix are introduced.
- Chapter 5 shows how to pass a `string` object directly to a file stream object's `open` member function, without the need to call the `c_str()` member function. (A discussion of the `c_str()`function still exists for anyone using a legacy compiler.)

- The range-based `for` loop is introduced in Chapter 7. This new looping mechanism automatically iterates over each element of an array, `vector`, or other collection, without the need of a counter variable or a subscript.
- Chapter 7 shows how a `vector` can be initialized with an initialization list.
- The `nullptr` key word is introduced as the standard way of representing a null pointer.
- Smart pointers are introduced in Chapter 9, with an example of dynamic memory allocation using `unique_ptr`.
- Chapter 10 discusses the new, overloaded `to_string` functions for converting numeric values to `string` objects.
- The `string` class's new `back()` and `front()` member functions are included in Chapter 10's overview of the `string` class.
- Strongly typed `enums` are discussed in Chapter 11.
- Chapter 13 shows how to use the smart pointer `unique_ptr` to dynamically allocate an object.
- Chapter 15 discusses the `override` key word and demonstrates how it can help prevent subtle overriding errors. The `final` key word is discussed as a way of preventing a virtual member function from being overridden.

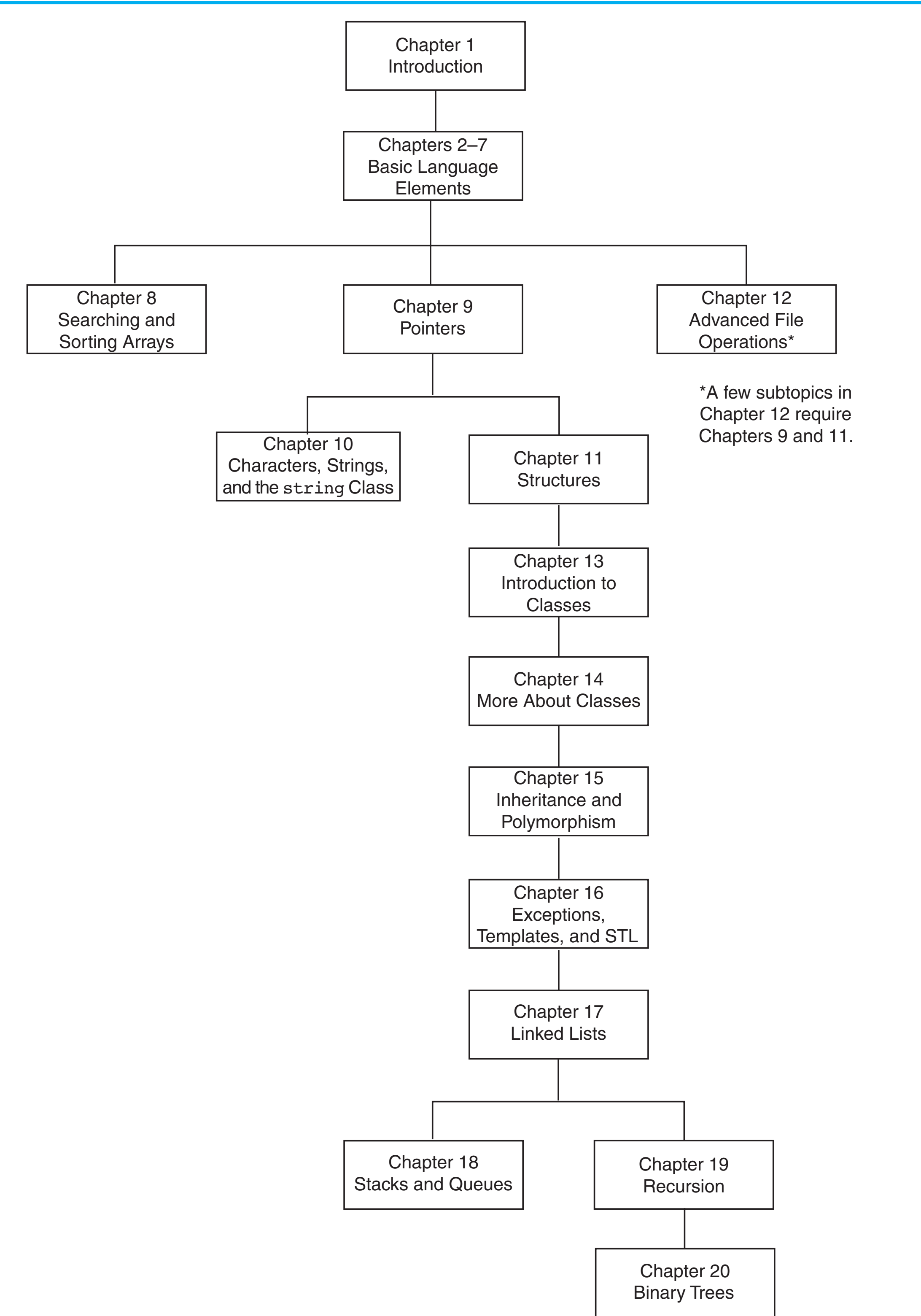
In addition to the C++11 topics, the following general improvements were made:

- Several new programming problems have been added to the text, and many of the existing programming problems have been modified to make them unique from previous editions.
- The discussion of early, historic computers in Chapter 1 is expanded.
- The discussion of literal values in Chapter 2 is improved.
- The introduction of the `char` data type in Chapter 2 is reorganized to use character literals in variable assignments before using ASCII values in variable assignments.
- The discussion of random numbers in Chapter 3 is expanded and improved, with the addition of a new *In the Spotlight* section.
- A new *Focus on Object-Oriented Programming* section has been added to Chapter 13, showing how to write a class that simulates dice.
- A new *Focus on Object-Oriented Programming* section has been added to Chapter 14, showing an object-oriented program that simulates the game of Cho-Han. The program uses objects for the dealer, two players, and a pair of dice.

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.

Figure P-1



Chapter 1 covers fundamental hardware, software, and programming concepts. You may choose to skip this chapter if the class has already mastered 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 either Chapter 9 or Chapter 12. (If you jump to Chapter 12 at this point, you will need to postpone sections 12.7, 12.8, and 12.10 until Chapters 9 and 11 have been covered.)

After Chapter 9 has been covered, either of Chapters 10 or 11 may be covered. After Chapter 11, you may cover Chapters 13 through 17 in sequence. Next you can proceed to either Chapter 18 or Chapter 19. Finally, Chapter 20 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.

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. An optional section explains the difference between ANSI standard and pre-standard C++ programs.

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

In this chapter the student learns to create and work with single and multidimensional 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: Sorting and Searching 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, Templates, and the Standard Template Library (STL)

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. Finally, the student is introduced to the containers, iterators, and algorithms offered by the Standard Template Library (STL).

Chapter 17: Linked Lists

This chapter introduces concepts and techniques needed to work with lists. A linked list ADT is developed and the student is taught to code operations such as creating a linked list, appending a node, traversing the list, searching for a node, inserting a node, deleting a node, and destroying a list. A linked list class template is also demonstrated.

Chapter 18: Stacks and Queues

In this chapter the student learns to create and use static and dynamic stacks and queues. The operations of stacks and queues are defined, and templates for each ADT are demonstrated.

Chapter 19: 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 20: 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: Getting Started with Alice

This appendix gives a quick introduction to Alice. Alice is free software that can be used to teach fundamental programming concepts using 3D graphics.

Appendix B: ASCII Character Set

A list of the ASCII and Extended ASCII characters and their codes.

Appendix C: Operator Precedence and Associativity

A chart showing the C++ operators and their precedence.

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

Appendix D: Introduction to Flowcharting

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

Appendix E: 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 F: Namespaces

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

Appendix G: 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 H: Header File and Library Function Reference

This appendix provides a reference for the C++ library functions and header files discussed in the book.

Appendix I: Binary Numbers and Bitwise Operations

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

Appendix J: 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 K: Stream Member Functions for Formatting

Covers stream member functions for formatting such as `setf`.

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: Solutions to Odd-Numbered Review Questions

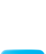
Another tool that students can use to gauge their progress.

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.pearsonglobaleditions.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 Companion Web site at www.pearsonglobaleditions.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 Companion Web site at www.pearsonglobaleditions.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 Companion Web site at www.pearsonglobaleditions.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
For easy access, a quick reference guide to the C++ language is printed on the last two pages of Appendix C in the book.

C++11
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 Gaddis Series Companion Web site at www.pearsonglobaleditions.com/gaddis:

- The source code for each example program in the book
- Access to the book's companion 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

Integrated Development Environment (IDE) Resource Kits

Professors who adopt this text can order it for students with a kit containing five popular C++ IDEs (Microsoft® Visual Studio Express Edition, Dev C++, NetBeans, Eclipse, and CodeLite) and access to a Web site containing written and video tutorials for getting started in each IDE. For ordering information, please contact your campus Pearson Education representative or visit www.pearsonglobaleditions.com/gaddis.

Instructor Resources

The following supplements are available to qualified instructors only:

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

Visit the Pearson Instructor Resource Center (`www.pearsonglobaleditions.com/gaddis`) 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:

`www.pearsonglobaleditions.com/gaddis`

Get this book the way you want it!

This book is part of Pearson Education's custom database for Computer Science textbooks. Use our online PubSelect system to select just the chapters you need from this,

and other, Pearson Education CS textbooks. You can edit the sequence to exactly match your course organization and teaching approach. Visit `www.pearsoncustom.com/cs` for details.

Which Gaddis C++ book is right for you?

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

- *Starting Out with C++: 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
LATE INTRODUCTION OF OBJECTS Classes are introduced in Chapter 13 of the standard text and Chapter 11 of the brief text, after control structures, functions, arrays, and pointers. Advanced OOP topics, such as inheritance and polymorphism, are covered in the following two chapters.	**EARLIER INTRODUCTION OF OBJECTS** Classes are introduced in Chapter 7, after control structures and functions, but before arrays and pointers. Their use is then integrated into the remainder of the text. Advanced OOP topics, such as inheritance and polymorphism, are covered in Chapters 11 and 15.
INTRODUCTION OF DATA STRUCTURES AND RECURSION Linked lists, stacks and queues, and binary trees are introduced in the final chapters of the standard text. Recursion is covered after stacks and queues, but before binary trees. These topics are not covered in the brief text, though it does have appendices dealing with linked lists and recursion.	**INTRODUCTION OF DATA STRUCTURES AND RECURSION** Linked lists, stacks and queues, and binary trees are introduced in the final chapters of the text, after the chapter on recursion.

Acknowledgments

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

Reviewers for the 8th Edition

Robert Burn
Diablo Valley College

Michael Dixon
Sacramento City College

Qiang Duan
Penn State University—Abington

Daniel Edwards
Ohlone College

Xisheng Fang
Ohlone College

Ken Hang
Green River Community College

Kay Johnson
Community College of Rhode Island

Michelle Levine
Broward College

Cindy Lindstrom
Lakeland College

Susan Reeder
Seattle University

Sandra Roberts
Snead College

Lopa Roychoudhuri
Angelo State University

Richard Snyder
Lehigh Carbon Community College

Donald Southwell
Delta College

Chadd Williams
Pacific University

Reviewers for Previous Editions

Ahmad Abuhejleh
University of Wisconsin–River Falls

David Akins
El Camino College

Steve Allan
Utah State University

Vicki Allan
Utah State University

Karen M. Arlien
Bismark State College

Mary Astone
Troy University

Ijaz A. Awan
Savannah 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

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

Jeanne Douglas
University of Vermont

Michael Dowell
Augusta State U

William E. Duncan
Louisiana State University

Judy Etchison
Southern Methodist University

Dennis Fairclough
Utah Valley State 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 Gleiberman
Touro College

Barbara Guillott
Louisiana State University

Ranette Halverson, Ph.D.
Midwestern State University

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

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

Jennifer Li
Ohlone College

Norman H. Liebling
San Jacinto 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

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

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

Vida Winans
Illinois Institute of Technology

Reviewers for the 8th Global Edition

Shaligram Prajapat
Devi Ahilya University

Vikas Saxena
Jaypee Institute of Information Technology

Andres Baravelle
University of East London

I would like to thank my family for their love and support in all of my many projects. I am extremely fortunate to have Matt Goldstein as my editor. I am also fortunate to have Kathryn Ferranti as marketing coordinator. She does a great job getting my books out to the academic community. I had a great production team led by Marilyn Lloyd and Kayla Smith-Tarbox. 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.

CHAPTER 1

Introduction to Computers and Programming

TOPICS

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

1.1 Why Program?

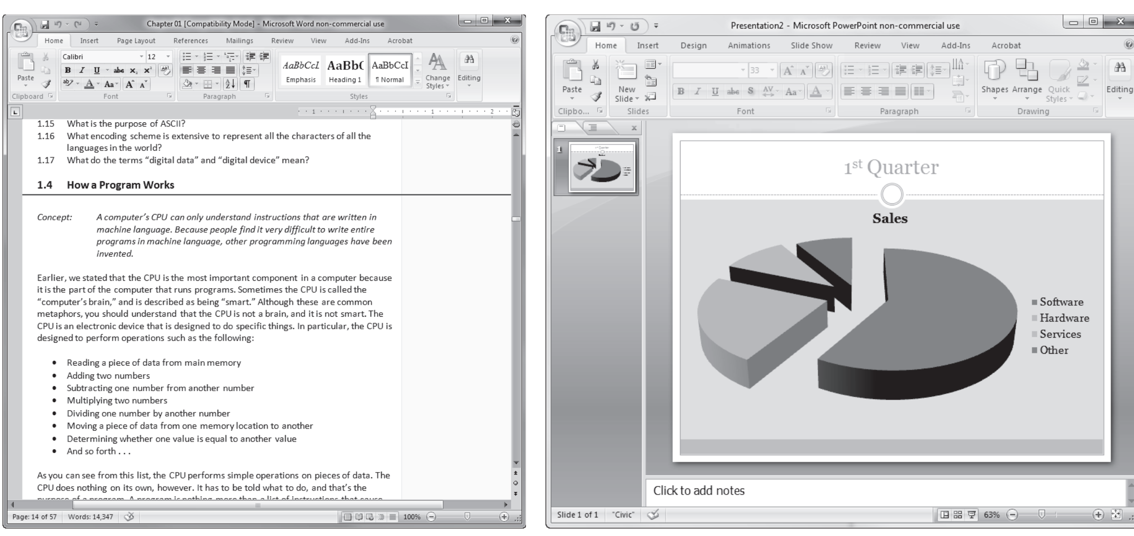
CONCEPT: Computers can do many different jobs because they are programmable.

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 analyze data, make presentations, conduct business transactions, communicate with customers and coworkers, control machines in manufacturing facilities, and do many other things. At home, people use computers for tasks such as paying bills, shopping online, social networking, and playing computer games. And don't forget that smart phones, iPods®, 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 that 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 that 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, academics, 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"
- Manuals and other forms of written documentation

There is also a scientific, or engineering, side to programming. Because programs rarely work right the first time they are written, a lot of 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.

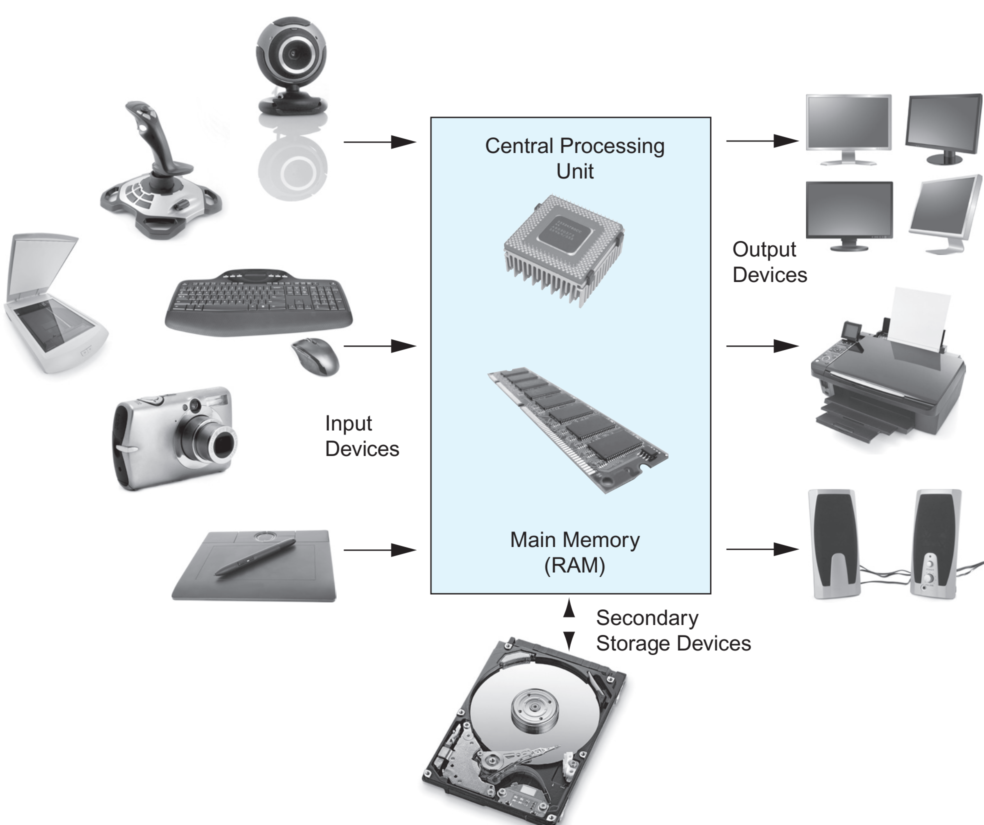
Hardware

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

- 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

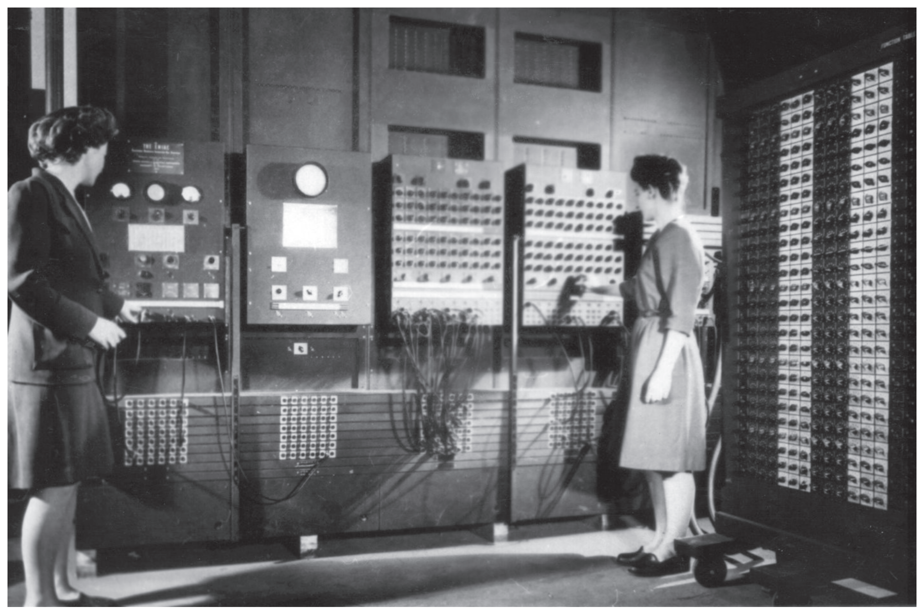


The CPU

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

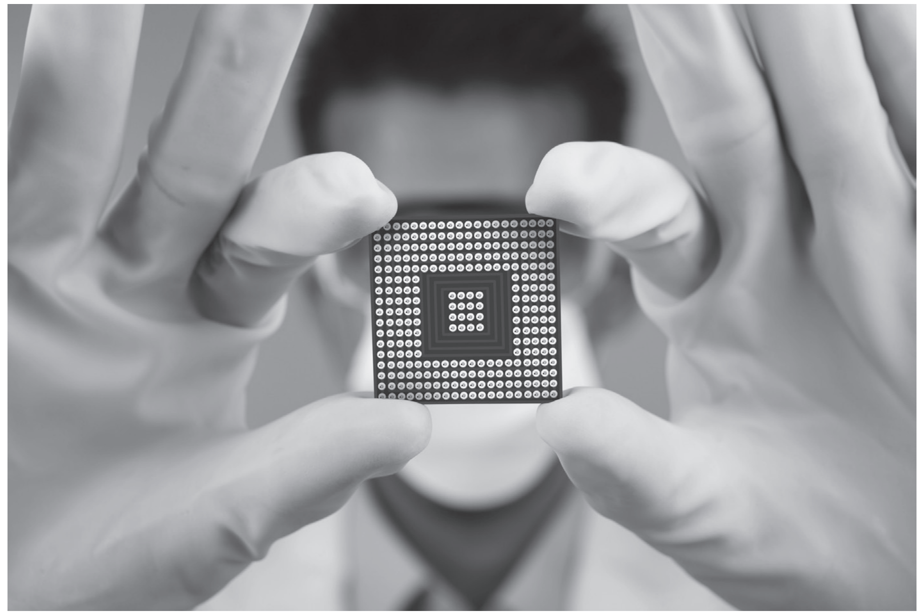
Figure 1-3



the 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.

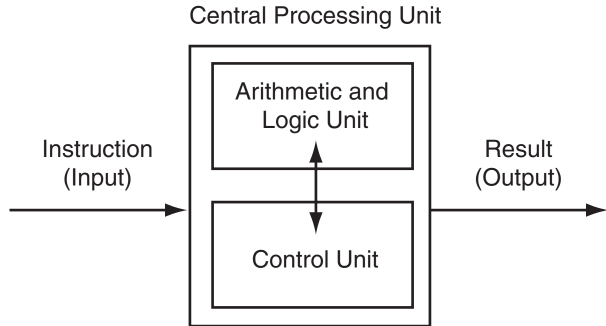
Today, CPUs are small chips known as *microprocessors*. Figure 1-4 shows a photo of a lab technician holding a modern-day microprocessor. In addition to being much smaller than the old electro-mechanical CPUs in early computers, microprocessors are also much more powerful.

Figure 1-4



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



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

Fetch	The CPU's control unit fetches, from main memory, the next instruction in the sequence of program instructions.
Decode	The instruction is encoded in the form of a number. The control unit decodes the instruction and generates an electronic signal.
Execute	The signal is routed to the appropriate component of the computer (such as the ALU, a disk drive, or some other device). The signal causes the component to perform an operation.

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

Main Memory

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 that the program is working with. 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

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16 **149**	17	18	19
20	21	22	23 **72**	24	25	26	27	28	29

Secondary Storage

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

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

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

Optical devices such as the *CD* (compact disc) and the *DVD* (digital versatile disc) are also popular for data storage. Data is not recorded magnetically on an optical disc, but 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 information the computer collects from the outside world. The device that collects the information and sends it to the computer is called an *input device*. Common input devices are the keyboard, mouse, scanner, digital camera, and microphone. Disk drives, CD/DVD drives, and USB 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 monitors, printers, and speakers. Disk drives, USB drives, and CD/DVD recorders can also be considered output devices because the CPU sends them information to be saved.

Software

If a computer is to function, software is not optional. Everything that 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 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* and 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, and then use special software to convert their programs to machine language.

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

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    }
```

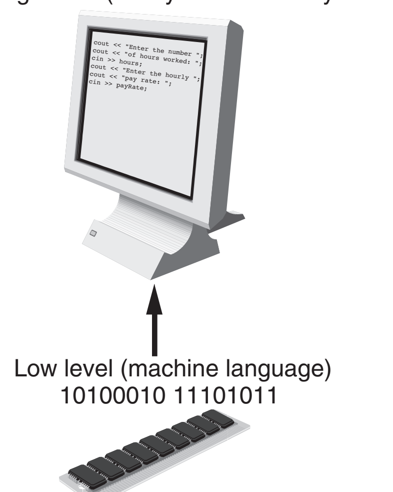
Program Output with Example Input Shown in Bold

```
How many hours did you work? 10 [Enter]
How much do you get paid per hour? 15 [Enter]
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



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

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 invented at Sun Microsystems. Java may be used to develop programs that run over the Internet, in a Web browser.
JavaScript	JavaScript can be used to write small programs that run in Web pages. 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 and 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 Mac OS 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*.

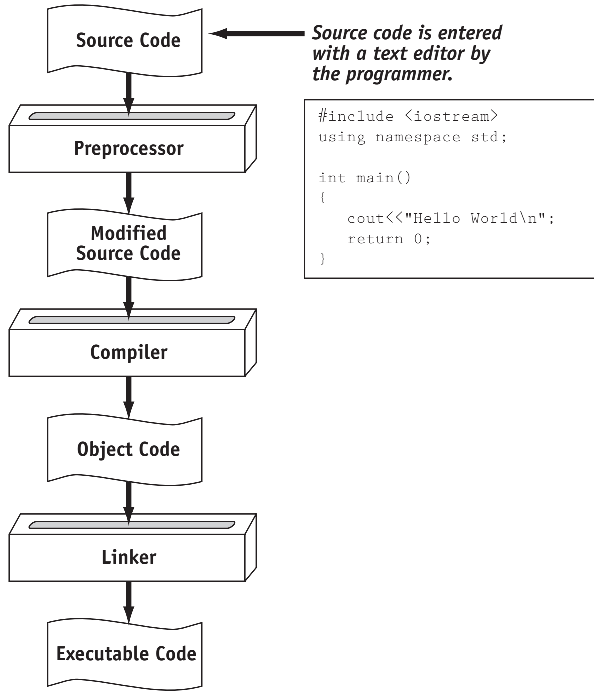
Although an object file contains machine language instructions, it is not a complete program. Here is why: C++ is conveniently equipped with a library of prewritten code for performing common operations or sometimes-difficult tasks. For example, the library contains hardware-specific code for displaying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

Figure 1-8 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
```

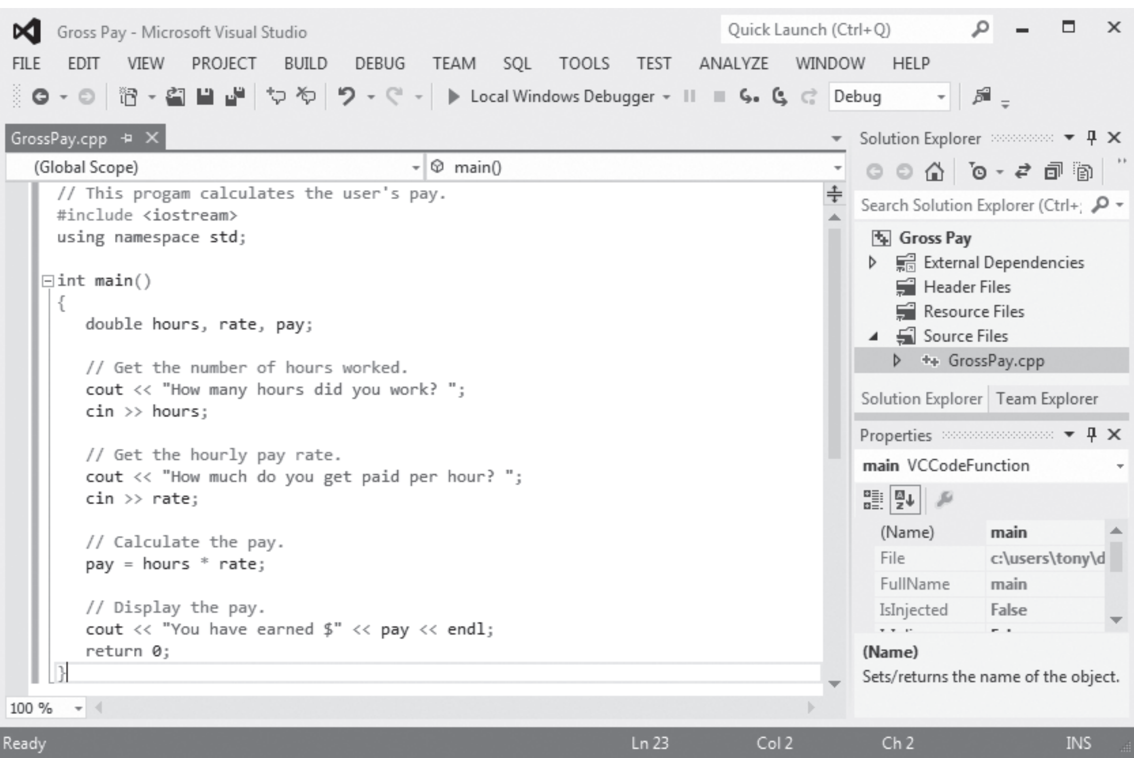
Figure 1-8



Appendix G explains how compiling works in .Net. You can download Appendix G from the book's companion Web site at www.pearsonglobaleditions.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



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

Let's look at some specific parts of Program 1-1 (the pay-calculating program) to see examples of each element listed in the table above. For 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 that 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 is shown with each of its lines numbered. Most of the lines contain something meaningful; however, some of the lines are empty. The blank lines are only there to make the program more readable.

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

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

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 that in Program 1-1 the words `hours`, `rate`, and `pay` appear in several places. All three of these are the names of variables. The `hours` variable is used to store the number of hours the user 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 is discussed further in Chapter 2.

Variables are symbolic names that represent locations in the computer's random-access memory (RAM). When information is stored in a variable, it is actually stored in RAM. Assume a program has a variable named `length`. Figure 1-10 illustrates the way the variable name represents a memory location.

Figure 1-10

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23 **72**	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, and 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

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:

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

2. **Visualize the program running on the computer.**

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

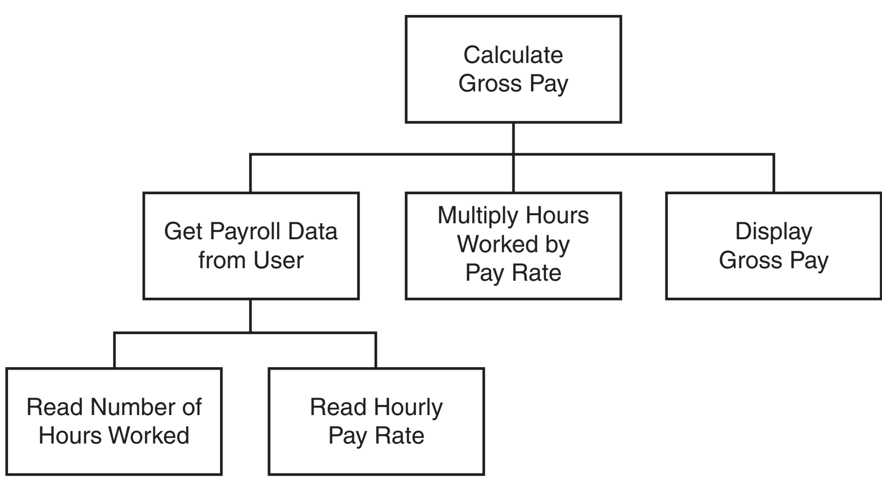
```
How many hours did you work? 10
How much do you get paid per hour? 15
You have earned $150
```

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.

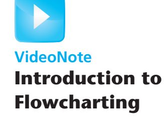
3. **Use design tools such as a hierarchy chart, flowcharts, or pseudocode to create a model of the program.**

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



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



VideoNote
Introduction to Flowcharting

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 D, 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:

VideoNote
Designing a Program with Pseudocode

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 run-time errors. A run-time error is an error that occurs while the program is running. These are usually logical errors, such as mathematical mistakes.

Testing for run-time 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 run-time 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 run-time 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 though 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 run-time 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 that are 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.

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 by Figure 1-13.

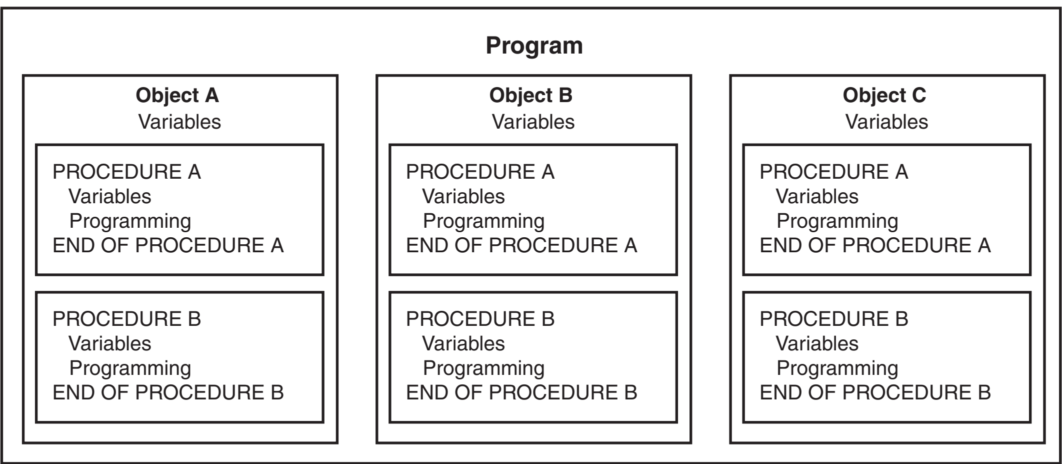
Figure 1-13

```
Program
  PROCEDURE A
    Variables
    Programming
  END OF PROCEDURE A

  PROCEDURE B
    Variables
    Programming
  END OF PROCEDURE B
```

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



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 run-time 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. What is a source code and an object code?
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. The __________ is the part of a computer in which programs actually execute.
11. The main memory is commonly known as __________.
12. The two general categories of software are __________ and __________.
13. A program is a set of __________.
14. A(n) __________ is an ordered set of steps written in human language to perform a task or to solve any problem.
15. __________ is the only language computers really process.
16. __________ languages are close to the level of humans in terms of readability.
17. C++ is a high-level language that has __________ features.
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. A(n) __________ translates each source code instruction to the equivalent machine language instruction(s).

22. __________ characters or symbols mark the beginning or ending of programming statements, or separate items in a list.
23. The statement `#include<iostream>` is a(n) __________.
24. A(n) __________ is a named storage location.
25. When data is stored in a variable, it is actually stored in the __________ of the computer.
26. The three primary activities of a program are __________, __________, and __________.
27. __________ is information a program gathers from the outside world.
28. A(n) __________ is a diagrammatic representation of the logical flow of a program.
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 D 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

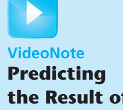
VideoNote **Designing the Account Balance Program**

    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?


VideoNote
Predicting the Result of Problem 33

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 `x` starts with the value 20.
    The variable `y` starts with the value 10.
    Store the value of `x` in a variable `t`.
    Store the value of `y` in `x`.
    Store the value of `t` in `y`.
    Display the value of `x` on the screen.
    Display the value of `y` 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, and 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.

CHAPTER

2 Introduction to C++

TOPICS

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

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

Line 3 reads:

```
using namespace std;
```

Programs usually contain 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 contents of the function `main`.

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

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

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

To put it simply, this line displays a message on the screen. You will read more about `cout` and the `<<` operator later in this chapter. The message “Programming is great fun!” is printed without the quotation marks. In programming terms, the group of characters inside the quotation marks is called a *string literal* 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
`//`	Double slash	Marks the beginning of a comment.
`#`	Pound sign	Marks the beginning of a preprocessor directive.
`< >`	Opening and closing brackets	Encloses a filename when used with the `#include` directive.
`( )`	Opening and closing parentheses	Used in naming a function, as in `int main()`
`{ }`	Opening and closing braces	Encloses a group of statements, such as the contents of a function.
`" "`	Opening and closing quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen.
`;`	Semicolon	Marks the end of a complete programming statement.

Checkpoint

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

```
int main()
}
// A crazy mixed up program
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.

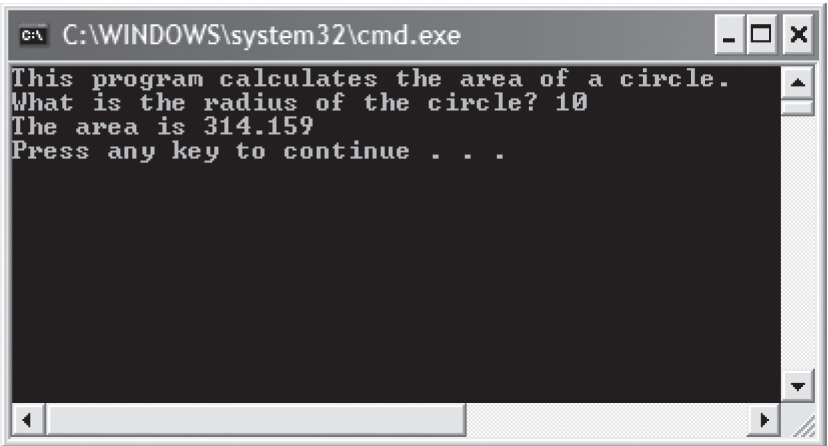
2.2 The cout Object

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 **out**put.)

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

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

```
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 a single line. Unless you specify otherwise, the information you send to `cout` is displayed in a continuous stream. Sometimes this can produce less-than-desirable results. Program 2-4 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 sellersduring the month of June:Computer
gamesCoffeeAspirin
```

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

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

O	n	e	\n	T	w	o	\n	T	h	r	e	e	\n

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:

Program Output

```
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 On paper, write a program that will display your name on the first line, your street address on the second line, your city, state, and ZIP code on the third line, and your telephone number on the fourth line. Place a comment with today's date at the top of the program. Test your program by entering, compiling, and running it.

2.4 Variables and Literals

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;
```

VideoNote
Variabe Definitions

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 that 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 that `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;
```

(program continues)

Program 2-9 *(continued)*

```
 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

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 that 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?
`dayOfWeek`	Legal.
`3dGraph`	Illegal. Variable names cannot begin with a digit.
`_employee_num`	Legal.
`June1997`	Legal.
`Mixture#3`	Illegal. Variable names may only use letters, digits, 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, that they don't even have a name. Then there is textual information. Names and addresses, for instance, are stored as groups of characters. When you write a program you must determine what types of information it will be likely to encounter.

If you are writing a program to calculate the number of miles to a distant star, you'll need variables that can hold very large numbers. If you are designing software to record microscopic dimensions, you'll need to store very small and precise numbers. Additionally, if you are writing a program that must perform thousands of intensive calculations, you'll want 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 that 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      days = 189000;
14      cout << "We have made a long journey of " << miles;
15      cout << " miles.\n";
16      cout << "Our checking account balance is " << checking;
17      cout << "\nAbout " << days << " days ago Columbus ";
18      cout << "stood on this spot.\n";
19      return 0;
20  }
```

Program Output

```
We have made a long journey of 4276 miles.
Our checking account balance is -20
About 189000 days ago Columbus stood on this spot.
```

In most programs you will need more than one variable of any given data type. If a program uses two integers, `length` and `width`, they 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 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
 5  int main()
```

(program continues)

Program 2-11 *(continued)*

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

Program 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`.

If you want an integer literal to be treated as a `long long 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 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 −40,000 to +40,000, 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 B, which shows the ASCII character set. Notice that the number 65 is the code for A, 66 is the code for B, and so on. Program 2-13 demonstrates that when you work with characters, you are actually working with numbers.

Program 2-13

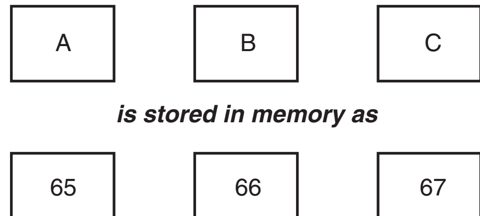
```
1  // This program demonstrates the close relationship between
2  // characters and integers.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8      char letter;
9
10     letter = 65;
11     cout << letter << endl;
12     letter = 66;
13     cout << letter << endl;
14     return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

Figure 2-4



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 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 B, 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

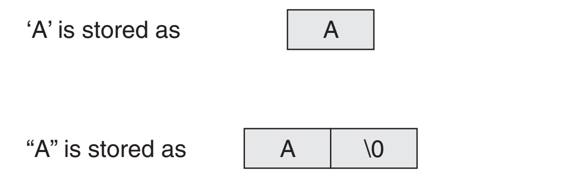
First, notice the quotation marks are not stored with the string. They are simply a way of marking the beginning and end of the string in your source code. Second, notice the very last byte of the string. It contains the null terminator, which is represented by the `\0` character. The addition of this 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.

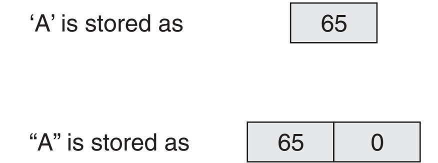
Now let's compare the way a string and a `char` are stored. Suppose you have the literals `'A'` and `"A"` in a program. Figure 2-6 depicts their internal storage.

Figure 2-6



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

Figure 2-7



Because 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 B, 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 B)

```
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 and 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 ten.

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 sizes and ranges of floating-point data types usually found on PCs.

Table 2-8 Floating Point Data Types on PCs

Data Type	Key Word	Description
Single precision	`float`	4 bytes. Numbers between ±3.4E-38 and ±3.4E38
Double precision	`double`	8 bytes. Numbers between ±1.7E-308 and ±1.7E308
Long double precision	`long double`	8 bytes*. Numbers between ±1.7E-308 and ±1.7E308

*Some compilers use 10 bytes for long doubles. This allows a range of ±3.4E-4932 to ±1.1E4832

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 that 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`, many 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
89.2l
```

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:

Program Output

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 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.

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 that 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.

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

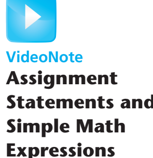
```
 1  // This program can't find its variable.
 2  #include <iostream>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      cout << value; // ERROR! value not defined yet!
 8
 9      int value = 100;
10      return 0;
11  }
```

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.


VideoNote
Assignment Statements and Simple Math Expressions

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++ only has 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 98 to `temperature`:

```
temperature = 112 - 14;
```

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` times `regularHours` and stores the result in `regularWages`:

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

Here is line 19, which multiplies `overtimePayRate` times `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      return 0;
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 ÷ 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`, etc.

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.
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     return 0;
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 that 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: 8/20/2014
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 that 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: 8/20/2014
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 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 following 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%, 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, and 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      return 0;
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 feet per second^2)
9.8	The gravitational acceleration constant (in meters per second^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 indentions in a program so visual cues are created. These cues quickly tell a programmer important information about a program.

For example, notice in Program 2-30 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 can the values of three variables length, breadth, and height be displayed using one statement, such that each variable is displayed in a new line?
4. Assume that x is an integer variable, y is a floating point variable, and z is a character variable. Write statements in C++ to perform the following operations:
   A) Add 10 to x.
   B) Square the value of y.
   C) Divide x by 7 and stores the result in y.
   D) Store '&' in z.
   E) Display the values of y and z on the screen.
   F) Display the rightmost digit of x on the screen.
   G) Display the number of bytes used by an integer variable.
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\namaze\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. Which of the following statements is correct?
   A) `cout << x.`
   B) `cout >> x;`
   C) `cout << x;`
   D) `cout >> x.`
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. When you are writing `cout`, you are using
   A) a ternary operator
   B) a library function
   C) a stream insertion operator
   D) a stream object
12. Preprocessor directives begin with a
   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. If ch is a char type of variable, which of the following statements is wrong?
   A) `ch="abcd";`
   B) `ch='a';`
   C) `ch='6';`
   D) `ch='?';`

15. Which of the following is wrong with respect to an arithmetic statement?
    A) The `lvalue` is assigned to the `rvalue`.
    B) It always has a binary operator in it.
    C) It usually evaluates to a numeric value.
    D) It occurs within a function.
16. Which of the following are *not* valid `cout` statements? (Circle 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 The `cout` object is used along with the stream insertion operator.
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 The variable name `Qwert@#1` is legal.

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 10.5 in the *radius* variable.
    Store 20 in the *height* variable.
    Multiply 3.141593, the square of *radius*, and the *height* and store the result in the *volume* variable.
    Display the contents of the *volume* 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

1. **Product of Three Numbers**

   Write a program that assigns the values 10, 20, and 30 in three integer variables and then stores the product of these three variables in another variable, `product`.

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. **Conversion**

   Write a program that finds the length of a wire in millimeters if its length in feet-inches is 9 feet and 8 inches. Note that 1 feet = 12 inches and 1 inch = 25.4 mm.

VideoNote
Solving the Restaurant Bill Problem

4. Restaurant Bill

Write a program that computes the tax and tip on a restaurant bill for a patron with a $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 and 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:

`payAmount` This variable will hold the amount of pay the employee earns each pay period. Initialize the variable with 2200.0.

`payPeriods` This variable will hold the number of pay periods in a year. Initialize the variable with 26.

`annualPay` 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

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%.

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 manuals, 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):

MPG = Miles Driven/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:

Distance = Number of Gallons × Average Miles per Gallon

12. Weight of Freight

One ton is equivalent to 2240 pounds. Write a program that calculates the weight, in tons, of freight that weighs 10158 pounds.

13. Surface Area and Volume of a Sphere

The surface area of a sphere of radius `r` is calculated by using the formula `A = 4πr2`, and its volume is calculated using the formula `V = (4/3) πr3`. Write a program that calculates the surface area and volume of a sphere that has a radius of 8 cm. (Note: `π = 3.1415926535`)

14. List of Purchases

Write a program that displays the following list of purchases. Include all the newlines, tabs and quotes shown.

```
LIST OF ITEMS PURCHASED
"Price of item 1" =     $32.95
"Price of item 2" =     $42.78
"Price of item 3" =     $57.65
```

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

CHAPTER 3

Expressions and Interactivity

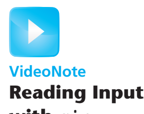
TOPICS

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

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.



VideoNote
Reading Input with `cin`

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.

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      return 0;
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 [Enter]
What is the width of the rectangle? 20 [Enter]
The area of the rectangle is 200.
```

Instead of calculating the area of one rectangle, this program can be used to 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 that 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

```
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` 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      return 0;
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      return 0;
18  }
```

Program Output with Example Input Shown in Bold

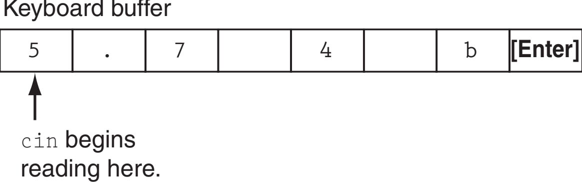
```
Enter an integer, a double, and a character: 4 5.7 b [Enter]
Whole: 4
Fractional: 5.7
Letter: b
```

As you can see in the example output, the values are stored in 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 [Enter]
```

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



When the user presses the Enter 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 [Enter] 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      return 0;
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:

A) 6 is divided by 3, yielding a result of 2
B) 12 is added to 2, yielding a result of 14

It could be diagrammed in the following way:

```
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)`

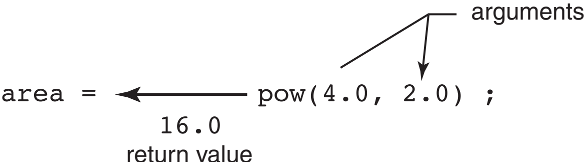
No Exponents Please!

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

$$\text{area} = 4^2$$

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

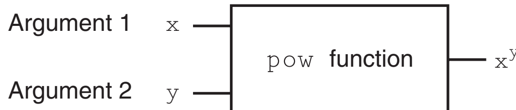
```
x = 3 * pow(6.0, 3.0);
```

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



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 `double`s. 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 [Enter]
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 and 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`, and 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     return 0;
26 }
```

Program Output with Example Input Shown in Bold

Enter the first test score: **90 [Enter]**
Enter the second test score: **80 [Enter]**
Enter the third test score: **100 [Enter]**
The average score is 90

Checkpoint

3.7 Complete the table below by writing the value of each expression in the "Value" column.

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

3.8 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 `value3`)?
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

Volume $= \pi r^2 h$

where

π is 3.14159
r is the radius of the tank
h is the height of the tank

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