

C++ PROGRAMMING: FROM PROBLEM ANALYSIS TO PROGRAM DESIGN

SIXTH EDITION

D.S. MALIK

Australia Brazil Japan Korea Mexico Singapore Spain United Kingdom United States

This is an electronic version of the print textbook. Due to electronic rights restrictions, some third party content may be suppressed. Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. The publisher reserves the right to remove content from this title at any time if subsequent rights restrictions require it. For valuable information on pricing, previous editions, changes to current editions, and alternate formats, please visit www.cengage.com/highered to search by ISBN#, author, title, or keyword for materials in your areas of interest.

C++ Programming: From Problem Analysis to Program Design, Sixth Edition

D.S. Malik

Executive Editor: Marie Lee

Acquisitions Editor: Brandi Shailer

Senior Product Manager: Alyssa Pratt

Associate Product Manager: Stephanie

Lorenz

Content Project Manager: Matthew

Hutchinson

Art Director: Faith Brosnan

Print Buyer: Julio Esperas

Cover Designer: Roycroft Design/

www.roycroftdesign.com

Cover Photo: ª Masterfile Royalty Free

Proofreader: Andrea Schein

Indexer: Elizabeth Cunningham

Compositor: Integra Software Services

ª Cengage Learning

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be

reproduced, transmitted, stored or used in any form or by any means graphic, electronic, or mechanical, including but not limited to

photocopying, recording, scanning, digitizing, taping, Web distribution, information

networks, or information storage and retrieval systems, except as permitted under Section or of the United States Copyright Act, without the prior written permission of the publisher.

Printed in the United States of America 1 2 3 4 5 6 7 16 17 16 15 14 13 12

| For product information and technology assistance, contact us at Cengage Learning Customer & Sales Support,  www.cengage.com/support  For permission to use material from this text or product, submit all requests online at www.cengage.com/permissions  Further permissions questions can be emailed to  permissionrequest@cengage.com |
| --- |

Library of Congress Control Number:

ISBN- : - - - -

Cengage Learning

Channel Center Street

Boston, MA

USA

Some of the product names and company names used in this book have been used for identification purposes only and may be trademarks or registered trademarks of their respective manufacturers and sellers.

Any fictional data related to persons or companies or URLs used throughout this book is intended for instructional purposes only. At the time this book was printed, any such data was fictional and not belonging to any real persons or companies.

Cengage Learning reserves the right to revise this publication and make changes from time to time in its content without notice.

The programs in this book are for instructional purposes only. They have been tested with care, but are not guaranteed for any particular intent beyond educational purposes. The author and the publisher do not offer any warranties or representations, nor do they accept any liabilities with respect to the programs.

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil and Japan. Locate your local office at:

www.cengage.com/global

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

To learn more about Cengage Learning, visit

www.cengage.com

Purchase any of our products at your local college store or at our preferred online store www.CengageBrain.com

TO

My Daughter

Shelly Malik

*This page intentionally left blank*

BRIEF CONTENTS

PREFACE xxix

1. An Overview of Computers and Programming Languages 1 2. Basic Elements of C++ 27 3. Input/Output 121 4. Control Structures I (Selection) 183 5. Control Structures II (Repetition) 259 6. User-Defined Functions 335

7. User-Defined Simple Data Types, Namespaces,

and the string Type 451 8. Arrays and Strings 505 9. Records (structs) 591 10. Classes and Data Abstraction 629 11. Inheritance and Composition 709 12. Pointers, Classes, Virtual Functions, and Abstract Classes 781 13. Overloading and Templates 853 14. Exception Handling 943 15. Recursion 985 16. Searching, Sorting, and the vector Type 1015 17. Linked Lists 1057 18. Stacks and Queues 1149

vi | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

APPENDIX A Reserved Words 1249 APPENDIX B Operator Precedence 1251 APPENDIX C Character Sets 1253 APPENDIX D Operator Overloading 1257 APPENDIX E Additional C++ Topics 1259 APPENDIX F Header Files 1281

APPENDIX G Memory Size on a System and Random

Number Generator 1291

APPENDIX H Standard Template Library (STL) 1293 APPENDIX I Answers to Odd-Numbered Exercises 1335 INDEX 1371

TABLE OF CONTENTS

Preface xxix

AN OVERVIEW OF COMPUTERS AND PROGRAMMING 1

LANGUAGES 1 Introduction 2 A Brief Overview of the History of Computers 2

Elements of a Computer System 3 Hardware 4 Central Processing Unit and Main Memory 4 Input /Output Devices 5 Software 6

The Language of a Computer 6 The Evolution of Programming Languages 8 Processing a C++ Program 10

Programming with the Problem

Analysis–Coding–Execution Cycle 12

Programming Methodologies 20 Structured Programming 20 Object-Oriented Programming 20

ANSI/ISO Standard C++ 22 Quick Review 22 Exercises 24

viii | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

BASIC ELEMENTS OF C++ 27

2

A Quick Look at a C++ Program 28

The Basics of a C++ Program 34 Comments 34 Special Symbols 35 Reserved Words (Keywords) 36 Identifiers 36 Whitespaces 37

Data Types 37 Simple Data Types 38 Floating-Point Data Types 41

Data Types and Variables 42

Arithmetic Operators, Operator Precedence,

and Expressions 43 Order of Precedence 46 Expressions 48 Mixed Expressions 49

Type Conversion (Casting) 51 string Type 53

Variables, Assignment Statements, and Input

Statements 54 Allocating Memory with Constants and Variables 54 Putting Data into Variables 57 Assignment Statement 57 Saving and Using the Value of an Expression 61 Declaring and Initializing Variables 62 Input (Read) Statement 63 Variable Initialization 66

Increment and Decrement Operators 70 Output 72

Preprocessor Directives 79 namespace and Using cin and cout in a Program 80 Using the string Data Type in a Program 81

Table of Contents | ix

Creating a C++ Program 81 Debugging: Understanding and Fixing Syntax Errors 85

Program Style and Form 89 Syntax 89 Use of Blanks 90 Use of Semicolons, Brackets, and Commas 90 Semantics 90 Naming Identifiers 90 Prompt Lines 91 Documentation 92 Form and Style 92

More on Assignment Statements 94 Programming Example: Convert Length 96 Programming Example: Make Change 99 Quick Review 103 Exercises 105 Programming Exercises 114

INPUT/OUTPUT 121

3

I/O Streams and Standard I/O Devices 122 cin and the Extraction Operator >> 123

Using Predefined Functions in a Program 128 cin and the get Function 131 cin and the ignore Function 133 The putback and peek Functions 134 The Dot Notation between I/O Stream Variables and I/O

Functions: A Precaution 137

Input Failure 138 The clear Function 140

x | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

Output and Formatting Output 142 setprecision Manipulator 142 fixed Manipulator 143 showpoint Manipulator 144 setw 147

Additional Output Formatting Tools 149 setfill Manipulator 149 left and right Manipulators 151

Input/Output and the string Type 153

Debugging: Understanding Logic Errors and Debugging

with cout Statements 154 File Input/Output 157

Programming Example: Movie Tickets Sale and

Donation to Charity 161 Programming Example: Student Grade 167 Quick Review 170 Exercises 171 Programming Exercises 177

CONTROL STRUCTURES I (SELECTION) 183

4

Control Structures 184

Relational Operators 185 Relational Operators and Simple Data Types 186 Comparing Characters 187 Relational Operators and the string Type 188

Logical (Boolean) Operators and Logical Expressions 190 Order of Precedence 192 int Data Type and Logical (Boolean) Expressions 195 bool Data Type and Logical (Boolean) Expressions 196

Selection: if and if...else 196 One-Way Selection 197 Two-Way Selection 200

Table of Contents | xi

Compound (Block of) Statements 203 Multiple Selections: Nested if 204 Comparing if...else Statements with a Series of if

Statements 206 Short-Circuit Evaluation 207 Comparing Floating-Point Numbers for Equality:

A Precaution 208 Associativity of Relational Operators: A Precaution 209 Avoiding Bugs by Avoiding Partially Understood Concepts

and Techniques 211 Input Failure and the if Statement 214 Confusion between the Equality Operator (==) and the

Assignment Operator (=) 217 Conditional Operator (?:) 219 Program Style and Form (Revisited): Indentation 219

Using Pseudocode to Develop, Test, and Debug

a Program 220

switch Structures 223 Avoiding Bugs by Avoiding Partially Understood Concepts and Techniques (Revisited) 229

Terminating a Program with the assert Function 231 Programming Example: Cable Company Billing 233 Quick Review 239 Exercises 240 Programming Exercises 251

CONTROL STRUCTURES II (REPETITION) 259

5

Why Is Repetition Needed? 260

while Looping (Repetition) Structure 261 Designing while Loops 263 Case 1: Counter-Controlled while Loops 264 Case 2: Sentinel-Controlled while Loops 268 Telephone Digits 271

xii | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

Case 3: Flag-Controlled while Loops 273 Number Guessing Game 274 Case 4: EOF-Controlled while Loops 277 eof Function 277 More on Expressions in while Statements 282

Programming Example: Fibonacci Number 283 for Looping (Repetition) Structure 287 Programming Example: Classifying Numbers 295

do...while Looping (Repetition) Structure 298 Divisibility Test by 3 and 9 301 Choosing the Right Looping Structure 303

break and continue Statements 303 Nested Control Structures 305 Avoiding Bugs by Avoiding Patches 310 Debugging Loops 313 Quick Review 314 Exercises 315 Programming Exercises 328

USER-DEFINED FUNCTIONS 335

6

Predefined Functions 336 User-Defined Functions 340

Value-Returning Functions 341 Syntax: Value-Returning function 343 Syntax: Formal Parameter List 343 Function Call 343 Syntax: Actual Parameter List 344 return Statement 344 Syntax: return Statement 344 Function Prototype 348 Syntax: Function Prototype 349

Table of Contents | xiii

Value-Returning Functions: Some Peculiarities 350 More Examples of Value-Returning Functions 352 Flow of Execution 361

Void Functions 364 Value Parameters 370

Reference Variables as Parameters 371 Calculate Grade 372

Value and Reference Parameters and Memory Allocation 376 Reference Parameters and Value-Returning Functions 386 Scope of an Identifier 386 Global Variables, Named Constants, and Side Effects 390 Static and Automatic Variables 395 Debugging: Using Drivers and Stubs 396 Function Overloading: An Introduction 399 Functions with Default Parameters 400 Programming Example: Classify Numbers 403 Programming Example: Data Comparison 408 Quick Review 418 Exercises 422 Programming Exercises 436

USER-DEFINED SIMPLE DATA TYPES,

7

NAMESPACES, AND THE STRING TYPE 451

Enumeration Type 452 Declaring Variables 454 Assignment 454 Operations on Enumeration Types 455 Relational Operators 455 Input /Output of Enumeration Types 456

xiv | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

Functions and Enumeration Types 459 Declaring Variables When Defining the Enumeration Type 460 Anonymous Data Types 461 typedef Statement 461

Programming Example: The Game of Rock, Paper,

and Scissors 463 Namespaces 471

string Type 476 Additional string Operations 480

Programming Example: Pig Latin Strings 490 Quick Review 494 Exercises 496 Programming Exercises 501

ARRAYS AND STRINGS 505

8

Arrays 507 Accessing Array Components 509 Processing One-Dimensional Arrays 511 Array Index Out of Bounds 515 Array Initialization During Declaration 516 Partial Initialization of Arrays During Declaration 516 Some Restrictions on Array Processing 517 Arrays as Parameters to Functions 518 Constant Arrays as Formal Parameters 519 Base Address of an Array and Array in Computer Memory 521 Functions Cannot Return a Value of the Type Array 524 Integral Data Type and Array Indices 526 Other Ways to Declare Arrays 527

Searching an Array for a Specific Item 527 Selection Sort 530

C-Strings (Character Arrays) 535 String Comparison 537 Reading and Writing Strings 539

Table of Contents | xv

String Input 539 String Output 540 Specifying Input/Output Files at Execution Time 541 string Type and Input/Output Files 541

Parallel Arrays 542

Two- and Multidimensional Arrays 543 Accessing Array Components 545 Two-Dimensional Array Initialization During Declaration 546 Two-Dimensional Arrays and Enumeration Types 546 Initialization 549 Print 550 Input 550 Sum by Row 550 Sum by Column 551 Largest Element in Each Row and Each Column 551 Passing Two-Dimensional Arrays as Parameters to Functions 552 Arrays of Strings 555 Arrays of Strings and the string Type 555 Arrays of Strings and C-Strings (Character Arrays) 555 Another Way to Declare a Two-Dimensional Array 556 Multidimensional Arrays 557

Programming Example: Code Detection 559 Programming Example: Text Processing 565 Quick Review 572 Exercises 573 Programming Exercises 584

RECORDS (STRUCTS) 591

9

Records (structs) 592 Accessing struct Members 594 Assignment 596 Comparison (Relational Operators) 597 Input /Output 598

xvi | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

struct Variables and Functions 598 Arrays versus structs 599 Arrays in structs 600 structs in Arrays 602 structs within a struct 604

Programming Example: Sales Data Analysis 608 Quick Review 622 Exercises 622 Programming Exercises 626

10

CLASSES AND DATA ABSTRACTION 629

Classes 630 Unified Modeling Language Class Diagrams 634 Variable (Object) Declaration 634 Accessing Class Members 635 Built-in Operations on Classes 636 Assignment Operator and Classes 637 Class Scope 637 Functions and Classes 638 Reference Parameters and Class Objects (Variables) 638 Implementation of Member Functions 639 Accessor and Mutator Functions 644 Order of public and private Members of a Class 647 Constructors 649 Invoking a Constructor 651 Invoking the Default Constructor 651 Invoking a Constructor with Parameters 651 Constructors and Default Parameters 654 Classes and Constructors: A Precaution 654 Arrays of Class Objects (Variables) and Constructors 655 Destructors 657

Data Abstraction, Classes, and Abstract Data Types 658 A struct Versus a class 660

Table of Contents | xvii

Information Hiding 661 Executable Code 665 More Examples of Classes 667 Static Members of a Class 673 Programming Example: Juice Machine 679 Quick Review 693 Exercises 695 Programming Exercises 703

INHERITANCE AND COMPOSITION 709

11

Inheritance 710 Redefining (Overriding) Member Functions

of the Base Class 713 Constructors of Derived and Base Classes 720 Destructors in a Derived Class 729 Multiple Inclusions of a Header File 730 C++ Stream Classes 731 Protected Members of a Class 733 Inheritance as public, protected, or private 733 (Accessing protected Members in the Derived Class) 734

Composition (Aggregation) 737

Object-Oriented Design (OOD) and Object-Oriented

Programming (OOP) 742 Identifying Classes, Objects, and Operations 744

Programming Example: Grade Report 745 Quick Review 766 Exercises 767 Programming Exercises 776

xviii | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

12

POINTERS, CLASSES, VIRTUAL FUNCTIONS, AND ABSTRACT CLASSES 781

Pointer Data Type and Pointer Variables 782 Declaring Pointer Variables 782

Address of Operator (&) 783 Dereferencing Operator (\*) 784 Classes, Structs, and Pointer Variables 789 Initializing Pointer Variables 792

Dynamic Variables 792 Operator new 793 Operator delete 794

Operations on Pointer Variables 798

Dynamic Arrays 800 Functions and Pointers 803 Pointers and Function Return Values 803 Dynamic Two-Dimensional Arrays 804

Shallow versus Deep Copy and Pointers 807

Classes and Pointers: Some Peculiarities 809 Destructor 809 Assignment Operator 811 Copy Constructor 812

Inheritance, Pointers, and Virtual Functions 819 Classes and Virtual Destructors 826

Abstract Classes and Pure Virtual Functions 826 Address of Operator and Classes 835 Quick Review 837 Exercises 840 Programming Exercises 849

13

Table of Contents | xix

OVERLOADING AND TEMPLATES 853 Why Operator Overloading Is Needed 854

Operator Overloading 855 Syntax for Operator Functions 856 Overloading an Operator: Some Restrictions 856 Pointer this 857 Friend Functions of Classes 861 Operator Functions as Member Functions and Nonmember

Functions 864 Overloading Binary Operators 867 Overloading the Stream Insertion (<<) and Extraction (>>)

Operators 873 Overloading the Assignment Operator (=) 878 Overloading Unary Operators 886 Operator Overloading: Member versus Nonmember 892 Classes and Pointer Member Variables (Revisited) 893 Operator Overloading: One Final Word 893

Programming Example: clockType 893 Programming Example: Complex Numbers 902 Overloading the Array Index (Subscript) Operator ([]) 907 Programming Example: newString 909 Function Overloading 915

Templates 916 Function Templates 916 Class Templates 918

Quick Review 926 Exercises 928 Programming Exercises 934

xx | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

14 15

EXCEPTION HANDLING 943

Handling Exceptions within a Program 944 C++ Mechanisms of Exception Handling 948 try/catch Block 948 Using C++ Exception Classes 955

Creating Your Own Exception Classes 959 Rethrowing and Throwing an Exception 968

Exception-Handling Techniques 972 Terminate the Program 972 Fix the Error and Continue 972 Log the Error and Continue 974

Stack Unwinding 974 Quick Review 978 Exercises 980 Programming Exercises 984

RECURSION 985

Recursive Definitions 986 Direct and Indirect Recursion 988 Infinite Recursion 988

Problem Solving Using Recursion 989 Tower of Hanoi: Analysis 999

Recursion or Iteration? 999

Programming Example: Converting a Number from Binary to Decimal 1001

Programming Example: Converting a Number from Decimal to Binary 1005

Quick Review 1008 Exercises 1009 Programming Exercises 1012

16 17

Table of Contents | xxi

SEARCHING, SORTING, AND THE VECTOR TYPE 1015

List Processing 1016 Searching 1016 Bubble Sort 1017 Insertion Sort 1021 Binary Search 1025 Performance of Binary Search 1028

vector Type (class) 1029 Programming Example: Election Results 1034 Quick Review 1049 Exercises 1050 Programming Exercises 1053

LINKED LISTS 1057

Linked Lists 1058 Linked Lists: Some Properties 1059 Deletion 1065 Building a Linked List 1066

Linked List as an ADT 1071 Structure of Linked List Nodes 1072 Member Variables of the class linkedListType 1072 Linked List Iterators 1073 Print the List 1079 Length of a List 1079 Retrieve the Data of the First Node 1080 Retrieve the Data of the Last Node 1080 Begin and End 1080 Copy the List 1081 Destructor 1082 Copy Constructor 1082 Overloading the Assignment Operator 1083

xxii | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

Unordered Linked Lists 1083 Search the List 1084 Insert the First Node 1085 Insert the Last Node 1086 Header File of the Unordered Linked List 1091

Ordered Linked Lists 1092 Search the List 1093 Insert a Node 1094 Insert First and Insert Last 1098 Delete a Node 1099 Header File of the Ordered Linked List 1100

Print a Linked List in Reverse Order

(Recursion Revisited) 1103 printListReverse 1105

Doubly Linked Lists 1106 Default Constructor 1109 isEmptyList 1109 Destroy the List 1109 Initialize the List 1110 Length of the List 1110 Print the List 1110 Reverse Print the List 1110 Search the List 1111 First and Last Elements 1111

Circular Linked Lists 1117 Programming Example: DVD Store 1118 Quick Review 1138 Exercises 1138 Programming Exercises 1144

18

Table of Contents | xxiii

STACKS AND QUEUES 1149

Stacks 1150 Stack Operations 1152

Implementation of Stacks as Arrays 1154 Initialize Stack 1157 Empty Stack 1158 Full Stack 1158 Push 1158 Return the Top Element 1160 Pop 1160 Copy Stack 1162 Constructor and Destructor 1162 Copy Constructor 1163 Overloading the Assignment Operator (=) 1163 Stack Header File 1164

Programming Example: Highest GPA 1168

Linked Implementation of Stacks 1172 Default Constructor 1175 Empty Stack and Full Stack 1175 Initialize Stack 1176 Push 1176 Return the Top Element 1178 Pop 1178 Copy Stack 1180 Constructors and Destructors 1181 Overloading the Assignment Operator (=) 1181 Stack as Derived from the class unorderedLinkedList 1184

Application of Stacks: Postfix Expressions Calculator 1185 Main Algorithm 1188 Function evaluateExpression 1188 Function evaluateOpr 1190 Function discardExp 1192 Function printResult 1192

xxiv | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

Removing Recursion: Nonrecursive Algorithm to

Print a Linked List Backward 1195

Queues 1199 Queue Operations 1200 Implementation of Queues as Arrays 1202 Linked Implementation of Queues 1211 Queue Derived from the

class unorderedLinkedListType 1216

Application of Queues: Simulation 1217 Designing a Queuing System 1218 Customer 1219 Server 1222 Server List 1225 Waiting Customers Queue 1230 Main Program 1232

Quick Review 1237 Exercises 1238 Programming Exercises 1245

APPENDIX A: RESERVED WORDS 1249 APPENDIX B: OPERATOR PRECEDENCE 1251

APPENDIX C: CHARACTER SETS 1253

ASCII (American Standard Code for Information

Interchange) 1253

EBCDIC (Extended Binary Coded Decimal

Interchange Code) 1254 APPENDIX D: OPERATOR OVERLOADING 1257

Table of Contents | xxv

APPENDIX E: ADDITIONAL C++ TOPICS 1259

Binary (Base 2) Representation of a Nonnegative Integer 1259 Converting a Base 10 Number to a Binary Number

(Base 2) 1259 Converting a Binary Number (Base 2) to Base 10 1261 Converting a Binary Number (Base 2) to Octal (Base 8) and Hexadecimal (Base 16) 1262

More on File Input/Output 1264 Binary Files 1264 Random File Access 1270

Naming Conventions of Header Files in ANSI/ISO Standard C++ and Standard C++ 1278

APPENDIX F: HEADER FILES 1281 Header File cassert (assert.h) 1281 Header File cctype (ctype.h) 1282 Header File cfloat (float.h) 1283 Header File climits (limits.h) 1284

Header File cmath (math.h) 1286 Header File cstddef (stddef.h) 1287 Header File cstring (string.h) 1287

APPENDIX G: MEMORY SIZE ON A SYSTEM AND RANDOM NUMBER GENERATOR 1291

Random Number Generator 1292

APPENDIX H: STANDARD TEMPLATE LIBRARY (STL) 1293 Components of the STL 1293

Container Types 1294 Sequence Containers 1294 Sequence Container: Vectors 1294

xxvi | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

Member Functions Common to All Containers 1303 Member Functions Common to Sequence Containers 1305 copy Algorithm 1306 Sequence Container: deque 1310 Sequence Container: list 1313

Iterators 1318 IOStream Iterators 1319 Container Adapters 1319

Algorithms 1323 STL Algorithm Classification 1323 STL Algorithms 1326 Functions fill and fill\_n 1326 Functions find and find\_if 1328 Functions remove and replace 1329 Functions search, sort, and binary\_search 1331

APPENDIX I: ANSWERS TO ODD-NUMBERED

EXERCISES 1335 Chapter 1 1335 Chapter 2 1338 Chapter 3 1340 Chapter 4 1341 Chapter 5 1344 Chapter 6 1347 Chapter 7 1350 Chapter 8 1351 Chapter 9 1353 Chapter 10 1354 Chapter 11 1358 Chapter 12 1360

Table of Contents | xxvii

Chapter 13 1361 Chapter 14 1362 Chapter 15 1364 Chapter 16 1364 Chapter 17 1366 Chapter 18 1367

INDEX 1371

*This page intentionally left blank*

PREFACE

WELCOME TO THE SIXTH EDITION OF C++ Programming: From Problem Analysis to Program Design. Designed for a first Computer Science (CS1) C++ course, this text provides a breath of fresh air to you and your students. The CS1 course serves as the cornerstone of the Computer Science curriculum. My primary goal is to motivate and excite all CS1 students, regardless of their level. Motivation breeds excitement for learning. Motivation and excitement are critical factors that lead to the success of the programming student. This text is a culmination and development of my classroom notes throughout more than fifty semesters of teaching successful programming to Computer Science students.

| Warning: This text can be expected to create a serious reduction in the demand for programming help during your office hours. Other side effects include significantly diminished student dependency on others while learning to program. |
| --- |

C++ Programming: From Problem Analysis to Program Design started as a collection of brief examples, exercises, and lengthy programming examples to supplement the books that were in use at our university. It soon turned into a collection large enough to develop into a text. The approach taken in this book is, in fact, driven by the students’ demand for clarity and readability. The material was written and rewritten until the students felt comfortable with it. Most of the examples in this book resulted from student interaction in the classroom.

As with any profession, practice is essential. Cooking students practice their recipes. Budding violinists practice their scales. New programmers must practice solving problems and writing code. This is not a C++ cookbook. We do not simply list the C++ syntax followed by an example; we dissect the ‘‘why’’ behind all the concepts. The crucial question of ‘‘why?’’ is answered for every topic when first introduced. This technique offers a bridge to learning C++. Students must understand the ‘‘why?’’ in order to be motivated to learn.

Traditionally, a C++ programming neophyte needed a working knowledge of another programming language. This book assumes no prior programming experience. However, some adequate mathematics background, such as college algebra, is required.

xxx | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

Changes in the Sixth Edition

The sixth edition contains 200 new exercises, and more than 25 new programming exercises. Earlier editions contain two chapters on user-defined functions. In this edition, without sacrificing the rigor, these chapters are combined into one chapter so that user-defined functions can be learned without interruption. Since Chapters 6 and 7 of earlier editions have been combined into one chapter, the sixth edition contains one less chapter than the earlier editions.

The first part of Chapter 2 is rewritten and reorganized. Chapter 10, on searching and sorting algorithms and the class vector is now Chapter 16. However, the selection sorting algorithm is moved from Chapter 10 to Chapter 8 (arrays and string). So in addition to learning about array and strings, the reader can also study a sequential search algorithm and a selection sort algorithm. Even though additional searching and sorting algorithms are covered in Chapter 16, Chapter 16 can be studied right after studying Chapter 8. This edition also includes various new examples, such as Examples 3-4, 3-8, 3-9, 4-8, 5-3, 5-4, 6-1, 8-4, 10-8, 11-2, 12-5, and 14-14.

Approach

The programming language C++, which evolved from C, is no longer considered an industry-only language. Numerous colleges and universities use C++ for their first program ming language course. C++ is a combination of structured programming and object-oriented programming, and this book addresses both types.

This book can be easily divided into two parts: structured programming and object-oriented programming. The first 9 chapters form the structured programming part; Chapters 10 through 14, 17, and 18 form the object-oriented part. However, only the first six chapters are essential to move on to the object-oriented portion.

In July 1998, ANSI/ISO Standard C++ was officially approved. This book focuses on ANSI/ ISO Standard C++. Even though the syntax of Standard C++ and ANSI/ISO Standard C++ is very similar, Chapter 7 discusses some of the features of ANSI/ISO Standard C++ that are not available in Standard C++.

Chapter 1 briefly reviews the history of computers and programming languages. The reader can quickly skim through this chapter and become familiar with some of the hardware components and the software parts of the computer. This chapter contains a section on processing a C++ program. This chapter also describes structured and object-oriented programming.

Chapter 2 discusses the basic elements of C++. After completing this chapter, students become familiar with the basics of C++ and are ready to write programs that are complicated enough to do some computations. Input/output is fundamental to any programming language. It is introduced early, in Chapter 3, and is covered in detail.

Chapters 4 and 5 introduce control structures to alter the sequential flow of execution. Chapter 6 studies user-defined functions. It is recommended that readers with no prior programming background spend extra time on Chapter 6. Several examples are provided to help readers understand the concepts of parameter passing and the scope of an identifier.

Preface | xxxi

Chapter 7 discusses the user-defined simple data type (enumeration type), the namespace mechanism of ANSI/ISO Standard C++, and the string type. The earlier versions of C did not include the enumeration type. Enumeration types have very limited use; their main purpose is to make the program readable. This book is organized such that readers can skip the section on enumeration types during the first reading without experiencing any disconti nuity, and then later go through this section.

Chapter 8 discusses arrays in detail. This chapter also discusses a sequential search algorithm and a selection sort algorithm. Chapter 9 introduces records (structs). The introduction of structs in this book is similar to C structs. This chapter is optional; it is not a prerequisite for any of the remaining chapters.

Chapter 10 begins the study of object-oriented programming (OOP) and introduces classes. The first half of this chapter shows how classes are defined and used in a program. The second half of the chapter introduces abstract data types (ADTs). This chapter shows how classes in C++ are a natural way to implement ADTs. Chapter 11 continues with the fundamentals of object-oriented design (OOD) and OOP and discusses inheritance and composition. It explains how classes in C++ provide a natural mechanism for OOD and how C++ supports OOP. Chapter 11 also discusses how to find the objects in a given problem.

Chapter 12 studies pointers in detail. After introducing pointers and how to use them in a program, this chapter highlights the peculiarities of classes with pointer data members and how to avoid them. Moreover, this chapter also discusses how to create and work with dynamic two-dimensional arrays. Chapter 12 also discusses abstract classes and a type of polymorphism accomplished via virtual functions.

Chapter 13 continues the study of OOD and OOP. In particular, it studies polymorphism in C++. The chapter specifically discusses two types of polymorphism—overloading and templates.

Chapter 14 discusses exception handling in detail. Chapter 15 introduces and discusses recursion. Moreover, this is a stand-alone chapter, so it can be studied anytime after Chapter 9. Chapter 16 describes various searching and sorting algorithms as well as an introduction to the vector class.

Chapters 17 and 18 are devoted to the study of data structures. Discussed in detail are linked lists in Chapter 17 and stacks and queues in Chapter 18. The programming code developed in these chapters is generic. These chapters effectively use the fundamentals of OOD.

Appendix A lists the reserved words in C++. Appendix B shows the precedence and associativity of the C++ operators. Appendix C lists the ASCII (American Standard Code for Information Interchange) and EBCDIC (Extended Binary Coded Decimal Interchange Code) character sets. Appendix D lists the C++ operators that can be overloaded.

Appendix E has three objectives. First, we discuss how to convert a number from decimal to binary and binary to decimal. We then discuss binary and random access files in detail. Finally, we describe the naming conventions of the header files in both ANSI/ISO Standard C++ and Standard C++. Appendix F discusses some of the most widely used library routines, and includes the names of the standard C++ header files. The programs in Appendix G show how to print the memory size for the built-in data types on your system as well as how to use a random number generator. Appendix H gives an introduction to

xxxii | C++ Programming: From Problem Analysis to Program Design, Sixth Edition

the Standard Template Library, and Appendix I provides the answers to odd-numbered exercises in the book.

How to Use the Book

This book can be used in various ways. Figure 1 shows the dependency of the chapters.

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7 Chapter 8\*

Chapter 16 Chapter 9 Chapter 10

Chapter 14

Chapter 15

Chapter 17

Chapter 18

FIGURE 1 Chapter dependency diagram

Chapter 11 Chapter 12 Chapter 13

Preface | xxxiii

In Figure 1, dotted lines mean that the preceding chapter is used in one of the sections of the chapter and is not necessarily a prerequisite for the next chapter. For example, Chapter 8 covers arrays in detail. In Chapters 9 and 10, we show the relationship between arrays and structs and arrays and classes, respectively. However, if Chapter 10 is studied before Chapter 8, then the section dealing with arrays in Chapter 10 can be skipped without any discontinuation. This particular section can be studied after studying Chapter 8.

It is recommended that the first six chapters be covered sequentially. After covering the first six chapters, if the reader is interested in learning OOD and OOP early, then Chapter 10 can be studied right after Chapter 6. Chapter 7 can be studied anytime after Chapter 6.

After studying the first six chapters in sequence, some of the approaches are: 1. Study chapters in the sequence: 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18. 2. Study chapters in the sequence: 8, 10, 12, 13, 11, 15, 17, 18, 16, 15. 3. Study chapters in the sequence: 10, 8, 16, 12, 13, 11, 15, 17, 18, 14. 4. Study chapters in the sequence: 10, 8, 12, 13, 11, 15, 17, 18, 16, 14.

FEATURES OF THE BOOK

Four-color interior

design shows

accurate C++

code and related

comments.

One video is

available for each

chapter on the

optional

CourseMate that

accompanies this

text. Each video is

designed to

explain how a

program works.

More than 300 visual diagrams, both extensive and exhaustive, illustrate difficult concepts.

Numbered Examples illustrate the key

concepts with their relevant code. The programming code in these examples is followed by a Sample Run. An explanation then follows that

describes what each line in the code does.

Notes highlight

important facts

about the concepts introduced in the chapter.

Programming Examples are

where everything in the

chapter comes together.

These examples teach

problem-solving skills and

include the concrete stages

of input, output, problem

analysis and algorithm

design, class design, and

a program listing. All

programs are designed to

be methodical, consistent,

and user-friendly. Each

Programming Example

starts with a problem

analysis that is followed

by the algorithm design

and/or class design, and

every step of the algorithm

is coded in C++. In

addition to helping

students learn problem

solving techniques, these

detailed programs show

the student how to

implement concepts in

an actual C++ program.

We strongly recommend

that students study the

Programming Examples

carefully in order to learn

C++ effectively. Students

typically learn much from

completely worked-out

programs. Further,

programming examples considerably reduce the students’ need for help outside the classroom and bolster the students’ self-confidence.

Exercises further reinforce learning and ensure that students have, in fact, mastered the material.

Programming

Exercises challenge students to write C++ programs with a specified

outcome.

SUPPLEMENTAL

RESOURCES

CourseMate

Make the most of your study time with everything you need to succeed in one place. Read your textbook, highlight and take notes, review flashcards, watch videos, and take practice quizzes online. Learn more at www.cengage.com/coursemate.

The C++ Programming CourseMate includes the following features:

• Videos step you through programs in each chapter, while integrated quizzes provide immediate feedback to gauge your understanding.

• Lab Manual lets you apply material with a wealth of practical, hands-on exercises. • Interactive Quizzes and Study Games drill key chapter concepts, while open ended Assignments develop critical thinking skills.

• Interactive eBook, flashcards, and more!

Instructors may add CourseMate to the textbook package, or students may purchase CourseMate directly through www.cengagebrain.com.

Source Code

The source code, in ANSI/ISO Standard C++, is available for students to download at www.cengagebrain.com and through the CourseMate available for this text. These files are also available to instructors on the Instructor Resources CD and at login.cengage.com. The input files needed to run some of the programs are also included with the source code.

Instructor Resources

The following supplemental materials are available when this book is used in a classroom setting. All instructor teaching tools are available with this book on a single CD-ROM. Many are also available for download at login.cengage.com.

Supplemental Resources | xli

Electronic Instructor’s Manual

The Instructor’s Manual follows the text chapter-by-chapter and includes material to assist in planning and organizing an effective, engaging course.

The Manual includes Overviews, Chapter Objectives, Teaching Tips, Quick Quizzes, Class Discussion Topics, Additional Projects, Additional Resources, and Key Terms. A Sample Syllabus is also available.

ExamView

This textbook is accompanied by ExamView, a powerful testing software package that allows instructors to create and administer printed, computer (LAN-based), and Internet exams. ExamView includes hundreds of questions that correspond to the topics covered in this text, enabling students to generate detailed study guides that include page references for further review. These computer-based and Internet testing components allow students to take exams at their computers, and save the instructor time because each exam is graded automatically. The test banks are also available in Blackboard, WebCT, and Angel compatible formats.

PowerPoint Presentations

This book comes with PowerPoint slides to accompany each chapter. Slides may be used to guide classroom presentation, to make available to students for chapter review, or to print as classroom handouts. Instructors can add their own slides for additional topics that they introduce to the class, as well as customize the slides with the complete Figure Files from the text.

Solution Files

The solution files for all Programming Exercises, in ANSI/ISO C++, are available at login.cengage.com, and are also available on the Instructor Resources CD-ROM. The input files needed to run some of the Programming Exercises are also included with the solution files.

ACKNOWLEDGMENTS

There are many people that I must thank who, one way or another, contributed to the success of this book. First, I would like to thank all the students who, during the preparation, were spontaneous in telling me if certain portions needed to be reworded for better understanding and clearer reading. Next, I would like to thank those who e-mailed numerous comments to help improve upon the next edition. I am thankful to Professors S.C. Cheng and Randall Crist for constantly supporting this project. I am also very grateful to the reviewers who reviewed earlier versions of this book and offered many critical suggestions on how to improve it.

I owe a great deal to the following reviewers who made helpful, critical suggestions for improving this edition of the text: Gary Bricher, Lane Community College; Cliff Brozo, Monroe College; and Marie Pullan, Farmingdale State College.

Next, I express thanks to Brandi Shailer, Acquisition Editor, for recognizing the importance and uniqueness of this project. All this would not have been possible without the careful planning of Senior Product Manager, Alyssa Pratt. I extend my sincere thanks to Alyssa, as well as to Content Project Manager, Mathew Hutchinson. My special thanks are to Stephanie Lorenz for using her expertise in carefully editing the videos. I also thank Sreemannarayana Reddy of Integra Software Services for assisting us in keeping the project on schedule. I would like to thank Chris Scriver and Serge Palladino of the QA department at Cengage Learning for patiently and carefully testing the code and discovering typos and errors.

I am thankful to my parents for their blessings.

Finally, I am thankful for the support of my wife Sadhana and especially my daughter, Shelly, to whom this book is dedicated. They cheered me up whenever I was overwhelmed during the writing of this book.

I welcome any comments concerning the text. Comments may be forwarded to the following e-mail address: malik@creighton.edu

D. S. Malik

CHAPTER 1

AN OVERVIEW OF

COMPUTERS AND

PROGRAMMING

LANGUAGES

IN THIS CHAPTER, YOU WILL:

. Learn about different types of computers

. Explore the hardware and software components of a computer system . Learn about the language of a computer

. Learn about the evolution of programming languages

. Examine high-level programming languages

. Discover what a compiler is and what it does

. Examine a C++ program

. Explore how a C++ program is processed

. Learn what an algorithm is and explore problem-solving techniques

. Become aware of structured design and object-oriented design programming methodologies . Become aware of Standard C++ and ANSI/ISO Standard C++

2 | Chapter 1: An Overview of Computers and Programming Languages

Introduction

Terms such as ‘‘the Internet,’’ which were unfamiliar just 20 years ago are now common. Students in elementary school regularly ‘‘surf ’’ the Internet and use computers to design their classroom projects. Many people use the Internet to look for information and to commu nicate with others. This is all made possible by the availability of different software, also known as computer programs. Without software, a computer is useless. Software is devel oped by using programming languages. The programming language C++ is especially well suited for developing software to accomplish specific tasks. Our main objective is to help you learn how to write programs in the C++ programming language. Before you begin programming, it is useful to understand some of the basic terminology and different components of a computer. We begin with an overview of the history of computers.

A Brief Overview of the History of Computers

The first device known to carry out calculations was the abacus. The abacus was invented in Asia but was used in ancient Babylon, China, and throughout Europe until the late middle ages. The abacus uses a system of sliding beads in a rack for addition and subtraction. In 1642, the French philosopher and mathematician Blaise Pascal invented the calculating device called the Pascaline. It had eight movable dials on wheels and could calculate sums up to eight figures long. Both the abacus and Pascaline could perform only addition and subtrac tion operations. Later in the 17th century, Gottfried von Leibniz invented a device that was able to add, subtract, multiply, and divide. In 1819, Joseph Jacquard, a French weaver, discovered that the weaving instructions for his looms could be stored on cards with holes punched in them. While the cards moved through the loom in sequence, needles passed through the holes and picked up threads of the correct color and texture. A weaver could rearrange the cards and change the pattern being woven. In essence, the cards programmed a loom to produce patterns in cloth. The weaving industry may seem to have little in common with the computer industry. However, the idea of storing information by punching holes on a card proved to be of great importance in the later development of computers.

In the early and mid-1800s, Charles Babbage, an English mathematician and physical scientist, designed two calculating machines: the difference engine and the analytical engine. The difference engine could perform complex operations such as squaring numbers automatically. Babbage built a prototype of the difference engine, but did not build the actual device. The first complete difference engine was completed in London in 2002, 153 years after it was designed. It consists of 8,000 parts, weighs five tons, and measures 11 feet long. A replica of the difference engine was completed in 2008 and is on display at the Computer History Museum in Mountain View, California (http://www.computerhistory.org/ babbage/). Most of Babbage’s work is known through the writings of his colleague Ada Augusta, Countess of Lovelace. Augusta is considered the first computer programmer.

At the end of the 19th century, U.S. Census officials needed help in accurately tabulating the census data. Herman Hollerith invented a calculating machine that ran on electricity and used punched cards to store data. Hollerith’s machine was immensely successful.

Elements of a Computer System | 3

Hollerith founded the Tabulating Machine Company, which later became the computer 1

and technology corporation known as IBM.

The first computer-like machine was the Mark I. It was built, in 1944, jointly by IBM and Harvard University under the leadership of Howard Aiken. Punched cards were used to feed data into the machine. The Mark I was 52 feet long, weighed 50 tons, and had 750,000 parts. In 1946, the ENIAC (Electronic Numerical Integrator and Calculator) was built at the University of Pennsylvania. It contained 18,000 vacuum tubes and weighed some 30 tons.

The computers that we know today use the design rules given by John von Neumann in the late 1940s. His design included components such as an arithmetic logic unit, a control unit, memory, and input/output devices. These components are described in the next section. Von Neumann’s computer design makes it possible to store the programming instructions and the data in the same memory space. In 1951, the UNIVAC (Universal Automatic Computer) was built and sold to the U.S. Census Bureau.

In 1956, the invention of transistors resulted in smaller, faster, more reliable, and more energy-efficient computers. This era also saw the emergence of the software development industry, with the introduction of FORTRAN and COBOL, two early programming languages. In the next major technological advancement, transistors were replaced by tiny integrated circuits, or ‘‘chips.’’ Chips are smaller and cheaper than transistors and can contain thousands of circuits on a single chip. They give computers tremendous processing speed.

In 1970, the microprocessor, an entire CPU on a single chip, was invented. In 1977, Stephen Wozniak and Steven Jobs designed and built the first Apple computer in their garage. In 1981, IBM introduced its personal computer (PC). In the 1980s, clones of the IBM PC made the personal computer even more affordable. By the mid-1990s, people from many walks of life were able to afford them. Computers continue to become faster and less expensive as technology advances.

Modern-day computers are powerful, reliable, and easy to use. They can accept spoken-word instructions and imitate human reasoning through artificial intelligence. Expert systems assist doctors in making diagnoses. Mobile computing applications are growing significantly. Using hand-held devices, delivery drivers can access global positioning satellites (GPS) to verify customer locations for pickups and deliveries. Cell phones permit you to check your e-mail, make airline reservations, see how stocks are performing, and access your bank accounts.

Although there are several categories of computers, such as mainframe, midsize, and micro, all computers share some basic elements, described in the next section.

Elements of a Computer System

A computer is an electronic device capable of performing commands. The basic commands that a computer performs are input (get data), output (display result), storage, and perfor mance of arithmetic and logical operations. There are two main components of a computer system: hardware and software. In the next few sections, we give a brief overview of these components. Let’s look at hardware first.

4 | Chapter 1: An Overview of Computers and Programming Languages

Hardware

Major hardware components include the central processing unit (CPU); main memory (MM), also called random access memory (RAM); input/output devices; and secondary storage. Some examples of input devices are the keyboard, mouse, and secondary storage. Examples of output devices are the screen, printer, and secondary storage. Let’s look at each of these components in greater detail.

Central Processing Unit and Main Memory

The central processing unit is the ‘‘brain’’ of the computer and the single most expensive piece of hardware in a computer. The more powerful the CPU, the faster the computer. Arithmetic and logical operations are carried out inside the CPU. Figure 1-1(a) shows some hardware components.

Central

Input Device

. . .

. . .

Processing

1000 54

Unit (CPU) Main Memory

Output Device

1001 .

.

.

2000 2001 .

.

.

A. .

.

.

.

.

Secondary Storage

Main Memory

(a) (b)

FIGURE 1-1 Hardware components of a computer and main memory

Main memory, or random access memory, is connected directly to the CPU. All programs must be loaded into main memory before they can be executed. Similarly, all data must be brought into main memory before a program can manipulate it. When the computer is turned off, everything in main memory is lost.

Main memory is an ordered sequence of cells, called memory cells. Each cell has a unique location in main memory, called the address of the cell. These addresses help you access the information stored in the cell. Figure 1-1(b) shows main memory with some data.

Elements of a Computer System | 5

Today’s computers come with main memory consisting of millions to billions of cells. 1

Although Figure 1-1(b) shows data stored in cells, the content of a cell can be either a programming instruction or data. Moreover, this figure shows the data as numbers and letters. However, as explained later in this chapter, main memory stores everything as sequences of 0s and 1s. The memory addresses are also expressed as sequences of 0s and 1s.

SECONDARY STORAGE

Because programs and data must be stored in main memory before processing and because everything in main memory is lost when the computer is turned off, information stored in main memory must be transferred to some other device for permanent storage. The device that stores information permanently (unless the device becomes unusable or you change the information by rewriting it) is called secondary storage. To be able to transfer information from main memory to secondary storage, these components must be directly connected to each other. Examples of secondary storage are hard disks, flash drives, floppy disks, ZIP disks, CD-ROMs, and tapes.

Input /Output Devices

For a computer to perform a useful task, it must be able to take in data and programs and display the results of calculations. The devices that feed data and programs into computers are called input devices. The keyboard, mouse, and secondary storage are examples of input devices. The devices that the computer uses to display results are called output devices. A monitor, printer, and secondary storage are examples of output devices. Figure 1-2 shows some input and output devices.



Input devices Output devices

FIGURE 1-2 Some input and output devices

6 | Chapter 1: An Overview of Computers and Programming Languages

Software

Software are programs written to perform specific tasks. For example, word processors are programs that you use to write letters, papers, and even books. All software is written in programming languages. There are two types of programs: system programs and application programs.

System programs control the computer. The system program that loads first when you turn on your PC is called the operating system. Without an operating system, the computer is useless. The operating system monitors the overall activity of the computer and provides services. Some of these services include memory management, input/output activities, and storage management. The operating system has a special program that organizes secondary storage so that you can conveniently access information.

Application programs perform a specific task. Word processors, spreadsheets, and games are examples of application programs. The operating system is the program that runs application programs.

The Language of a Computer

When you press A on your keyboard, the computer displays A on the screen. But what is actually stored inside the computer’s main memory? What is the language of the computer? How does it store whatever you type on the keyboard?

Remember that a computer is an electronic device. Electrical signals are used inside the computer to process information. There are two types of electrical signals: analog and digital. Analog signals are continuous wave forms used to represent such things as sound. Audio tapes, for example, store data in analog signals. Digital signals represent information with a sequence of 0s and 1s. A 0 represents a low voltage, and a 1 represents a high voltage. Digital signals are more reliable carriers of information than analog signals and can be copied from one device to another with exact precision. You might have noticed that when you make a copy of an audio tape, the sound quality of the copy is not as good as the original tape. On the other hand, when you copy a CD, the copy is as good as the original. Computers use digital signals.

Because digital signals are processed inside a computer, the language of a computer, called machine language, is a sequence of 0s and 1s. The digit 0 or 1 is called a binary digit, or bit. Sometimes a sequence of 0s and 1s is referred to as a binary code or a binary number.

Bit: A binary digit 0 or 1.

A sequence of eight bits is called a byte. Moreover, 210 bytes = 1024 bytes is called a kilobyte (KB). Table 1-1 summarizes the terms used to describe various numbers of bytes.

The Language of a Computer | 7

1 TABLE 1-1 Binary Units Unit Symbol Bits/Bytes

|  |
| --- |
| KB |
| MB |
| GB |
| TB |
| PB |
| EB |

Byte 8 bits

Kilobyte210 bytes ¼ 1024 bytes

Megabyte1024 KB ¼ 210 KB ¼ 220 bytes ¼ 1,048,576 bytes Gigabyte1024 MB ¼ 210 MB ¼ 230 bytes ¼ 1,073,741,824 bytes

Terabyte1024 GB ¼ 210 GB ¼ 240 bytes ¼

1,099,511,627,776 bytes

Petabyte1024 TB ¼ 210 TB ¼ 250 bytes ¼

1,125,899,906,842,624 bytes

Exabyte1024 PB ¼ 210 PB ¼ 260 bytes ¼

1,152,921,504,606,846,976 bytes

Zettabyte ZB 1024 EB ¼ 210 EB ¼ 270 bytes ¼

1,180,591,620,717,411,303,424 bytes

Every letter, number, or special symbol (such as \* or {) on your keyboard is encoded as a sequence of bits, each having a unique representation. The most commonly used encoding scheme on personal computers is the seven-bit American Standard Code for Information Interchange (ASCII). The ASCII data set consists of 128 characters numbered 0 through 127. That is, in the ASCII data set, the position of the first character is 0, the position of the second character is 1, and so on. In this scheme, A is encoded as the binary number 1000001. In fact, A is the 66th character in the ASCII character code, but its position is 65 because the position of the first character is 0. Furthermore, the binary number 1000001 is the binary representation of 65. The character 3 is encoded as 0110011. Note that in the ASCII character set, the position of the character 3 is 51, so the character 3 is the 52nd character in the ASCII set. It also follows that 0110011 is the binary representation of 51. For a complete list of the printable ASCII character set, refer to Appendix C.

The number system that we use in our daily life is called the decimal system, or base 10. Because everything inside a computer is represented as a sequence of 0s and 1s, that is, binary numbers, the number system that a computer uses is called binary, or base 2. We indicated in the preceding paragraph that the number 1000001 is the binary representation of 65. Appendix E describes how to convert a number from base 10 to base 2 and vice versa.

8 | Chapter 1: An Overview of Computers and Programming Languages

Inside the computer, every character is represented as a sequence of eight bits, that is, as a byte. Now the eight-bit binary representation of 65 is 01000001. Note that we added 0 to the left of the seven-bit representation of 65 to convert it to an eight-bit representa tion. Similarly, the eight-bit binary representation of 51 is 00110011.

ASCII is a seven-bit code. Therefore, to represent each ASCII character inside the computer, you must convert the seven-bit binary representation of an ASCII character to an eight-bit binary representation. This is accomplished by adding 0 to the left of the seven-bit ASCII encoding of a character. Hence, inside the computer, the character A is represented as 01000001, and the character 3 is represented as 00110011.

There are other encoding schemes, such as EBCDIC (used by IBM) and Unicode, which is a more recent development. EBCDIC consists of 256 characters; Unicode consists of 65,536 characters. To store a character belonging to Unicode, you need two bytes.

The Evolution of Programming Languages

The most basic language of a computer, the machine language, provides program instructions in bits. Even though most computers perform the same kinds of operations, the designers of the computer may have chosen different sets of binary codes to perform the operations. Therefore, the machine language of one machine is not necessarily the same as the machine language of another machine. The only consistency among com puters is that in any modern computer, all data is stored and manipulated as binary codes.

Early computers were programmed in machine language. To see how instructions are written in machine language, suppose you want to use the equation:

wages = rate  hours

to calculate weekly wages. Further, suppose that the binary code 100100 stands for load, 100110 stands for multiplication, and 100010 stands for store. In machine language, you might need the following sequence of instructions to calculate weekly wages:

100100 010001

100110 010010

100010 010011

To represent the weekly wages equation in machine language, the programmer had to remember the machine language codes for various operations. Also, to manipulate data, the programmer had to remember the locations of the data in the main memory. This need to remember specific codes made programming not only very difficult, but also error prone.

Assembly languages were developed to make the programmer’s job easier. In assembly language, an instruction is an easy-to-remember form called a mnemonic. Table 1-2 shows some examples of instructions in assembly language and their corresponding machine language code.

The Evolution of Programming Languages | 9

TABLE 1-2 Examples of Instructions in Assembly Language and Machine Language 1

Assembly Language Machine Language

LOAD 100100

STOR 100010

MULT 100110

ADD 100101

SUB 100011

Using assembly language instructions, you can write the equation to calculate the weekly wages as follows:

LOAD rate

MULT hours

STOR wages

As you can see, it is much easier to write instructions in assembly language. However, a computer cannot execute assembly language instructions directly. The instructions first have to be translated into machine language. A program called an assembler translates the assembly language instructions into machine language.

Assembler: A program that translates a program written in assembly language into an equivalent program in machine language.

Moving from machine language to assembly language made programming easier, but a programmer was still forced to think in terms of individual machine instructions. The next step toward making programming easier was to devise high-level languages that were closer to natural languages, such as English, French, German, and Spanish. Basic, FORTRAN, COBOL, Pascal, C, C++, C#, and Java are all high-level languages. You will learn the high-level language C++ in this book.

In C++, you write the weekly wages equation as follows:

wages = rate \* hours;

The instruction written in C++ is much easier to understand and is self-explanatory to a novice user who is familiar with basic arithmetic. As in the case of assembly language, however, the computer cannot directly execute instructions written in a high-level language. To run on a computer, these C++ instructions first need to be translated into machine language. A program called a compiler translates instructions written in high level languages into machine code.

Compiler: A program that translates instructions written in a high-level language into the equivalent machine language.

10 | Chapter 1: An Overview of Computers and Programming Languages

Processing a C++ Program

In the previous sections, we discussed machine language and high-level languages and showed a C++ program. Because a computer can understand only machine language, you are ready to review the steps required to process a program written in C++.

Consider the following C++ program:

#include <iostream>

using namespace std;

int main()

{

cout << "My first C++ program." << endl;

}

return 0;

At this point, you need not be too concerned with the details of this program. However, if you run (execute) this program, it will display the following line on the screen:

My first C++ program.

Recall that a computer can understand only machine language. Therefore, in order to run this program successfully, the code must first be translated into machine language. In this section, we review the steps required to execute programs written in C++.

The following steps, as shown in Figure 1-3, are necessary to process a C++ program.

1. You use a text editor to create a C++ program following the rules, or syntax, of the high-level language. This program is called the source code, or source program. The program must be saved in a text file that has the extension .cpp. For example, if you saved the preceding program in the file named FirstCPPProgram, then its complete name is FirstCPPProgram.cpp.

Source program: A program written in a high-level language.

2. The C++ program given in the preceding section contains the statement #include <iostream>. In a C++ program, statements that begin with the symbol # are called preprocessor directives. These statements are pro cessed by a program called preprocessor.

3. After processing preprocessor directives, the next step is to verify that the program obeys the rules of the programming language—that is, the program is syntactically correct—and translate the program into the equivalent machine language. The compiler checks the source program for syntax errors and, if no error is found, translates the program into the equivalent machine language. The equivalent machine language program is called an object program.

Processing a C++ Program | 11

Object program: The machine language version of the high-level language 1

program.

4. The programs that you write in a high-level language are developed using an integrated development environment (IDE). The IDE contains many programs that are useful in creating your program. For example, it contains the necessary code (program) to display the results of the program and several mathematical functions to make the programmer’s job somewhat easier. Therefore, if certain code is already available, you can use this code rather than writing your own code. Once the program is developed and successfully compiled, you must still bring the code for the resources used from the IDE into your program to produce a final program that the computer can execute. This prewritten code (program) resides in a place called the library. A program called a linker combines the object program with the programs from libraries.

Linker: A program that combines the object program with other programs in the library and is used in the program to create the executable code.

5. You must next load the executable program into main memory for execu tion. A program called a loader accomplishes this task.

Loader: A program that loads an executable program into main memory. 6. The final step is to execute the program.

Figure 1-3 shows how a typical C++ program is processed.

C++ Program

Library

Editor

Preprocessor Compiler

Linker

Loader

Execution

Syntax Error

Step 1 Step 2 Step 3

Step 4 Step 5 Step 6

FIGURE 1-3 Processing a C++ program

As a programmer, you need to be concerned only with Step 1. That is, you must learn, understand, and master the rules of the programming language to create source programs.

12 | Chapter 1: An Overview of Computers and Programming Languages

As noted earlier, programs are developed using an IDE. Well-known IDEs used to create programs in the high-level language C++ include Visual C++ 2008 Express, Visual C++ 2010 Express, Visual Studio 2010 (from Microsoft), and C++ Builder (from Borland). You can also use Dev-C++ IDE from Bloodshed Software to create and test C++ programs. These IDEs contain a text editor to create the source program, a compiler to check the source program for syntax errors, a program to link the object code with the IDE resources, and a program to execute the program.

These IDEs are quite user friendly. When you compile your program, the compiler not only identifies the syntax errors, but also typically suggests how to correct them. More over, with just a simple command, the object code is linked with the resources used from the IDE. For example, the command that does the linking on Visual C++ 2008 Express, Visual C++ 2010 Express, and Visual Studio 2010 is Build or Rebuild. (For further clarification regarding the use of these commands, check the documentation of these IDEs.) If the program is not yet compiled, each of these commands first compiles the program and then links and produces the executable code.

The Web site http://msdn.microsoft.com/en-us/beginner/bb964629.aspx contains a video that explains how to use Visual C++ 2008 Express to write C++ programs.

Programming with the Problem

Analysis–Coding–Execution Cycle

Programming is a process of problem solving. Different people use different techniques to solve problems. Some techniques are nicely outlined and easy to follow. They not only solve the problem, but also give insight into how the solution was reached. These problem-solving techniques can be easily modified if the domain of the problem changes.

To be a good problem solver and a good programmer, you must follow good problem solving techniques. One common problem-solving technique includes analyzing a pro blem, outlining the problem requirements, and designing steps, called an algorithm, to solve the problem.

Algorithm: A step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.

In a programming environment, the problem-solving processrequires the following three steps:

1. Analyze the problem, outline the problem and its solution requirements, and design an algorithm to solve the problem.

2. Implement the algorithm in a programming language, such as C++, and verify that the algorithm works.

3. Maintain the program by using and modifying it if the problem domain changes. Figure 1-4 summarizes this three-step programming process.

Programming with the Problem Analysis–Coding–Execution Cycle | 13

1

Problem

Analysis

Algorithm

Design

Coding

Preprocessor

Library

Compiler No Error

Linker

Loader

Error

Execution Error

No Error

Results

FIGURE 1-4 Problem analysis–coding–execution cycle

To develop a program to solve a problem, you start by analyzing the problem. You then design the algorithm; write the program instructions in a high-level language, or code the program; and enter the program into a computer system.

Analyzing the problem is the first and most important step. This step requires you to do the following:

1. Thoroughly understand the problem.

2. Understand the problem requirements. Requirements can include whether the program requires interaction with the user, whether it manipulates data,

14 | Chapter 1: An Overview of Computers and Programming Languages

whether it produces output, and what the output looks like. If the program manipulates data, the programmer must know what the data is and how it is represented. That is, you need to look at sample data. If the program produces output, you should know how the results should be generated and formatted.

3. If the problem is complex, divide the problem into subproblems and repeat Steps 1 and 2. That is, for complex problems, you need to analyze each subproblem and understand each subproblem’s requirements.

After you carefully analyze the problem, the next step is to design an algorithm to solve the problem. If you broke the problem into subproblems, you need to design an algorithm for each subproblem. Once you design an algorithm, you need to check it for correctness. You can sometimes test an algorithm’s correctness by using sample data. At other times, you might need to perform some mathematical analysis to test the algorithm’s correctness.

Once you have designed the algorithm and verified its correctness, the next step is to convert it into an equivalent programming code. You then use a text editor to enter the programming code or the program into a computer. Next, you must make sure that the program follows the language’s syntax. To verify the correctness of the syntax, you run the code through a compiler. If the compiler generates error messages, you must identify the errors in the code, remove them, and then run the code through the compiler again. When all the syntax errors are removed, the compiler generates the equivalent machine code, the linker links the machine code with the system’s resources, and the loader places the program into main memory so that it can be executed.

The final step is to execute the program. The compiler guarantees only that the program follows the language’s syntax. It does not guarantee that the program will run correctly. During execution, the program might terminate abnormally due to logical errors, such as division by zero. Even if the program terminates normally, it may still generate erroneous results. Under these circumstances, you may have to reexamine the code, the algorithm, or even the problem analysis.

Your overall programming experience will be successful if you spend enough time to complete the problem analysis before attempting to write the programming instructions. Usually, you do this work on paper using a pen or pencil. Taking this careful approach to programming has a number of advantages. It is much easier to discover errors in a program that is well analyzed and well designed. Furthermore, a carefully analyzed and designed program is much easier to follow and modify. Even the most experienced programmers spend a considerable amount of time analyzing a problem and designing an algorithm.

Throughout this book, you will not only learn the rules of writing programs in C++, but you will also learn problem-solving techniques. Most of the chapters contain programming exam ples that discuss programming problems. These programming examples teach techniques of how to analyze and solve problems, design algorithms, code the algorithms into C++, and also help you understand the concepts discussed in the chapter. To gain the full benefit of this book, we recommend that you work through these programming examples.

Next, we provide examples of various problem-analysis and algorithm-design techniques.

Programming with the Problem Analysis–Coding–Execution Cycle | 15

1

EXAMPLE 1-1

In this example, we design an algorithm to find the perimeter and area of a rectangle.

To find the perimeter and area of a rectangle, you need to know the rectangle’s length and width.

The perimeter and area of the rectangle are then given by the following formulas: perimeter = 2  (length + width)

area = length  width

The algorithm to find the perimeter and area of the rectangle is:

1. Get the length of the rectangle.

2. Get the width of the rectangle.

3. Find the perimeter using the following equation:

perimeter = 2  (length + width)

4. Find the area using the following equation:

area = length  width

EXAMPLE 1-2

In this example, we design an algorithm that calculates the sales tax and the price of an item sold in a particular state.

The sales tax is calculated as follows: The state’s portion of the sales tax is 4%, and the city’s portion of the sales tax is 1.5%. If the item is a luxury item, such as a car more than $50,000, then there is a 10% luxury tax.

To calculate the price of the item, we need to calculate the state’s portion of the sales tax, the city’s portion of the sales tax, and, if it is a luxury item, the luxury tax. Suppose salePrice denotes the selling price of the item, stateSalesTax denotes the state’s sales tax, citySalesTax denotes the city’s sales tax, luxuryTax denotes the luxury tax, salesTax denotes the total sales tax, and amountDue denotes the final price of the item.

To calculate the sales tax, we must know the selling price of the item and whether the item is a luxury item.

The stateSalesTax and citySalesTax can be calculated using the following formulas: stateSalesTax = salePrice  0.04

citySalesTax = salePrice  0.015

**Watch**

**the Video**

16 | Chapter 1: An Overview of Computers and Programming Languages

Next, you can determine luxuryTax as follows:

if (item is a luxury item)

luxuryTax = salePrice  0.1

otherwise

luxuryTax = 0

Next, you can determine salesTax as follows:

salesTax = stateSalesTax + citySalesTax + luxuryTax

Finally, you can calculate amountDue as follows:

amountDue = salePrice + salesTax

The algorithm to determine salesTax and amountDue is, therefore: 1. Get the selling price of the item.

2. Determine whether the item is a luxury item.

3. Find the state’s portion of the sales tax using the formula:

stateSalesTax = salePrice  0.04

4. Find the city’s portion of the sales tax using the formula:

citySalesTax = salePrice  0.015

5. Find the luxury tax using the following formula:

if (item is a luxury item)

luxuryTax = salePrice  0.1

otherwise

luxuryTax = 0

6. Find salesTax using the formula:

salesTax = stateSalesTax + citySalesTax + luxuryTax

7. Find amountDue using the formula:

amountDue = salePrice + salesTax

EXAMPLE 1-3

In this example, we design an algorithm that calculates the monthly paycheck of a salesperson at a local department store.

Every salesperson has a base salary. The salesperson also receives a bonus at the end of each month, based on the following criteria: If the salesperson has been with the store for five years or less, the bonus is $10 for each year that he or she has worked there. If the salesperson has been with the store for more than five years, the bonus is $20 for each year that he or she has worked there. The salesperson can earn an additional bonus as follows: If the total sales made

Programming with the Problem Analysis–Coding–Execution Cycle | 17

1 by the salesperson for the month are at least $5,000 but less than $10,000, he or she receives a

3% commission on the sale. If the total sales made by the salesperson for the month are at least $10,000, he or she receives a 6% commission on the sale.

To calculate a salesperson’s monthly paycheck, you need to know the base salary, the number of years that the salesperson has been with the company, and the total sales made by the sales person for that month. Suppose baseSalary denotes the base salary, noOfServiceYears denotes the number of years that the salesperson has been with the store, bonus denotes the bonus, totalSales denotes the total sales made by the salesperson for the month, and additionalBonus denotes the additional bonus.

You can determine the bonus as follows:

if (noOfServiceYears is less than or equal to five)

bonus = 10  noOfServiceYears

otherwise

bonus = 20  noOfServiceYears

Next, you can determine the additional bonus of the salesperson as follows:

if (totalSales is less than 5000)

additionalBonus = 0

otherwise

if (totalSales is greater than or equal to 5000 and

totalSales is less than 10000)

additionalBonus = totalSales  (0.03)

otherwise

additionalBonus = totalSales  (0.06)

Following the above discussion, you can now design the algorithm to calculate a salesperson’s monthly paycheck:

1. Get baseSalary.

2. Get noOfServiceYears.

3. Calculate bonus using the following formula:

if (noOfServiceYears is less than or equal to five)

bonus = 10  noOfServiceYears

otherwise

bonus = 20  noOfServiceYears

4. Get totalSales.

5. Calculate additionalBonus using the following formula:

if (totalSales is less than 5000)

additionalBonus = 0

otherwise

if (totalSales is greater than or equal to 5000 and

totalSales is less than 10000)

additionalBonus = totalSales  (0.03)

otherwise

additionalBonus = totalSales  (0.06)

18 | Chapter 1: An Overview of Computers and Programming Languages

6. Calculate payCheck using the equation:

payCheck = baseSalary + bonus + additionalBonus

EXAMPLE 1-4

In this example, we design an algorithm to play a number-guessing game.

The objective is to randomly generate an integer greater than or equal to 0 and less than 100. Then prompt the player (user) to guess the number. If the player guesses the number correctly, output an appropriate message. Otherwise, check whether the guessed number is less than the random number. If the guessed number is less than the random number generated, output the message, ‘‘Your guess is lower than the number. Guess again!’’; otherwise, output the message, ‘‘Your guess is higher than the number. Guess again!’’. Then prompt the player to enter another number. The player is prompted to guess the random number until the player enters the correct number.

The first step is to generate a random number, as described above. C++ provides the means to do so, which is discussed in Chapter 5. Suppose num stands for the random number and guess stands for the number guessed by the player.

After the player enters the guess, you can compare the guess with the random number as follows:

if (guess is equal to num)

Print "You guessed the correct number."

otherwise

if guess is less than num

Print "Your guess is lower than the number. Guess again!" otherwise

Print "Your guess is higher than the number. Guess again!" You can now design an algorithm as follows:

1. Generate a random number and call it num.

2. Repeat the following steps until the player has guessed the correct number: a. Prompt the player to enter guess.

b.

if (guess is equal to num)

Print "You guessed the correct number."

otherwise

if guess is less than num

Print "Your guess is lower than the number. Guess again!"

otherwise

Print "Your guess is higher than the number. Guess again!"

In Chapter 5, we use this algorithm to write a C++ program to play the guessing the number game.

Programming with the Problem Analysis–Coding–Execution Cycle | 19

1

EXAMPLE 1-5

There are 10 students in a class. Each student has taken five tests, and each test is worth 100 points. We want to design an algorithm to calculate the grade for each student, as well as the class average. The grade is assigned as follows: If the average test score is greater than or equal to 90, the grade is A; if the average test score is greater than or equal to 80 and less than 90, the grade is B; if the average test score is greater than or equal to 70 and less than 80, the grade is C; if the average test score is greater than or equal to 60 and less than 70, the grade is D; otherwise, the grade is F. Note that the data consists of students’ names and their test scores.

This is a problem that can be divided into subproblems as follows: There are five tests, so you design an algorithm to find the average test score. Next, you design an algorithm to determine the grade. The two subproblems are to determine the average test score and to determine the grade.

Let us first design an algorithm to determine the average test score. To find the average test score, add the five test scores and then divide the sum by 5. Therefore, the algorithm is the following:

1. Get the five test scores.

2. Add the five test scores. Suppose sum stands for the sum of the test scores. 3. Suppose average stands for the average test score. Then

average = sum / 5;

Next, you design an algorithm to determine the grade. Suppose grade stands for the grade assigned to a student. The following algorithm determines the grade:

if average is greater than or equal to 90

grade = A

otherwise

if average is greater than or equal to 80 and less than 90 grade = B

otherwise

if average is greater than or equal to 70 and less than 80 grade = C

otherwise

if average is greater than or equal to 60 and less than 70 grade = D

otherwise

grade = F

You can use the solutions to these subproblems to design the main algorithm as follows: (Suppose totalAverage stands for the sum of the averages of each student’s test average.)

1. totalAverage = 0;

2. Repeat the following steps for each student in the class:

a. Get student’s name.

b. Use the algorithm as discussed above to find the average test score.

20 | Chapter 1: An Overview of Computers and Programming Languages

c. Use the algorithm as discussed above to find the grade.

d. Update totalAverage by adding the current student’s average test score. 3. Determine the class average as follows:

classAverage = totalAverage / 10

A programming exercise in Chapter 8 asks you to write a C++ program to determine the average test score and grade for each student in a class.

Programming Methodologies

Two popular approaches to programming design are the structured approach and the object-oriented approach, which are outlined below.

Structured Programming

Dividing a problem into smaller subproblems is called structured design. Each subproblem is then analyzed, and a solution is obtained to solve the subproblem. The solutions to all of the subproblems are then combined to solve the overall problem. This process of imple menting a structured design is called structured programming. The structured-design approach is also known as top-down design, bottom-up design, stepwise refinement, and modular programming.

Object-Oriented Programming

Object-oriented design (OOD) is a widely used programming methodology. In OOD, the first step in the problem-solving process is to identify the components called objects, which form the basis of the solution, and to determine how these objects interact with one another. For example, suppose you want to write a program that automates the video rental process for a local video store. The two main objects in this problem are the video and the customer.

After identifying the objects, the next step is to specify for each object the relevant data and possible operations to be performed on that data. For example, for a video object, the data might include:

• movie name

• starring actors

• producer

• production company

• number of copies in stock

Some of the operations on a video object might include:

• checking the name of the movie

• reducing the number of copies in stock by one after a copy is rented • incrementing the number of copies in stock by one after a customer returns a particular video

Programming Methodologies | 21

This illustrates that each object consists of data and operations on that data. An object 1

combines data and operations on the data into a single unit. In OOD, the final program is a collection of interacting objects. A programming language that implements OOD is called an object-oriented programming (OOP) language. You will learn about the many advantages of OOD in later chapters.

Because an object consists of data and operations on that data, before you can design and use objects, you need to learn how to represent data in computer memory, how to manipulate data, and how to implement operations. In Chapter 2, you will learn the basic data types of C++ and discover how to represent and manipulate data in computer memory. Chapter 3 discusses how to input data into a C++ program and output the results generated by a C++ program.

To create operations, you write algorithms and implement them in a programming language. Because a data element in a complex program usually has many operations, to separate operations from each other and to use them effectively and in a convenient manner, you use functions to implement algorithms. After a brief introduction in Chapters 2 and 3, you will learn the details of functions in Chapter 6. Certain algorithms require that a program make decisions, a process called selection. Other algorithms might require certain statements to be repeated until certain conditions are met, a process called repetition. Still other algorithms might require both selection and repetition. You will learn about selection and repetition mechanisms, called control structures, in Chapters 4 and 5. Also, in Chapter 8, using a mechanism called an array, you will learn how to manipulate data when data items are of the same type, such as items in a list of sales figures.

Finally, to work with objects, you need to know how to combine data and operations on the data into a single unit. In C++, the mechanism that allows you to combine data and operations on the data into a single unit is called a class. You will learn how classes work, how to work with classes, and how to create classes in the chapter Classes and Data Abstraction (later in this book).

As you can see, you need to learn quite a few things before working with the OOD methodology. To make this learning easier and more effective, this book purposely divides control structures into two chapters (4 and 5).

For some problems, the structured approach to program design will be very effective. Other problems will be better addressed by OOD. For example, if a problem requires manipulating sets of numbers with mathematical functions, you might use the struc tured design approach and outline the steps required to obtain the solution. The C++ library supplies a wealth of functions that you can use effectively to manipulate numbers. On the other hand, if you want to write a program that would make a candy machine operational, the OOD approach is more effective. C++ was designed especially to implement OOD. Furthermore, OOD works well and is used in conjunction with structured design.

22 | Chapter 1: An Overview of Computers and Programming Languages

Both the structured design and OOD approaches require that you master the basic compo nents of a programming language to be an effective programmer. In Chapters 2 to 8, you will learn the basic components of C++, such as data types, input/output, control structures, user-defined functions, and arrays, required by either type of programming. We illustrate how these concepts work using the structured programming approach. Starting with the chapter Classes and Data Abstraction, we use the OOD approach.

ANSI/ISO Standard C++

The programming language C++ evolved from C and was designed by Bjarne Stroustrup at Bell Laboratories in the early 1980s. From the early 1980s through the early 1990s, several C++ compilers were available. Even though the fundamental features of C++ in all compilers were mostly the same, the C++ language, referred to in this book as Standard C++, was evolving in slightly different ways in different compilers. As a consequence, C++ programs were not always portable from one compiler to another.

To address this problem, in the early 1990s, a joint committee of the American National Standard Institution (ANSI) and International Standard Organization (ISO) was established to standardize the syntax of C++. In mid-1998, ANSI/ISO C++ language standards were approved. Most of today’s compilers comply with these new standards. Over the last several years, the C++ committee met several times to further standardize the syntax of C++. In mid-2010, the second standard of C++ was voted on and approved. The main objective of this standard, referred to as C++0X, or tentatively as C++11, is to make the C++ code cleaner and more effective. For example, the new standard introduces the data type long long to deal with large integers, auto declaration of variables using initialization statements, enhancing the functionality of for loops to effectively work with arrays and containers, and new algorithms. However, not all new features of this new standard have been implemented by all the compilers currently available. In this book, we introduce the new C++ features that we know have been implemented by the well-known compilers and also comment on the ones that will be implemented in the future.

This book focuses on the syntax of C++ as approved by ANSI/ISO, referred to as ANSI/ ISO Standard C++.

QUICK REVIEW

1. A computer is an electronic device capable of performing arithmetic and logical operations.

2. A computer system has two components: hardware and software. 3. The central processing unit (CPU) and the main memory are examples of hardware components.

4. All programs must be brought into main memory before they can be executed. 5. When the power is switched off, everything in main memory is lost.

Quick Review | 23

6. Secondary storage provides permanent storage for information. Hard disks, 1

flash drives, floppy disks, ZIP disks, CD-ROMs, and tapes are examples of secondary storage.

7. Input to the computer is done via an input device. Two common input devices are the keyboard and the mouse.

8. The computer sends its output to an output device, such as the computer screen. 9. Software are programs run by the computer.

10. The operating system monitors the overall activity of the computer and provides services.

11. The most basic language of a computer is a sequence of 0s and 1s called machine language. Every computer directly understands its own machine language.

12. A bit is a binary digit, 0 or 1.

13. A byte is a sequence of eight bits.

14. A sequence of 0s and 1s is referred to as a binary code or a binary number. 15. One kilobyte (KB) is 210 ¼ 1024 bytes; one megabyte (MB) is 220 ¼ 1,048,576 bytes; one gigabyte (GB) is 230 ¼ 1,073,741,824 bytes; one terabyte (TB) is 240 ¼ 1,099,511,627,776 bytes; one petabyte (PB) is 250 ¼ 1,125,899,906,842,624 bytes; one exabyte (EB) is 260 ¼ 1,152,921,504,606,846,976 bytes; and one zettabyte (ZB) is 270 ¼ 1,180,591,620,717,411,303,424 bytes.

16. Assembly language uses easy-to-remember instructions called mnemonics. 17. Assemblers are programs that translate a program written in assembly language into machine language.

18. Compilers are programs that translate a program written in a high-level language into machine code, called object code.

19. A linker links the object code with other programs provided by the integrated development environment (IDE) and used in the program to produce execu table code.

20. Typically, six steps are needed to execute a C++ program: edit, preprocess, compile, link, load, and execute.

21. A loader transfers executable code into main memory.

22. An algorithm is a step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.

23. The problem-solving process has three steps: analyze the problem and design an algorithm, implement the algorithm in a programming language, and maintain the program.

24. Programs written using the structured design approach are easier to understand, easier to test and debug, and easier to modify.

25. In structured design, a problem is divided into smaller subproblems. Each subproblem is solved, and the solutions to all of the subproblems are then combined to solve the problem.

24 | Chapter 1: An Overview of Computers and Programming Languages

26. In object-oriented design (OOD), a program is a collection of interacting objects. 27. An object consists of data and operations on that data.

28. The ANSI/ISO Standard C++ syntax was approved in mid-1998.

EXERCISES

1. Mark the following statements as true or false.

a. The first device known to carry out calculations was the Pascaline. b. Modern-day computers can accept spoken-word instructions but cannot imitate human reasoning.

c. In ASCII coding, every character is coded as a sequence of 8 bits. d. A compiler translates a high-level program into assembly language. e. The arithmetic operations are performed inside the CPU, and if an error is found, it outputs the logical errors.

f. A sequence of 0s and 1s is called a decimal code.

g. A linker links and loads the object code from main memory into the CPU for execution.

h. Development of a C++ program includes six steps.

i. A program written in a high-level programming language is called a source program.

j. ZB stands for zero byte.

k. The first step in the problem-solving process is to analyze the problem. l. In object-oriented design, a program is a collection of interacting functions. 2. What are the basic commands performed by a computer?

3. Name three hardware components.

4. Why is secondary storage needed?

5. What is the function of an operating system?

6. What are the two types of programs?

7. What are the differences between machine languages and high-level languages? 8. What is a source program?

9. Why do you need a compiler?

10. What kind of errors are reported by a compiler?

11. Why do you need to translate a program written in a high-level language into machine language?

12. Why would you prefer to write a program in a high-level language rather than a machine language?

13. What is linking?

Exercises | 25

14. What are the advantages of problem analysis and algorithm design over directly writing a 1

program in a high-level language?

15. Design an algorithm to find the weighted average of four test scores. The four test scores and their respective weights are given in the following format:

testScore1 weightTestScore1

...

For example, sample data is as follows:

75 0.20

95 0.35

85 0.15

65 0.30

16. Design an algorithm to convert the change given in quarters, dimes, nickels, and pennies into pennies.

17. Given the radius, in inches, and price of a pizza, design an algorithm to find the price of the pizza per square inch.

18. To make a profit, the prices of the items sold in a furniture store are marked up by 80%. After marking up the prices each item is put on sale at a discount of 10%. Design an algorithm to find the selling price of an item sold at the furniture store. What information do you need to find the selling price?

19. Suppose a, b, and c denote the lengths of the sides of a triangle. Then the area of the triangle can be calculated using the formula:

sðs aÞðs bÞðs cÞ p ;

ffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffiffi

where s = (1/2)(a + b + c). Design an algorithm that uses this formula to find the area of a triangle. What information do you need to find the area?

20. Jason typically uses the Internet to buy various items. If the total cost of the items ordered, at one time, is $200 or more, then the shipping and handling is free; otherwise, the shipping and handling is $10 per item. Design an algorithm that prompts Jason to enter the number of items ordered and the price of each item. The algorithm then outputs the total billing amount. Your algorithm must use a loop (repetition structure) to get the price of each item. (For simplicity, you may assume that Jason orders no more than five items at a time.)

21. Suppose that the cost of sending an international fax is calculated as follows: The service charge is $3.00, $.20 per page for the first 10 pages, and $0.10 for each additional page. Design an algorithm that asks the user to enter the number of pages to be faxed. The algorithm then uses the number of pages to be faxed to calculate the amount due.

22. An ATM allows a customer to withdraw a maximum of $500 per day. If a customer withdraws more than $300, the service charge is 4% of the amount over $300. If the customer does not have sufficient money in the account, the ATM informs the customer about the insufficient funds and gives the customer

26 | Chapter 1: An Overview of Computers and Programming Languages

the option to withdraw the money for a service charge of $25.00. If there is no money in the account or if the account balance is negative, the ATM does not allow the customer to withdraw any money. If the amount to be withdrawn is greater than $500, the ATM informs the customer about the maximum amount that can be withdrawn. Write an algorithm that allows the customer to enter the amount to be withdrawn. The algorithm then checks the total amount in the account, dispenses the money to the customer, and debits the account by the amount withdrawn and the service charges, if any.

23. You are given a list of students’ names and their test scores. Design an algorithm that does the following:

a. Calculates the average test scores.

b. Determines and prints the names of all the students whose test scores are below the average test score.

c. Determines the highest test score.

d. Prints the names of all the students whose test scores are the same as the highest test score.

(You must divide this problem into subproblems as follows: The first subproblem determines the average test score. The second subproblem determines and prints the names of all the students whose test scores are below the average test score. The third subproblem determines the highest test score. The fourth subproblem prints the names of all the students whose test scores are the same as the highest test score. The main algorithm combines the solutions of the subproblems.)

CHAPTER 2

BASIC ELEMENTS OF C++

IN THIS CHAPTER, YOU WILL:

. Become familiar with the basic components of a C++ program, including functions, special symbols, and identifiers

. Explore simple data types

. Discover how to use arithmetic operators

. Examine how a program evaluates arithmetic expressions

. Learn what an assignment statement is and what it does

. Become familiar with the string data type

. Discover how to input data into memory using input statements

. Become familiar with the use of increment and decrement operators

. Examine ways to output results using output statements

. Learn how to use preprocessor directives and why they are necessary

. Learn how to debug syntax errors

. Explore how to properly structure a program, including using comments to document a program . Learn how to write a C++ program

28 | Chapter 2: Basic Elements of C++

In this chapter, you will learn the basics of C++. As your objective is to learn the C++ programming language, two questions naturally arise. First, what is a computer program? Second, what is programming? A computer program, or a program, is a sequence of statements whose objective is to accomplish a task. Programming is a process of planning and creating a program. These two definitions tell the truth, but not the whole truth, about programming. It may very well take an entire book to give a good and satisfactory definition of programming. You might gain a better grasp of the nature of programming from an analogy, so let us turn to a topic about which almost everyone has some knowledge—cooking. A recipe is also a program, and everyone with some cooking experience can agree on the following:

1. It is usually easier to follow a recipe than to create one.

2. There are good recipes and there are bad recipes.

3. Some recipes are easy to follow and some are not easy to follow. 4. Some recipes produce reliable results and some do not.

5. You must have some knowledge of how to use cooking tools to follow a recipe to completion.

6. To create good new recipes, you must have a lot of knowledge and a good understanding of cooking.

These same six points are also true about programming. Let us take the cooking analogy one step further. Suppose you need to teach someone how to become a chef. How would you go about it? Would you first introduce the person to good food, hoping that a taste for good food develops? Would you have the person follow recipe after recipe in the hope that some of it rubs off? Or would you first teach the use of tools and the nature of ingredients, the foods and spices, and explain how they fit together? Just as there is disagreement about how to teach cooking, there is disagreement about how to teach programming.

Learning a programming language is like learning to become a chef or learning to play a musical instrument. All three require direct interaction with the tools. You cannot become a good chef or even a poor chef just by reading recipes. Similarly, you cannot become a player by reading books about musical instruments. The same is true of programming. You must have a fundamental knowledge of the language, and you must test your programs on the computer to make sure that each program does what it is supposed to do.

A Quick Look at a C++ Program

In this chapter, you will learn the basic elements and concepts of the C++ programming language to create C++ programs. In addition to giving examples to illustrate various concepts, we will also show C++ programs to clarify these concepts. In this section, we provide an example of a C++ program that computes the perimeter and area of a

A Quick Look at a C++ Program | 29

rectangle. At this point you need not be too concerned with the details of this program. You only need to know the effect of an output statement, which is introduced in this program.

2

In Example 1-1 (Chapter 1), we designed an algorithm to find the perimeter and area of a rectangle. Given the length and width of a rectangle, the C++ program, in Example 2-1, computes and displays the perimeter and area.

EXAMPLE 2-1

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Given the length and width of a rectangle, this C++ program // computes and outputs the perimeter and area of the rectangle. //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <iostream>

using namespace std;

int main()

{

double length;

double width;

double area;

double perimeter;

cout << "Program to compute and output the perimeter and " << "area of a rectangle." << endl;

length = 6.0;

width = 4.0;

perimeter = 2 \* (length + width);

area = length \* width;

cout << "Length = " << length << endl;

cout << "Width = " << width << endl;

cout << "Perimeter = " << perimeter << endl;

cout << "Area = " << area << endl;

return 0;

}

Sample Run: (When you compile and execute this program, the following five lines are displayed on the screen.)

Program to compute and output the perimeter and area of a rectangle. Length = 6

Width = 4

Perimeter = 20

Area = 24

30 | Chapter 2: Basic Elements of C++

These lines are displayed by the execution of the following statements:

cout << "Program to compute and output the perimeter and " << "area of a rectangle." << endl;

cout << "Length = " << length << endl;

cout << "Width = " << width << endl;

cout << "Perimeter = " << perimeter << endl;

cout << "Area = " << area << endl;

Next we explain how this happens. Let us first consider the following statement:

cout << "Program to compute and output the perimeter and " << "area of a rectangle." << endl;

This is an example of a C++ output statement. It causes the computer to evaluate the expression after the pair of symbols << and display the result on the screen.

Usually, a C++ program contains various types of expressions such as arithmetic and strings. For example, length + width is an arithmetic expression. Anything in double quotes is a string. For example, "Program to compute and output the perimeter and " is a string. Similarly, "area of a rectangle." is also a string. Typically, a string evaluates to itself. Arithmetic expressions are evaluated according to rules of arithmetic operations, which you typically learn in an algebra course. Later in this chapter, we will explain how arithmetic expressions and strings are formed and evaluated.

Also note that in an output statement, endl causes the insertion point to move to the beginning of the next line. (Note that in endl, the last letter is lowercase el. Also, on the screen, the insertion point is where the cursor is.) Therefore, the preceding statement causes the system to display the following line on the screen.

Program to compute and output the area and perimeter of a rectangle. Let us now consider the following statement:

cout << "Length = " << length << endl;

This output statement consists of two expressions. The first expression, (after the first <<), is "Length = " and the second expression, (after the second <<), consists of the identifier length. The expression "Length = " is a string and evaluates to itself. (Notice the space after =.) The second expression, which consists of the identifier length, evaluates to whatever the value of length is. Because the value assigned to length is 6.0, length evaluates to 6.0. Therefore, the output of the preceding statement is:

Length = 6

Note that the value of length is output as 6 not as 6.0. We will explain in the next chapter how to force the program to output the value of length as 6.0. The meaning of the remaining output statements is similar.

The last statement, that is, return 0;

A Quick Look at a C++ Program | 31

returns the value 0 to the operating system when the program terminates. We will 2

elaborate on this statement later in this chapter.

Before we identify various parts of a C++ program, let’s look at one more output statement. Consider the following statement:

cout << "7 + 8 = " << 7 + 8 << endl;

In this output statement, the expression "7 + 8 = ", which is a string, evaluates to itself. Let us consider the second expression, 7 + 8. This expression consists of the numbers 7 and 8, and the C++ arithmetic operator +. Therefore, the result of the expression 7 + 8 is the sum of 7 and 8, which is 15. Thus, the output of the preceding statement is:

7 + 8 = 15

In the next chapter, until we explain how to properly construct a C++ program, we will be using output statements such as the preceding ones to explain various concepts. After finishing Chapter 2, you should be able to write C++ programs well enough to do some computations and show results.

Next, let us note the following about the previous C++ program. A C++ program is a collection of functions, one of which is the function main. Roughly speaking, a function is a set of statements whose objective is to accomplish something. The preceding program consists of only the function main; all C++ programs require a function main.

The first four lines of the program begins with the pair of symbols // (shown in green), which are comments. Comments are for the user; they typically explain the purpose of the programs, that is, the meaning of the statements. (We will elaborate on how to include comments in a program later in this chapter.) The next line of the program, that is,

#include <iostream>

allows us to use the (predefined object) cout to generate output and the (manipulator) endl. The statement

using namespace std;

allows you to use cout and endl without the prefix std::. It means that if you do not include this statement, then cout should be used as std::cout and endl should be used as std::endl. We will elaborate on this later in this chapter.

Next consider the following line:

int main()

32 | Chapter 2: Basic Elements of C++

This is the heading of the function main. The next line consists of a left brace. This marks the beginning of the (body) of the function main. The right brace (at the last line of the program) matches this left brace and marks the end of the body of the function main. We will explain the meaning of the other terms, such as the ones shown in blue, later in this book. Note that in C++, << is an operator, called the stream insertion operator.

Before ending this section, let us identify certain parts of the C++ program in Figure 2-1.

**//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Given the length and width of a rectangle, this C++ program // computes and outputs the perimeter and area of the rectangle.**

**//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* #include** <iostream>

**using namespace** std;

**int** main()

Comments

{

**double** length; **double** width; **double** area; **double** perimeter;

Variable declarations. A statement such as **double** length;

instructs the system to allocate memory space and name it length.

cout << "Program to compute and output the perimeter and " << "area of a rectangle." << endl;

length = 6.0; width = 4.0;

Assignment statement. This statement instructs the system to store 6.0 in the memory space length.

perimeter = 2 \* (length + width);

area = length \* width;

Assignment statement.

This statement instructs the system to evaluate the expression length \* width and store the result in the memory space area.

cout << "Length = " << length << endl; cout << "Width = " << width << endl; cout << "Perimeter = " << perimeter << endl; cout << "Area = " << area << endl;

**return** 0;

}

FIGURE 2-1 Various parts of a C++ program

Output statements. An output statement

instructs the system to display results.

A Quick Look at a C++ Program | 33

One of the terms that you will encounter throughout the text and that is also identified in Figure 2-1 is variable. Therefore, we introduce this term in this section. Recall from Chapter 1 that all data must be loaded into main memory before it can be manipulated. For example, given the length and width, the program in Figure 2-1 computes and

2

outputs the area and perimeter of a rectangle. This means that the values of length and width must be stored in main memory. Also, recall from Chapter 1 that main memory is an ordered sequence of cells and every cell has an address. Inside the computer, the address of a memory cell is in binary. Once we store the values of length and width, and because these values might be needed in more than one place in a program, we would like to know the locations where these values are stored and how to access those memory locations. C++ makes it easy for a programmer to specify the locations because the programmer can supply an alphabetic name for each of those locations. Of course, we must follow the rules to specify the names. For example, in the program in Figure 2-1, we are telling the system to allocate four memory spaces and name them length, width, area, and perimeter, respectively. (We will explain the meaning of the word double, shown in blue later in this chapter.) Essentially, a variable is a memory location whose contents can be changed. So length, width, area, and perimeter are variables. Also during program execution, the system will allocate four memory locations large enough to store decimal numbers and those memory locations will be named length, width, area, and perimeter, respectively, see Figure 2-2.

length width area perimeter

FIGURE 2-2 Memory allocation

The statement length = 6.0; will cause the system to store 6.0 in the memory location length, see Figure 2-3. Examples 2-14 and 2-19 further illustrate how data is manipulated in variables.

6.0

length width area perimeter

FIGURE 2-3 Memory spaces after the statement length = 6.0; executes

As we proceed through this chapter, we will explain the meaning of the remaining parts identified in Figure 2-1.

34 | Chapter 2: Basic Elements of C++

The Basics of a C++ Program

In the previous section, we gave an example of a C++ program and also identified certain parts of the program. In general, a C++ program is a collection of one or more subprograms, called functions. Roughly speaking, a subprogram or a function is a collection of statements, and when it is activated, or executed, it accomplishes something. Some functions, called predefined or standard functions, are already written and are provided as part of the system. But to accomplish most tasks, programmers must learn to write their own functions.

Every C++ program has a function called main. Thus, if a C++ program has only one function, it must be the function main. Until Chapter 6, other than using some of the predefined functions, you will mainly deal with the function main. By the end of this chapter, you will have learned how to write the function main.

If you have never seen a program written in a programming language, the C++ program in Example 2-1 may look like a foreign language. To make meaningful sentences in a foreign language, you must learn its alphabet, words, and grammar. The same is true of a program ming language. To write meaningful programs, you must learn the programming language’s special symbols, words, and syntax rules. The syntax rules tell you which statements (instructions) are legal or valid, that is, which are accepted by the programming language and which are not. You must also learn semantic rules, which determine the meaning of the instructions. The programming language’s rules, symbols, and special words enable you to write programs to solve problems.

Programming language: A set of rules, symbols, and special words.

In the remainder of this section, you will learn about some of the special symbols of a C++ program. Additional special symbols are introduced as other concepts are encoun tered in later chapters. Similarly, syntax and semantic rules are introduced and discussed throughout the book.

Comments

The program that you write should be clear not only to you, but also to the reader of your program. Part of good programming is the inclusion of comments in the program. Typically, comments can be used to identify the authors of the program, give the date when the program is written or modified, give a brief explanation of the program, and explain the meaning of key statements in a program. In the programming examples, for the programs that we write, we will not include the date when the program is written, consistent with the standard convention for writing such books.

Comments are for the reader, not for the compiler. So when a compiler compiles a program to check for the syntax errors, it completely ignores comments. Throughout this book, comments are shown in green.

The Basics of a C++ Program | 35

The program in Example 2-1 contains the following comments:

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Given the length and width of a rectangle, this C++ program // computes and outputs the perimeter and area of the rectangle.

2

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

There are two common types of comments in a C++ program—single-line comments and multiple-line comments.

Single-line comments begin with // and can be placed anywhere in the line. Everything encountered in that line after // is ignored by the compiler. For example, consider the following statement:

cout << "7 + 8 = " << 7 + 8 << endl;

You can put comments at the end of this line as follows:

cout << "7 + 8 = " << 7 + 8 << endl; //prints: 7 + 8 = 15 This comment could be meaningful for a beginning programmer.

Multiple-line comments are enclosed between /\* and \*/. The compiler ignores anything that appears between /\* and \*/. For example, the following is an example of a multiple-line comment:

/\*

You can include comments that can

\*/

occupy several lines.

In multiple-line comments, many programmers use single-line comments on every line to make the comments stand out more to the reader (as was done in the program in Example 2-1.)

Special Symbols

The smallest individual unit of a program written in any language is called a token. C++’s tokens are divided into special symbols, word symbols, and identifiers.

Following are some of the special symbols:

+ - \* /

. ; ? ,

<= != == >=

The first row includes mathematical symbols for addition, subtraction, multiplication, and division. The second row consists of punctuation marks taken from English grammar. Note that the comma is also a special symbol. In C++, commas are used to separate items in a list. Semicolons are also special symbols and are used to end a C++ statement. Note that a blank, which is not shown above, is also a special symbol. You create a blank symbol by pressing the space bar (only once) on the keyboard. The third row consists of

36 | Chapter 2: Basic Elements of C++

tokens made up of two characters that are regarded as a single symbol. No character can come between the two characters in the token, not even a blank.

Reserved Words (Keywords)

A second category of tokens is reserved word symbols. Some of the reserved word symbols include the following:

int, float, double, char, const, void, return

Reserved words are also called keywords. The letters that make up a reserved word are always lowercase. Like the special symbols, each is considered to be a single symbol. Furthermore, word symbols cannot be redefined within any program; that is, they cannot be used for anything other than their intended use. For a complete list of reserved words, see Appendix A.

Throughout this book, reserved words are shown in blue.

Identifiers

A third category of tokens is identifiers. Identifiers are names of things that appear in programs, such as variables, constants, and functions. All identifiers must obey C++’s rules for identifiers.

Identifier: A C++ identifier consists of letters, digits, and the underscore character (\_ ) and must begin with a letter or underscore.

Some identifiers are predefined; others are defined by the user. In the C++ program in Example 2-1, cout is a predefined identifier and length is a user-defined identifier. Two predefined identifiers that you will encounter frequently are cout and cin. You have already seen the effect of cout. Later in this chapter, you will learn how cin, which is used to input data, works. Unlike reserved words, predefined identifiers can be redefined, but it would not be wise to do so.

Identifiers can be made of only letters, digits, and the underscore character; no other symbols are permitted to form an identifier.

C++ is case sensitive—uppercase and lowercase letters are considered different. Thus, the identifier NUMBER is not the same as the identifier number. Similarly, the identifiers X and x are different.