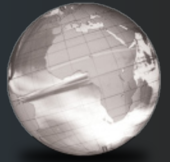
GLOBAL EDITION



Absolute C++

SIXTH EDITION

Walter Savitch

ALWAYS LEARNING

PEARSON

ABSOLUTE C++

6TH EDITION

GLOBAL EDITION

This page intentionally left blank

ABSOLUTE C++

6TH EDITION
GLOBAL EDITION

Walter Savitch
University of California, San Diego

Contributor
Kenrick Mock
University of Alaska Anchorage

PEARSON

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

Vice President and Editorial Director, ECS: Marcia J. Horton
Acquisitions Editor: Matt Goldstein
Editorial Assistant: Kelsey Loanes
Assistant Acquisitons Editor, Global Editions: Aditee Agarwal
Product Marketing Manager: Bram Van Kempen
Marketing Assistant: Jon Bryant
Senior Managing Editor: Scott Disanno
Production Project Manager: Rose Kernan
Program Manager: Carole Snyder
Project Editor, Global Editions: K.K. Neelakantan
Senior Manufacturing Controller, Global Editions: Kay Holman
Media Production Manager, Global Editions: Vikram Kumar
Global HE Director of Vendor Sourcing and Procurement: Diane Hynes
Director of Operations: Nick Sklitsis
Operations Specialist: Maura Zaldivar-Garcia
Cover Designer: Lumina Datamatics
Manager, Rights and Permissions: Rachel Youdelman
Associate Project Manager, Rights and Permissions: Timothy Nicholls
Full-Service Project Management: Niraj Bhatt, iEnergizer Aptara®, Ltd.
Cover Image: narak0rn/Shutterstock

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

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

© Pearson Education Limited 2016

The right of Walter Savitch to be identified as the author of this work has been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

Authorized adaptation from the United States edition, entitled Absolute C++, 6th Edition, ISBN 978-0-13-397078-4, by Walter Savitch published by Pearson Education © 2016.

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

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

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

10 9 8 7 6 5 4 3 2 1

ISBN 10: 1-292-09859-7
ISBN 13: 978-1-292-09859-3

Typeset in 10.5/12 Adobe Garamond by iEnergizer Aptara®, Ltd.

Printed and bound in Malaysia.

Preface

This book is designed to be a textbook and reference for programming in the C++ language. Although it does include programming techniques, it is organized around the features of the C++ language, rather than any particular curriculum of techniques. The main audience I had in mind is undergraduate students who had not had extensive programming experience with the C++ language. As such, this book is a suitable C++ text or reference for a wide range of users. The introductory chapters are written at a level that is accessible to beginners, while the boxed sections of those chapters serve to introduce more experienced programmers to basic C++ syntax. Later chapters are also understandable to beginners, but are written at a level suitable for students who have progressed to these more advanced topics. *Absolute C++* is also suitable for anyone learning the C++ language on their own. (For those who want a textbook with more pedagogical material and more on very basic programming technique, try my text *Problem Solving with C++*, Eighth Edition, Pearson Education.)

The C++ coverage in this book goes well beyond what a beginner needs to know. In particular, it has extensive coverage of inheritance, polymorphism, exception handling, and the Standard Template Library (STL), as well as basic coverage of patterns and the unified modeling language (UML).

CHANGES IN THIS EDITION

This sixth edition presents the same programming philosophy as the fifth edition. For instructors, you can teach the same course, presenting the same topics in the same order with no changes in the material covered or the chapters assigned. Changes include:

- Introduction to C++11 in the context of C++98. Examples of C++11 content include new integer types, the auto type, raw string literals, strong enumerations, nullptr, ranged for loop, conversion between strings and integers, member initializers, and constructor delegation.
- Additional material on sorting, the Standard Template Library, iterators, and exception handling.
- New appendix introducing the std::array class, regular expressions, threads, and smart pointers
- Correction of errata.
- Fifteen new Programming Projects.
- Five new VideoNotes for a total of sixty-nine VideoNotes. These VideoNotes walk students through the process of both problem solving and coding to help reinforce key programming concepts. An icon appears in the margin of the book when a VideoNote is available regarding the topic covered in the text.

ANSI/ISO C++ STANDARD

This edition is fully compatible with compilers that meet the latest ANSI/ISO C++ standard.

STANDARD TEMPLATE LIBRARY

The Standard Template Library (STL) is an extensive collection of preprogrammed data structure classes and algorithms. The STL is perhaps as big a topic as the core C++ language, so I have included a substantial introduction to STL. There is a full chapter on the general topic of templates and a full chapter on the particulars of STL, as well as other material on, or related to, STL at other points in the text.

OBJECT-ORIENTED PROGRAMMING

This book is organized around the structure of C++. As such, the early chapters cover aspects of C++ that are common to most high-level programming languages but are not particularly oriented toward object-oriented programming (OOP). For a reference book—and for a book for learning a second language—this makes sense. However, I consider C++ to be an OOP language. If you are programming in C++ and not C, you must be using the OOP features of C++. This text offers extensive coverage of encapsulation, inheritance, and polymorphism as realized in the C++ language. Chapter 20, on patterns and UML, gives additional coverage of OOP-related material.

FLEXIBILITY IN TOPIC ORDERING

This book allows instructors wide latitude in reordering the material. This is important if a book is to serve as a reference. This is also in keeping with my philosophy of accommodating the instructor's style, rather than tying the instructor to my own personal preference of topic ordering. Each chapter introduction explains what material must already have been covered before each section of the chapter can be covered.

ACCESSIBLE TO STUDENTS

It is not enough for a book to present the right topics in the right order. It is not even enough for it be correct and clear to an instructor. The material also needs to be presented in a way that is accessible to the novice. Like my other textbooks, which proved to be very popular with students, this book was written to be friendly and accessible to the student.

SUMMARY BOXES

Each major point is summarized in a boxed section. These boxed sections are spread throughout each chapter. They serve as summaries of the material, as a quick reference source, and as a quick way to learn the C++ syntax for a feature you know about in general but for which you do not know the C++ particulars.

SELF-TEST EXERCISES

Each chapter contains numerous self-test exercises. Complete answers for all the self-test exercises are given at the end of each chapter.

VIDEO NOTES

VideoNotes are step-by-step videos that guide readers through the solution to an end of chapter problem or further illuminate a concept presented in the text. Icons in the text indicate where a VideoNote enhances a topic. Fully navigable problems allow for self-paced instruction. VideoNotes are located at www.pearsonglobaleditions.com/savitch.

OTHER FEATURES

Pitfall sections, programming technique sections, and examples of complete programs with sample input and output are given throughout each chapter. Each chapter ends with a summary and a collection of programming projects.

SUPPORT MATERIAL

The following support materials are available to all users of this book at www.pearsonglobaleditions.com/savitch:

- Source code from the book

The following resources are available to qualified instructors only at www.pearsonglobaleditions.com/savitch.

- Instructor's Manual with Solutions
- PowerPoint® slides

HOW TO ACCESS INSTRUCTOR AND STUDENT RESOURCE MATERIALS

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

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

For a full demonstration, to see feedback from instructors and students, or to get started using MyProgrammingLab in your course, visit www.myprogramminglab.com.

ACKNOWLEDGMENTS

Numerous individuals have contributed invaluable help and support to making this book happen. Frank Ruggirello and Susan Hartman at Addison-Wesley first conceived the idea and supported the first edition, for which I owe them a debt of gratitude. A

special thanks to Matt Goldstein who was the editor for the second, third, and fourth editions. His help and support were critical to making this project succeed. Chelsea Kharakozova, Marilyn Lloyd, Yez Alayan, and the other fine people at Pearson Education also provided valuable support and encouragement.

The following reviewers provided suggestions for the book. I thank them all for their hard work and helpful comments.

Duncan Buell	University of South Carolina
Daniel Cliburn	University of the Pacific
Natacha Gueroguieva	College of Staten Island, CUNY
Archana Sharma Gupta	Delaware Technical and Community College
Kenneth Moore	Community College of Allegheny County
Robert Meyers	Florida State University
Clayton Price	Missouri University of Science and Technology
Terry Rooker	Oregon State University
William Smith	Tulsa Community College
Hugh Lauer	Worcester Polytechnic Institute
Richard Albright	University of Delaware
J. Boyd Trolinger	Butte College
Jerry K. Bilbrey, Jr	Francis Marion University
Albert M. K. Cheng	University of Houston
David Cherba	Michigan State University
Fredrick H. Colclough	Colorado Technical University
Drue Coles	Boston University
Stephen Corbesero	Moravian College
Christopher E. Cramer	
Ron DiNapoli	Cornell University
Qin Ding	Pennsylvania State University, Harrisburg
Martin Dulberg	North Carolina State University
H. E. Dunsmore	Purdue University
Evan Golub	University of Maryland
Terry Harvey	University of Delaware
Joanna Klukowska	Hunter College, CUNY
Lawrence S. Kroll	San Francisco State University
Stephen P. Leach	Florida State University
Alvin S. Lim	Auburn University
Tim H. Lin	Cal Poly Pomona
R. M. Lowe	Clemson University
Jeffrey L. Popyack	Drexel University

Amar Raheja	Cal Poly Pomona
Victoria Rayskin	University of Central Los Angeles
Loren Rhodes	Juniata College
Jeff Ringenberg	University of Michigan
Victor Shtern	Boston University
Aaron Striegel	University of Notre Dame
J. Boyd Trolinger	Butte College
Chrysafis Vogiatzis	University of Florida
Joel Weinstein	Northeastern University
Dick Whalen	College of Southern Maryland

A special thanks goes to Kenrick Mock (University of Alaska Anchorage) who executed the updating of this edition. He once again had the difficult job of satisfying me, the editor, and himself. I thank him for a truly excellent job.

Walter Savitch

Pearson would like to thank and acknowledge Muthuraj M, for contributing to the Global Edition, and Ela Kashyap, Amity University, Sandeep Singh, Jaypee Institute of Information Technology, Vikas Saxena, Jaypee Institute of Information Technology, and Piyali Bandopadhyay for reviewing the Global Edition.

LOCATION OF VIDEONOTES IN THE TEXT

www.pearsonglobaleditions.com/savitch

Chapter 1	Compiling and Running a C++ Program, page 32 C++11 Fixed Width Integer Types, page 38 Solution to Programming Project 1.11, page 73
Chapter 2	Nested Loop Example, page 113 Solution to Programming Project 2.5, page 124 Solution to Programming Project 2.9, page 125 Solution to Programming Project 2.10, page 126
Chapter 3	Generating Random Numbers, page 137 Scope Walkthrough, page 155 Solution to Programming Project 3.9, page 170
Chapter 4	Using an Integrated Debugger, page 203 Solution to Programming Project 4.4, page 210 Solution to Programming Project 4.11, page 211
Chapter 5	Array Walkthrough, page 217 Range-Based `for` Loop, page 222 Bubble Sort Walkthrough, page 247 Solution to Programming Project 5.7, page 266 Solution to Programming Project 5.15, page 270
Chapter 6	Solution to Programming Project 6.5, page 306 Solution to Programming Project 6.9, page 307
Chapter 7	Constructor Walkthrough, page 310 Default Initialization of Member Variables, page 329 Solution to Programming Project 7.4, page 351 Solution to Programming Project 7.7, page 352
Chapter 8	Solution to Programming Project 8.7, page 400
Chapter 9	Using `cin` and `getline` with the `string` class, page 433 Solution to Programming Project 9.11, page 451 Solution to Programming Project 9.13, page 452
Chapter 10	Example of Shallow Copy vs. Deep Copy, page 493 Solution to Programming Project 10.5, page 503
Chapter 11	Avoiding Multiple Definitions with `#ifndef`, page 518 Solution to Programming Project 11.5, page 546
Chapter 12	Walkthrough of the `stringstream` demo, page 588 Solution to Programming Project 12.17, page 600 Solution to Programming Project 12.25, page 604
Chapter 13	Recursion and the Stack, page 616 Walkthrough of Mutual Recursion, page 625 Solution to Programming Project 13.9, page 643 Solution to Programming Project 13.11, page 644
Chapter 14	Solution to Programming Project 14.7, page 691

Chapter 15	Solution to Programming Project 15.5, page 726 Solution to Programming Project 15.7, page 727
Chapter 16	Solution to Programming Project 16.3, page 764 Solution to Programming Project 16.7, page 765
Chapter 17	Solution to Programming Project 17.5, page 856 Solution to Programming Project 17.11, page 858
Chapter 18	Solution to Programming Project 18.5, page 893
Chapter 19	C++11 and Containers, page 927 Solution to Programming Project 19.9, page 948 Solution to Programming Project 19.12, page 949

This page intentionally left blank

Brief Contents

Chapter 1 C++ BASICS 29
Chapter 2 FLOW OF CONTROL 75
Chapter 3 FUNCTION BASICS 129
Chapter 4 PARAMETERS AND OVERLOADING 175
Chapter 5 ARRAYS 215
Chapter 6 STRUCTURES AND CLASSES 273
Chapter 7 CONSTRUCTORS AND OTHER TOOLS 309
Chapter 8 OPERATOR OVERLOADING, FRIENDS, AND REFERENCES 355
Chapter 9 STRINGS 401
Chapter 10 POINTERS AND DYNAMIC ARRAYS 453
Chapter 11 SEPARATE COMPILATION AND NAMESPACES 505
Chapter 12 STREAMS AND FILE I/O 549
Chapter 13 RECURSION 605
Chapter 14 INHERITANCE 647
Chapter 15 POLYMORPHISM AND VIRTUAL FUNCTIONS 697
Chapter 16 TEMPLATES 729
Chapter 17 LINKED DATA STRUCTURES 767
Chapter 18 EXCEPTION HANDLING 861
Chapter 19 STANDARD TEMPLATE LIBRARY 895
Chapter 20 PATTERNS AND UML (online at www.pearsonglobaleditions.com/savitch)
Appendix 1 C++ KEYWORDS AP-1
Appendix 2 PRECEDENCE OF OPERATORS AP-3
Appendix 3 THE ASCII CHARACTER SET AP-5
Appendix 4 SOME LIBRARY FUNCTIONS AP-7
Appendix 5 OLD AND NEW HEADER FILES AP-15
Appendix 6 ADDITIONAL C++11 LANGUAGE FEATURES AP-17
INDEX I-1

This page intentionally left blank

Contents

Chapter 1 C++ Basics 29

1.1 INTRODUCTION TO C++ 30

Origins of the C++ Language 30
C++ and Object-Oriented Programming 31
The Character of C++ 31
C++ Terminology 32
A Sample C++ Program 32
TIP: Compiling C++ Programs 33

1.2 VARIABLES, EXPRESSIONS, AND ASSIGNMENT STATEMENTS 34

Identifiers 35
Variables 36
Assignment Statements 39
Introduction to the `string` class 40
PITFALL: Uninitialized Variables 41
TIP: Use Meaningful Names 42
More Assignment Statements 42
Assignment Compatibility 43
Literals 44
Escape Sequences 46
Raw String Literals 47
Naming Constants 47
Arithmetic Operators and Expressions 48
Integer and Floating-Point Division 50
PITFALL: Division with Whole Numbers 51
Type Casting 52
Increment and Decrement Operators 54
PITFALL: Order of Evaluation 56

1.3 CONSOLE INPUT/OUTPUT 57

Output Using `cout` 57
New Lines in Output 58
TIP: End Each Program with `\n` or `endl` 59
Formatting for Numbers with a Decimal Point 59
Output with `cerr` 61
Input Using `cin` 61
TIP: Line Breaks in I/O 64

1.4 PROGRAM STYLE 65

Comments 65

1.5 LIBRARIES AND NAMESPACES 66
Libraries and include Directives 66
Namespaces 66
PITFALL: Problems with Library Names 67

Chapter Summary 68
Answers to Self-Test Exercises 69
Programming Projects 71

Chapter 2 Flow of Control 75

2.1 BOOLEAN EXPRESSIONS 76
Building Boolean Expressions 76
PITFALL: Strings of Inequalities 77
Evaluating Boolean Expressions 78
Precedence Rules 80
PITFALL: Integer Values Can Be Used as Boolean Values 84

2.2 BRANCHING MECHANISMS 86
if-else Statements 86
Compound Statements 88
PITFALL: Using = in Place of = = 89
Omitting the else 91
Nested Statements 91
Multiway if-else Statement 91
The switch Statement 92
PITFALL: Forgetting a break in a switch Statement 95
TIP: Use switch Statements for Menus 95
Enumeration Types 95
The Conditional Operator 97

2.3 LOOPS 97
The while and do-while Statements 98
Increment and Decrement Operators Revisited 101
The Comma Operator 102
The for Statement 104
TIP: Repeat-*N*-Times Loops 106
PITFALL: Extra Semicolon in a for Statement 107
PITFALL: Infinite Loops 107
The break and continue Statements 110
Nested Loops 113

2.4 INTRODUCTION TO FILE INPUT 113
Reading From a Text File Using `ifstream` 114

Chapter Summary 117
Answers to Self-Test Exercises 117
Programming Projects 123

Chapter 3 Function Basics 129

3.1 PREDEFINED FUNCTIONS 130
Predefined Functions That Return a Value 130
Predefined `void` Functions 135
A Random Number Generator 137

3.2 PROGRAMMER-DEFINED FUNCTIONS 141
Defining Functions That Return a Value 142
Alternate Form for Function Declarations 144
PITFALL: Arguments in the Wrong Order 145
PITFALL: Use of the Terms *Parameter* and *Argument* 145
Functions Calling Functions 145
EXAMPLE: A Rounding Function 145
Functions That Return a Boolean Value 148
Defining `void` Functions 149
`return` Statements in `void` Functions 151
Preconditions and Postconditions 151
`main` Is a Function 153
Recursive Functions 153

3.3 SCOPE RULES 155
Local Variables 155
Procedural Abstraction 157
Global Constants and Global Variables 158
Blocks 161
Nested Scopes 162
TIP: Use Function Calls in Branching and Loop Statements 162
Variables Declared in a `for` Loop 163

Chapter Summary 164
Answers to Self-Test Exercises 164
Programming Projects 168

Chapter 4 Parameters and Overloading 175

4.1 PARAMETERS 176
Call-by-Value Parameters 176
A First Look at Call-by-Reference Parameters 178
Call-by-Reference Mechanism in Detail 181
Constant Reference Parameters 183
EXAMPLE: The `swapValues` Function 183
TIP: Think of Actions, Not Code 184
Mixed Parameter Lists 185
TIP: What Kind of Parameter to Use 186
PITFALL: Inadvertent Local Variables 188
TIP: Choosing Formal Parameter Names 189
EXAMPLE: Buying Pizza 190

4.2 OVERLOADING AND DEFAULT ARGUMENTS 193
Introduction to Overloading 193
PITFALL: Automatic Type Conversion and Overloading 196
Rules for Resolving Overloading 197
EXAMPLE: Revised Pizza-Buying Program 199
Default Arguments 201

4.3 TESTING AND DEBUGGING FUNCTIONS 203
The `assert` Macro 203
Stubs and Drivers 204

Chapter Summary 207
Answers to Self-Test Exercises 207
Programming Projects 209

Chapter 5 Arrays 215

5.1 INTRODUCTION TO ARRAYS 216
Declaring and Referencing Arrays 216
TIP: Use `for` Loops with Arrays 219
PITFALL: Array Indexes Always Start with Zero 219
TIP: Use a Defined Constant for the Size of an Array 219
Arrays in Memory 220
PITFALL: Array Index out of Range 222
The Range-Based `for` Loop 222
Initializing Arrays 223

5.2 ARRAYS IN FUNCTIONS 225
Indexed Variables as Function Arguments 225
Entire Arrays as Function Arguments 226
The `const` Parameter Modifier 230
PITFALL: Inconsistent Use of `const` Parameters 231
Functions That Return an Array 232
EXAMPLE: Production Graph 232

5.3 PROGRAMMING WITH ARRAYS 237
Partially Filled Arrays 337
TIP: Do Not Skimp on Formal Parameters 238
EXAMPLE: Searching an Array 241
EXAMPLE: Sorting an Array 243
EXAMPLE: Bubble Sort 247

5.4 MULTIDIMENSIONAL ARRAYS 251
Multidimensional Array Basics 251
Multidimensional Array Parameters 252
EXAMPLE: Two-Dimensional Grading Program 252

Chapter Summary 258
Answers to Self-Test Exercises 259
Programming Projects 263

Chapter 6 Structures and Classes 273

6.1 STRUCTURES 274
Structure Types 276
PITFALL: Forgetting a Semicolon in a Structure Definition 280
Structures as Function Arguments 280
TIP: Use Hierarchical Structures 281
Initializing Structures 283

6.2 CLASSES 286
Defining Classes and Member Functions 286
Encapsulation 292
Public and Private Members 293
Accessor and Mutator Functions 296
TIP: Separate Interface and Implementation 298
TIP: A Test for Encapsulation 299
Structures versus Classes 300
TIP: Thinking Objects 302

Chapter Summary 302
Answers to Self-Test Exercises 303
Programming Projects 305

Chapter 7 Constructors and Other Tools 309

7.1 CONSTRUCTORS 310
Constructor Definitions 310
PITFALL: Constructors with No Arguments 315
Explicit Constructor Calls 316
TIP: Always Include a Default Constructor 317
EXAMPLE: `BankAccount` Class 319
Class Type Member Variables 326
Member Initializers and Constructor Delegation in C++11 329

7.2 MORE TOOLS 330
The `const` Parameter Modifier 330
PITFALL: Inconsistent Use of `const` 332
Inline Functions 336
Static Members 338
Nested and Local Class Definitions 341

7.3 VECTORS—A PREVIEW OF THE STANDARD TEMPLATE LIBRARY 342
Vector Basics 342
PITFALL: Using Square Brackets beyond the Vector Size 344
TIP: Vector Assignment Is Well Behaved 346
Efficiency Issues 346

Chapter Summary 348
Answers to Self-Test Exercises 348
Programming Projects 350

Chapter 8 Operator Overloading, Friends, and References 355

8.1 BASIC OPERATOR OVERLOADING 356
Overloading Basics 357
TIP: A Constructor Can Return an Object 362
Returning by `const` Value 363
Overloading Unary Operators 366
Overloading as Member Functions 366
TIP: A Class Has Access to All Its Objects 369
Overloading Function Application `( )` 369
PITFALL: Overloading `&&`, `||`, and the Comma Operator 370

8.2 FRIEND FUNCTIONS AND AUTOMATIC TYPE CONVERSION 370
Constructors for Automatic Type Conversion 370
PITFALL: Member Operators and Automatic Type Conversion 371
Friend Functions 372
Friend Classes 375
PITFALL: Compilers without Friends 376

8.3 REFERENCES AND MORE OVERLOADED OPERATORS 377
References 378
TIP: Returning Member Variables of a Class Type 379
Overloading >> and << 380
TIP: What Mode of Returned Value to Use 386
The Assignment Operator 389
Overloading the Increment and Decrement Operators 389
Overloading the Array Operator [ ] 392
Overloading Based on L-Value versus R-Value 394

Chapter Summary 394
Answers to Self-Test Exercises 395
Programming Projects 397

Chapter 9 Strings 401

9.1 AN ARRAY TYPE FOR STRINGS 402
C-String Values and C-String Variables 403
PITFALL: Using = and == with C-strings 406
Other Functions in <cstring> 408
EXAMPLE: Command-Line Arguments 410
C-String Input and Output 413

9.2 CHARACTER MANIPULATION TOOLS 415
Character I/O 415
The Member Functions `get` and `put` 416
EXAMPLE: Checking Input Using a Newline Function 418
PITFALL: Unexpected `'\n'` in Input 420
The `putback`, `peek`, and `ignore` Member Functions 421
Character-Manipulating Functions 423
PITFALL: `toupper` and `tolower` Return `int` Values 425

9.3 THE STANDARD CLASS `string` 427
Introduction to the Standard Class `string` 427
I/O with the Class `string` 430
TIP: More Versions of `getline` 433
PITFALL: Mixing `cin >> variable;` and `getline` 433
String Processing with the Class `string` 435
EXAMPLE: Palindrome Testing 438
Converting between `string` Objects and C-Strings 442
Converting between `string` Objects and Numbers 442

Chapter Summary 443
Answers to Self-Test Exercises 444
Programming Projects 447

Chapter 10 Pointers and Dynamic Arrays 453

10.1 POINTERS 454
Pointer Variables 455
Basic Memory Management 463
`nullptr` 465
PITFALL: Dangling Pointers 466
Dynamic Variables and Automatic Variables 466
TIP: Define Pointer Types 467
PITFALL: Pointers as Call-by-Value Parameters 469
Uses for Pointers 470

10.2 DYNAMIC ARRAYS 471
Array Variables and Pointer Variables 471
Creating and Using Dynamic Arrays 473
EXAMPLE: A Function That Returns an Array 476
Pointer Arithmetic 478
Multidimensional Dynamic Arrays 479

10.3 CLASSES, POINTERS, AND DYNAMIC ARRAYS 482
The `->` Operator 482
The `this` Pointer 483
Overloading the Assignment Operator 483
EXAMPLE: A Class for Partially Filled Arrays 490
Destructors 493
Copy Constructors 494

Chapter Summary 499
Answers to Self-Test Exercises 499
Programming Projects 501

Chapter 11 Separate Compilation and Namespaces 505

11.1 SEPARATE COMPILATION 506
Encapsulation Reviewed 507
Header Files and Implementation Files 507
EXAMPLE: `DigitalTime` Class 516
TIP: Reusable Components 517
Using `#ifndef` 517
TIP: Defining Other Libraries 519

11.2 NAMESPACES 521
Namespaces and `using` Directives 521
Creating a Namespace 523
`using` Declarations 526

Qualifying Names 527
TIP: Choosing a Name for a Namespace 529
EXAMPLE: A Class Definition in a Namespace 530
Unnamed Namespaces 531
PITFALL: Confusing the Global Namespace and the Unnamed Namespace 537
TIP: Unnamed Namespaces Replace the `static` Qualifier 538
TIP: Hiding Helping Functions 538
Nested Namespaces 539
TIP: What Namespace Specification Should You Use? 539

Chapter Summary 542
Answers to Self-Test Exercises 542
Programming Projects 544

Chapter 12 Streams and File I/O 549

12.1 I/O STREAMS 551
File I/O 551
PITFALL: Restrictions on Stream Variables 556
Appending to a File 556
TIP: Another Syntax for Opening a File 558
TIP: Check That a File Was Opened Successfully 560
Character I/O 562
Checking for the End of a File 563

12.2 TOOLS FOR STREAM I/O 567
File Names as Input 567
Formatting Output with Stream Functions 568
Manipulators 572
Saving Flag Settings 573
More Output Stream Member Functions 574
EXAMPLE: Cleaning Up a File Format 576
EXAMPLE: Editing a Text File 578

12.3 STREAM HIERARCHIES: A PREVIEW OF INHERITANCE 581
Inheritance among Stream Classes 581
EXAMPLE: Another `newLine` Function 583
Parsing Strings with the `stringstream` Class 587

12.4 RANDOM ACCESS TO FILES 590

Chapter Summary 592
Answers to Self-Test Exercises 592
Programming Projects 595

Chapter 13 Recursion 605

13.1 RECURSIVE void FUNCTIONS 607
EXAMPLE: Vertical Numbers 607
Tracing a Recursive Call 610
A Closer Look at Recursion 613
PITFALL: Infinite Recursion 614
Stacks for Recursion 616
PITFALL: Stack Overflow 617
Recursion versus Iteration 618

13.2 RECURSIVE FUNCTIONS THAT RETURN A VALUE 619
General Form for a Recursive Function That Returns a Value 619
EXAMPLE: Another Powers Function 620
Mutual Recursion 625

13.3 THINKING RECURSIVELY 627
Recursive Design Techniques 627
Binary Search 628
Coding 630
Checking the Recursion 634
Efficiency 634

Chapter Summary 636
Answers to Self-Test Exercises 637
Programming Projects 641

Chapter 14 Inheritance 647

14.1 INHERITANCE BASICS 648
Derived Classes 648
Constructors in Derived Classes 658
PITFALL: Use of Private Member Variables from the Base Class 660
PITFALL: Private Member Functions Are Effectively Not Inherited 662
The protected Qualifier 662
Redefinition of Member Functions 665
Redefining versus Overloading 666
Access to a Redefined Base Function 668
Functions That Are Not Inherited 669

14.2 PROGRAMMING WITH INHERITANCE 670
Assignment Operators and Copy Constructors in Derived Classes 670
Destructors in Derived Classes 671
EXAMPLE: Partially Filled Array with Backup 672
PITFALL: Same Object on Both Sides of the Assignment Operator 681

EXAMPLE: Alternate Implementation of `PFArrayDBak` 681
TIP: A Class Has Access to Private Members of All Objects of the Class 684
TIP: "Is a" versus "Has a" 684
Protected and Private Inheritance 685
Multiple Inheritance 686

Chapter Summary 687
Answers to Self-Test Exercises 687
Programming Projects 689

Chapter 15 Polymorphism and Virtual Functions 697

15.1 VIRTUAL FUNCTION BASICS 698
Late Binding 698
Virtual Functions in C++ 699
Provide Context with C++11's `override` Keyword 705
Preventing a Virtual Function from Being Overridden 706
TIP: The Virtual Property Is Inherited 706
TIP: When to Use a Virtual Function 707
PITFALL: Omitting the Definition of a Virtual Member Function 707
Abstract Classes and Pure Virtual Functions 708
EXAMPLE: An Abstract Class 709

15.2 POINTERS AND VIRTUAL FUNCTIONS 711
Virtual Functions and Extended Type Compatibility 711
PITFALL: The Slicing Problem 715
TIP: Make Destructors Virtual 716
Downcasting and Upcasting 717
How C++ Implements Virtual Functions 718

Chapter Summary 720
Answers to Self-Test Exercises 721
Programming Projects 721

Chapter 16 Templates 729

16.1 FUNCTION TEMPLATES 730
Syntax for Function Templates 731
PITFALL: Compiler Complications 734
TIP: How to Define Templates 736
EXAMPLE: A Generic Sorting Function 737
PITFALL: Using a Template with an Inappropriate Type 741

16.2 CLASS TEMPLATES 743
Syntax for Class Templates 744
EXAMPLE: An Array Template Class 748
The `vector` and `basic_string` Templates 754

16.3 TEMPLATES AND INHERITANCE 754
EXAMPLE: Template Class For a Partially Filled Array with Backup 755

Chapter Summary 760
Answers to Self-Test Exercises 760
Programming Projects 764

Chapter 17 Linked Data Structures 767

17.1 NODES AND LINKED LISTS 769
Nodes 769
Linked Lists 774
Inserting a Node at the Head of a List 776
PITFALL: Losing Nodes 779
Inserting and Removing Nodes Inside a List 779
PITFALL: Using the Assignment Operator with Dynamic Data Structures 783
Searching a Linked List 783
Doubly Linked Lists 786
Adding a Node to a Doubly Linked List 788
Deleting a Node from a Doubly Linked List 788
EXAMPLE: A Generic Sorting Template Version of Linked List Tools 795

17.2 LINKED LIST APPLICATIONS 799
EXAMPLE: A Stack Template Class 799
EXAMPLE: A Queue Template Class 806
TIP: A Comment on Namespaces 809
Friend Classes and Similar Alternatives 810
EXAMPLE: Hash Tables With Chaining 813
Efficiency of Hash Tables 819
EXAMPLE: A Set Template Class 820
Efficiency of Sets Using Linked Lists 826

17.3 ITERATORS 827
Pointers as Iterators 828
Iterator Classes 828
EXAMPLE: An Iterator Class 830

17.4 TREES 836
Tree Properties 837
EXAMPLE: A Tree Template Class 839

Chapter Summary 844
Answers to Self-Test Exercises 845
Programming Projects 854

Chapter 18 Exception Handling 861

18.1 EXCEPTION HANDLING BASICS 863
A Toy Example of Exception Handling 863
Defining Your Own Exception Classes 872
Multiple Throws and Catches 872
PITFALL: Catch the More Specific Exception First 876
TIP: Exception Classes Can Be Trivial 877
Throwing an Exception in a Function 877
EXAMPLE: Returning the High Score 879
Exception Specification 882
PITFALL: Exception Specification in Derived Classes 884

18.2 PROGRAMMING TECHNIQUES FOR EXCEPTION HANDLING 885
When to Throw an Exception 886
PITFALL: Uncaught Exceptions 887
PITFALL: Nested `try-catch` Blocks 888
PITFALL: Overuse of Exceptions 888
Exception Class Hierarchies 889
Testing for Available Memory 889
Rethrowing an Exception 890

Chapter Summary 890
Answers to Self-Test Exercises 890
Programming Projects 892

Chapter 19 Standard Template Library 895

19.1 ITERATORS 897
Iterator Basics 897
PITFALL: Compiler Problems 902
TIP: Use `auto` to Simplify Variable Declarations 903
Kinds of Iterators 903
Constant and Mutable Iterators 906
Reverse Iterators 908
Other Kinds of Iterators 909

19.2 CONTAINERS 910
Sequential Containers 910
PITFALL: Iterators and Removing Elements 915
TIP: Type Definitions in Containers 916
The Container Adapters `stack` and `queue` 916
PITFALL: Underlying Containers 917
The Associative Containers `set` and `map` 920
Efficiency 925
TIP: Use Initialization, Ranged for, and `auto` with Containers 927

19.3 GENERIC ALGORITHMS 928
Running Times and Big-*O* Notation 928
Container Access Running Times 932
Nonmodifying Sequence Algorithms 933
Modifying Sequence Algorithms 937
Set Algorithms 939
Sorting Algorithms 940

Chapter Summary 941
Answers to Self-Test Exercises 941
Programming Projects 943

Chapter 20 Patterns and UML (online at www.pearsonglobaleditions.com/savitch)

Appendix 1 C++ Keywords AP-1
Appendix 2 Precedence of Operators AP-3
Appendix 3 The ASCII Character Set AP-5
Appendix 4 Some Library Functions AP-7
Appendix 5 Old and New Header Files AP-15
Appendix 6 Additional C++11 Language Features AP-17
Index I-1

C++ Basics 1

1.1 INTRODUCTION TO C++ 30
Origins of the C++ Language 30
C++ and Object-Oriented Programming 31
The Character of C++ 31
C++ Terminology 32
A Sample C++ Program 32
Tip: Compiling C++ Programs 33

1.2 VARIABLES, EXPRESSIONS, AND ASSIGNMENT STATEMENTS 34
Identifiers 35
Variables 36
Assignment Statements 39
Introduction to the `string` class 40
Pitfall: Uninitialized Variables 41
Tip: Use Meaningful Names 42
More Assignment Statements 42
Assignment Compatibility 43
Literals 44
Escape Sequences 46
Raw String Literals 47
Naming Constants 47
Arithmetic Operators and Expressions 48
Integer and Floating-Point Division 50
Pitfall: Division with Whole Numbers 51
Type Casting 52
Increment and Decrement Operators 54
Pitfall: Order of Evaluation 56

1.3 CONSOLE INPUT/OUTPUT 57
Output Using `cout` 57
New Lines in Output 58
Tip: End Each Program with `\n` or `endl` 59
Formatting for Numbers with a Decimal Point 59
Output with `cerr` 61
Input Using `cin` 61
Tip: Line Breaks in I/O 64

1.4 PROGRAM STYLE 65
Comments 65

1.5 LIBRARIES AND NAMESPACES 66
Libraries and `include` Directives 66
Namespaces 66
Pitfall: Problems with Library Names 67

Chapter Summary 68
Answers to Self-Test Exercises 69
Programming Projects 71

1 C++ Basics

The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths. Its province is to assist us in making available what we are already acquainted with.

ADA AUGUSTA, *Sketch of the Analytical Engine Invented by Charles Babbage, Esq. with notes by trans. Ada Lovelace, in Scientific Memoirs, Vol 3. 1842*

Introduction

This chapter introduces the C++ language and gives enough detail to allow you to handle simple programs involving expressions, assignments, and console input/output (I/O). The details of assignments and expressions are similar to those of most other high-level languages. Every language has its own console I/O syntax, so if you are not familiar with C++, that may look new and different to you.

1.1 Introduction to C++

Language is the only instrument of science.

SAMUEL JOHNSON, *A Journey to the Western Islands of Scotland. London: London A. Strahan and T. Cadell, 1791*

This section gives an overview of the C++ programming language.

Origins of the C++ Language

The C++ programming language can be thought of as the C programming language with classes (and other modern features) added. The C programming language was developed by Dennis Ritchie of AT&T Bell Laboratories in the 1970s. It was first used for writing and maintaining the UNIX operating system. (Up until that time, UNIX systems programs were written either in assembly language or in a language called B, a language developed by Ken Thompson, the originator of UNIX.) C is a general-purpose language that can be used for writing any sort of program, but its success and popularity are closely tied to the UNIX operating system. If you wanted to maintain your UNIX system, you needed to use C. C and UNIX fit together so well that soon not just systems programs but almost all commercial programs that ran under UNIX were written in the C language. C became so popular that versions of the language were written for other popular operating systems; its use is thus not limited to computers that use UNIX. However, despite its popularity, C was not without its shortcomings.

The C language is peculiar because it is a high-level language with many of the features of a low-level language. C is somewhere in between the two extremes of a very high-level language and a low-level language, and therein lies both its strengths and its weaknesses. Like (low-level) assembly language, C language programs can directly manipulate the computer's memory. On the other hand, C has the features of a high-level language, which makes it easier to read and write than assembly language. This makes C an excellent choice for writing systems programs, but for other programs (and in some sense even for systems programs) C is not as easy to understand as other languages; also, it does not have as many automatic checks as some other high-level languages.

To overcome these and other shortcomings of C, Bjarne Stroustrup of AT&T Bell Laboratories developed C++ in the early 1980s. Stroustrup designed C++ to be a better C. Most of C is a subset of C++ and so most C programs are also C++ programs. (The reverse is not true; many C++ programs are definitely not C programs.) Unlike C, C++ has facilities for classes and so can be used for object-oriented programming.

C++14 is the most recent version of the standard of the C++ programming language. It was approved on August 19, 2014, by the International Organization for Standardization. At the time of this writing, the C++14 standard has not yet been published. C++14 consists primarily of small improvements over C++11, while C++11 included significant improvements over the prior version. Newer compilers that can handle C++11 or C++14 are able to compile and run programs written for older versions of C++. However, the C++11 and C++14 standards include new language features that are not compatible with older C++ compilers. This means that if you have an older C++ compiler, then you may not be able to compile and run C++11 or C++14 programs.

C++ and Object-Oriented Programming

Object-oriented programming (OOP) is a currently popular and powerful programming technique. The main characteristics of OOP are encapsulation, inheritance, and polymorphism. Encapsulation is a form of information hiding or abstraction. Inheritance has to do with writing reusable code. Polymorphism refers to a way that a single name can have multiple meanings in the context of inheritance. Having made those statements, we must admit that they will hold little meaning for readers who have not heard of OOP before. However, we will describe all these terms in detail later in this book. C++ accommodates OOP by providing classes, a kind of data type combining both data and algorithms. C++ is not what some authorities would call a "pure OOP language." C++ tempers its OOP features with concerns for efficiency and what some might call "practicality." This combination has made C++ currently the most widely used OOP language, although not all of its usage strictly follows the OOP philosophy.

The Character of C++

C++ has classes that allow it to be used as an object-oriented language. It allows for overloading of functions and operators. (All these terms will be explained eventually, so do not be concerned if you do not fully understand some terms.) C++'s connection to the C language gives it a more traditional look than newer object-oriented languages, yet it has

more powerful abstraction mechanisms than many other currently popular languages. C++ has a template facility that allows for full and direct implementation of algorithm abstraction. C++ templates allow you to code using parameters for types. The newest C++ standard, and most C++ compilers, allow multiple namespaces to accommodate more reuse of class and function names. The exception handling facilities in C++ are similar to what you would find in other programming languages. Memory management in C++ is similar to that in C. The programmer must allocate his or her own memory and handle his or her own garbage collection. Most compilers will allow you to do C-style memory management in C++ since C is essentially a subset of C++. However, C++ also has its own syntax for a C++ style of memory management, and you are advised to use the C++ style of memory management when coding in C++. This book uses only the C++ style of memory management.

C++ Terminology

functions

program

All procedure-like entities are called **functions** in C++. Things that are called *procedures*, *methods*, *functions*, or *subprograms* in other languages are all called *functions* in C++. As we will see in the next subsection, a C++ **program** is basically just a function called `main`; when you run a program, the run-time system automatically invokes the function named `main`. Other C++ terminology is pretty much the same as most other programming languages, and in any case, will be explained when each concept is introduced.

A Sample C++ Program

VideoNote
Compiling and Running a C++ Program

Display 1.1 contains a simple C++ program and two possible screen displays that might be generated when a user runs the program. A C++ program is really a function definition for a function named `main`. When the program is run, the function named `main` is invoked. The body of the function `main` is enclosed in braces, `{}`. When the program is run, the statements in the braces are executed.

The following two lines set up things so that the libraries with console input and output facilities are available to the program. The details concerning these two lines and related topics are covered in Section 1.3 and in Chapters 9, 11, and 12.

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

`int main()`

The following line says that `main` is a function with no parameters that returns an `int` (integer) value:

```
int main( )
```

Some compilers will allow you to omit the `int` or replace it with `void`, which indicates a function that does not return a value. However, the previous form is the most universally accepted way to start the `main` function of a C++ program.

`return 0;`

The program ends when the following statement is executed:

```
return 0;
```

This statement ends the invocation of the function `main` and returns 0 as the function's value. According to the ANSI/ISO C++ standard, this statement is not required, but many compilers still require it. Chapter 3 covers all these details about C++ functions.

Display 1.1 A Sample C++ Program

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

3  int main( )
4  {
5      int numberOfLanguages;

6      cout << "Hello reader.\n"
7           << "Welcome to C++.\n";

8      cout << "How many programming languages have you used? ";
9      cin >> numberOfLanguages;

10     if (numberOfLanguages < 1)
11         cout << "Read the preface. You may prefer\n"
12              << "a more elementary book by the same author.\n";
13     else
14         cout << "Enjoy the book.\n";

15     return 0;
16  }
```

Sample Dialogue 1

```
Hello reader.
Welcome to C++.
How many programming languages have you used? 0
Read the preface. You may prefer
a more elementary book by the same author.
```

User types in 0 on the keyboard. User input is shown in bold.

Sample Dialogue 2

```
Hello reader.
Welcome to C++.
How many programming languages have you used? 1
Enjoy the book
```

User types in 1 on the keyboard. User input is shown in bold.

TIP: Compiling C++ Programs

You may also need to specify which C++ standard to compile against. For example, to compile for C++11 using g++ 4.7 requires the compiler flag of `-std=c++11` to be added to the command line; otherwise, the compiler will assume that the C++ program is written to an older standard. The command line to compile a C++11 program named testing.cpp would look like

```
g++ testing.cpp -std=c++11
```

(continued)

TIP: (continued)

Check the documentation with your compiler to determine whether any special steps are needed to compile C++11 (or C++14) programs and to determine what language features are supported. ■

Variable declarations in C++ are similar to what they are in other programming languages. The following line from Display 1.1 declares the variable `numberOfLanguages`:

```
int numberOfLanguages;
```

The type `int` is one of the C++ types for whole numbers (integers).

If you have not programmed in C++ before, then the use of `cin` and `cout` for console I/O is likely to be new to you. That topic is covered a little later in this chapter, but the general idea can be observed in this sample program. For example, consider the following two lines from Display 1.1:

```
cout << "How many programming languages have you used? ";
cin >> numberOfLanguages;
```

string

C-string

The first line outputs the text within the quotation marks to the screen. The text inside the quotation marks is called a **string**, or to be more precise, a **C-string**. The second line reads in a number that the user enters at the keyboard and sets the value of the variable `numberOfLanguages` to this number.

The lines

```
cout << "Read the preface. You may prefer\n"
     << "a more elementary book by the same author.\n";
```

output two strings instead of just one string. The details are explained in Section 1.3 later in this chapter, but this brief introduction will be enough to allow you to understand the simple use of `cin` and `cout` in the examples that precede Section 1.3. The symbolism `\n` is the newline character, which instructs the computer to start a new line of output.

Although you may not yet be certain of the exact details of how to write such statements, you can probably guess the meaning of the `if-else` statement. The details will be explained in the next chapter.

(By the way, if you have not had at least some experience with some programming languages, you should read the preface to see if you might prefer a more elementary book. You need not have had any experience with C++ to read this book, but some minimal programming experience is strongly suggested.)

1.2 Variables, Expressions, and Assignment Statements

Once a person has understood the way variables are used in programming, he has understood the quintessence of programming.

E. W. DIJKSTRA, *Notes on Structured Programming. August 1969*

Variables, expressions, and assignments in C++ are similar to those in most other general-purpose languages.

Identifiers

identifier

The name of a variable (or other item you might define in a program) is called an **identifier**. A C++ identifier must start with either a letter or the underscore symbol, and all the rest of the characters must be letters, digits, or the underscore symbol. For example, the following are all valid identifiers:

```
x x1 x_1 _abc ABC123z7 sum RATE count data2 bigBonus
```

All the names shown are legal and would be accepted by the compiler, but the first five are poor choices for identifiers because they are not descriptive of the identifier's use. None of the following are legal identifiers, and all would be rejected by the compiler:

```
12  3X  %change  data-1  myfirst.c  PROG.CPP
```

The first three are not allowed because they do not start with a letter or an underscore. The remaining three are not identifiers because they contain symbols other than letters, digits, and the underscore symbol.

Although it is legal to start an identifier with an underscore, you should avoid doing so, because identifiers starting with an underscore are informally reserved for system identifiers and standard libraries.

case-sensitive

C++ is a **case-sensitive** language; that is, it distinguishes between uppercase and lowercase letters in the spelling of identifiers. Hence, the following are three distinct identifiers and could be used to name three distinct variables:

```
rate  RATE  Rate
```

However, it is not a good idea to use two such variants in the same program, since that might be confusing. Although it is not required by C++, variables are usually spelled with their first letter in lowercase. The predefined identifiers, such as `main`, `cin`, `cout`, and so forth, must be spelled in all lowercase letters. The convention that is now becoming universal in object-oriented programming is to spell variable names with a mix of upper- and lowercase letters (and digits), to always start a variable name with a lowercase letter, and to indicate "word" boundaries with an uppercase letter, as illustrated by the following variable names:

```
topSpeed, bankRate1, bankRate2, timeOfArrival
```

This convention is not as common in C++ as in some other object-oriented languages, but is becoming more widely used and is a good convention to follow.

A C++ identifier can be of any length, although some compilers will ignore all characters after some (large) specified number of initial characters.

Identifiers

A C++ identifier must start with either a letter or the underscore symbol, and the remaining characters must all be letters, digits, or the underscore symbol. C++ identifiers are case sensitive and have no limit to their length.

keyword or reserved word

There is a special class of identifiers, called **keywords** or **reserved words**, which have a predefined meaning in C++ and cannot be used as names for variables or anything else. In the code displays of this book keywords are shown in a different color. A complete list of keywords is given in Appendix 1.

Some predefined words, such as `cin` and `cout`, are not keywords. These predefined words are not part of the core C++ language, and you are allowed to redefine them. Although these predefined words are not keywords, they are defined in libraries required by the C++ language standard. Needless to say, using a predefined identifier for anything other than its standard meaning can be confusing and dangerous and thus should be avoided. The safest and easiest practice is to treat all predefined identifiers as if they were keywords.

Variables

declare

Every variable in a C++ program must be *declared* before it is used. When you **declare** a variable you are telling the compiler—and, ultimately, the computer—what kind of data you will be storing in the variable. For example, the following are two definitions that might occur in a C++ program:

```
int numberOfBeans;
double oneWeight, totalWeight;
```

The first defines the variable `numberOfBeans` so that it can hold a value of type `int`, that is, a whole number. The name `int` is an abbreviation for "integer." The type `int` is one of the types for whole numbers. The second definition declares `oneWeight` and `totalWeight` to be variables of type `double`, which is one of the types for numbers with a decimal point (known as **floating-point numbers**). As illustrated here, when there is more than one variable in a definition, the variables are separated by commas. Also, note that each definition ends with a semicolon.

floating-point number

Every variable must be declared before it is used; otherwise, variables may be declared anyplace. Of course, they should always be declared in a location that makes the program easier to read. Typically, variables are declared either just before they are used or at the start of a block (indicated by an opening brace, `{`). Any legal identifier, other than a reserved word, may be used for a variable name.[1]

C++ has basic types for characters, integers, and floating-point numbers (numbers with a decimal point). Display 1.2 lists the basic C++ types. The commonly used type for integers is `int`. The type `char` is the type for single characters. The type `char` can be treated as an integer type, but we do not encourage you to do so. The commonly used type for floating-point numbers is `double`, and so you should use `double` for floating-point

[1] C++ makes a distinction between *declaring* and *defining* an identifier. When an identifier is declared, the name is introduced. When it is defined, storage for the named item is allocated. For the kind of variables we discuss in this chapter, and for much more of the book, what we are calling a *variable declaration* both declares the variable and defines the variable, that is, allocates storage for the variable. Many authors blur the distinction between variable definition and variable declaration. The difference between declaring and defining an identifier is more important for other kinds of identifiers, which we will encounter in later chapters.

numbers unless you have a specific reason to use one of the other floating-point types. The type `bool` (short for *Boolean*) has the values `true` and `false`. It is not an integer type, but to accommodate older code, you can convert back and forth between `bool` and any of the integer types. The programmer can also define types for arrays, classes, and pointers, all of which are discussed in later chapters of this book.

Display 1.2 **Simple Types**

TYPE NAME	MEMORY USED	SIZE RANGE	PRECISION
`short` (also called `short int`)	2 bytes	−32,768 to 32,767	Not applicable
`int`	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`long` (also called `long int`)	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`float`	4 bytes	approximately 10^{-38} to 10^{38}	7 digits
`double`	8 bytes	approximately 10^{-308} to 10^{308}	15 digits
`long double`	10 bytes	approximately 10^{-4932} to 10^{4932}	19 digits
`char`	1 byte	All ASCII characters (Can also be used as an integer type, although we do not recommend doing so.)	Not applicable
`bool`	1 byte	`true`, `false`	Not applicable

The values listed here are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system. *Precision* refers to the number of meaningful digits, including digits in front of the decimal point. The ranges for the types `float`, `double`, and `long double` are the ranges for positive numbers. Negative numbers have a similar range, but with a negative sign in front of each number.

Variable Declarations

All variables must be declared before they are used. The syntax for variable declarations is as follows.

SYNTAX

Type_Name Variable_Name_1, Variable_Name_2, . . .;

EXAMPLES

```
int count, numberOfDragons, numberOfTrolls;
double distance;
```

unsigned

Each of the integer types has an **unsigned** version that includes only nonnegative values. These types are `unsigned short`, `unsigned int`, and `unsigned long`. Their ranges do not exactly correspond to the ranges of the positive values of the types `short`, `int`, and `long`, but are likely to be larger (since they use the same storage as their corresponding types `short`, `int`, or `long`, but need not remember a sign). You are unlikely to need these types, but may run into them in specifications for predefined functions in some of the C++ libraries, which we discuss in Chapter 3.

The size of integer data types can vary from one machine to another. For example, on a 32-bit machine an integer might be 4 bytes, while on a 64-bit machine an integer might be 8 bytes. Sometimes this is problematic if you need to know exactly what range of values can be stored in an integer type. To address this problem, new integer types have been added to C++11 that specify exactly the size and whether or not the data type is signed or unsigned. These types are accessible by including `<cstdint>` at the top of the program. Display 1.3 illustrates some of these number types. In this text we will primarily use the more ambiguous types of `int` and `long`, but consider the C++11 types if you want to specify an exact size.

Display 1.3 **Some C++11 Fixed-Width Integer Types**

VideoNote
C++11 Fixed Width Integer Types

TYPE NAME	MEMORY USED	SIZE RANGE
int8_t	1 byte	–128 to 127
uint8_t	1 byte	0 to 255
int16_t	2 bytes	–32,768 to 32,767
uint16_t	2 bytes	0 to 65,535
int32_t	4 bytes	–2,147,483,648 to 2,147,483,647
uint32_t	4 bytes	0 to 4,294,967,295
int64_t	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint64_t	8 bytes	0 to 18,446,744,073,709,551,615
long long	At least 8 bytes	

C++11 also included a type named **auto** that deduces the type of a variable based on an expression on the right side of the equal sign. For example,

```
auto x = expression;
```

defines a variable named x whose data type matches whatever is computed from "expression". This feature doesn't buy us much at this point, but it will save us some long, messy code when we start to work with longer data types that we define ourselves.

In the other direction, C++11 introduced a way to determine the type of a variable or expression. **decltype** (`expr`) is the declared type of variable or expression `expr` and can be used in declarations:

```
int x = 10;
decltype(x*3.5) y;
```

This code declares `y` to be the same type as `x*3.5`. The expression `x*3.5` is a `double`, so `y` is declared as a `double`.

Assignment Statements

assignment statement

The most direct way to change the value of a variable is to use an **assignment statement**. In C++ the equal sign is used as the assignment operator. An assignment statement always consists of a variable on the left-hand side of the equal sign and an expression on the right-hand side. An assignment statement ends with a semicolon. The expression on the right-hand side of the equal sign may be a variable, a number, or a more complicated expression made up of variables, numbers, operators, and function invocations. An assignment statement instructs the computer to evaluate (that is, to compute the value of) the expression on the right-hand side of the equal sign and to set the value of the variable on the left-hand side equal to the value of that expression. The following are examples of C++ assignment statements:

```
totalWeight = oneWeight * numberOfBeans;
temperature = 98.6;
count = count + 2;
```

Assignment Statements

In an assignment statement, first the expression on the right-hand side of the equal sign is evaluated and then the variable on the left-hand side of the equal sign is set equal to this value.

SYNTAX

Variable = Expression;

EXAMPLES

```
distance = rate * time;
count = count + 2;
```

The first assignment statement sets the value of `totalWeight` equal to the number in the variable `oneWeight` multiplied by the number in `numberOfBeans`. (Multiplication is expressed using the asterisk, `*`, in C++.) The second assignment statement sets the value of `temperature` to `98.6`. The third assignment statement increases the value of the variable `count` by `2`.

In C++, assignment statements can be used as expressions. When used as an expression, an assignment statement returns the value assigned to the variable. For example, consider

```
n = (m = 2);
```

The subexpression `(m = 2)` changes the value of `m` to `2` and returns the value `2`. Thus, this sets both `n` and `m` equal to `2`. As you will see when we discuss precedence of operators in detail in Chapter 2, you can omit the parentheses, so the assignment statement under discussion can be written as

```
n = m = 2;
```

We advise you not to use an assignment statement as an expression, but you should be aware of this behavior because it will help you understand certain kinds of coding errors. For one thing, it will explain why you will not get an error message when you mistakenly write

```
n = m = 2;
```

when you meant to write

```
n = m + 2;
```

(This is an easy mistake to make since `=` and `+` are on the same keyboard key.)

Lvalues and Rvalues

Authors often refer to ***lvalue*** and ***rvalue*** in C++ books. An **lvalue** is anything that can appear on the left-hand side of an assignment operator (=), which means any kind of variable. An **rvalue** is anything that can appear on the right-hand side of an assignment operator, which means any expression that evaluates to a value.

Introduction to the `string` class

string

Although C++ lacks a simple data type to directly manipulate strings (sequences of text), there is a `string` class that may be used to process strings in a manner similar to the data types we have seen thus far. The distinction between a class and a simple data type such as an `int` is discussed in Chapter 6. Further details about the `string` class are discussed in Chapter 9.

To use the string class we must first include the string library by adding the following line of code at the top of your program:

```
#include <string>
```

You declare variables of type string just as you declare variables of types int or double. For example, the following declares one variable of type string and stores the text "durian" in it:

```
string fruit;
fruit = "durian";
```

PITFALL: Uninitialized Variables

A variable has no meaningful value until a program gives it one. For example, if the variable minimumNumber has not been given a value either as the left-hand side of an assignment statement or by some other means (such as being given an input value with a cin statement), then the following is an error:

```
desiredNumber = minimumNumber + 10;
```

This is because minimumNumber has no meaningful value, and so the entire expression on the right-hand side of the equal sign has no meaningful value. A variable like minimumNumber that has not been given a value is said to be **uninitialized**. This situation is, in fact, worse than it would be if minimumNumber had no value at all. An uninitialized variable, like minimumNumber, will simply have some garbage value. The value of an uninitialized variable is determined by whatever pattern of zeros and ones was left in its memory location by the last program that used that portion of memory.

uninitialized variable

One way to avoid an uninitialized variable is to initialize variables at the same time they are declared. This can be done by adding an equal sign and a value, as follows:

```
int minimumNumber = 3;
```

This both declares minimumNumber to be a variable of type int and sets the value of the variable minimumNumber equal to 3. You can use a more complicated expression involving operations such as addition or multiplication when you initialize a variable inside the declaration in this way. As another example, the following declares three variables and initializes two of them:

```
double rate = 0.07, time, balance = 0.00;
```

C++ allows an alternative notation for initializing variables when they are declared. This alternative notation is illustrated by the following, which is equivalent to the preceding declaration:

```
double rate(0.07), time, balance(0.00);
```

Initializing Variables in Declarations

You can initialize a variable (that is, give it a value) at the time that you declare the variable.

SYNTAX

```
Type_Name Variable_Name_1 = Expression_for_Value_1,
          Variable_Name_2 = Expression_for_Value_2,...;
```

EXAMPLES

```
int count = 0, limit = 10, fudgeFactor = 2;
double distance = 999.99;
```

SYNTAX

Alternative syntax for initializing in declarations:

```
Type_Name Variable_Name_1 (Expression_for_Value_1),
          Variable_Name_2 (Expression_for_Value_2),...;
```

EXAMPLES

```
int count(0), limit(10), fudgeFactor(2);
double distance(999.99);
```

TIP: Use Meaningful Names

Variable names and other names in a program should at least hint at the meaning or use of the thing they are naming. It is much easier to understand a program if the variables have meaningful names. Contrast

```
x = y * z;
```

with the more suggestive

```
distance = speed * time;
```

The two statements accomplish the same thing, but the second is easier to understand. ■

More Assignment Statements

A shorthand notation exists that combines the assignment operator (=) and an arithmetic operator so that a given variable can have its value changed by adding, subtracting, multiplying by, or dividing by a specified value. The general form is

```
Variable Operator = Expression
```

which is equivalent to

```
Variable = Variable Operator (Expression)
```

The *`Expression`* can be another variable, a constant, or a more complicated arithmetic expression. The following list gives examples.

EXAMPLE	EQUIVALENT TO
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time / rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`

Self-Test Exercises

1. Give the declaration for two variables called `feet` and `inches`. Both variables are of type `int` and both are to be initialized to zero in the declaration. Give both initialization alternatives.
2. Give the declaration for two variables called `count` and `distance`. `count` is of type `int` and is initialized to zero. `distance` is of type `double` and is initialized to `1.5`. Give both initialization alternatives.
3. Write a program that contains statements that output the values of five or six variables that have been defined, but not initialized. Compile and run the program. What is the output? Explain.

Assignment Compatibility

As a general rule, you cannot store a value of one type in a variable of another type. For example, most compilers will object to the following:

```
int intVariable;
intVariable = 2.99;
```

The problem is a type mismatch. The constant `2.99` is of type `double`, and the variable `intVariable` is of type `int`. Unfortunately, not all compilers will react the same way to the previous assignment statement. Some will issue an error message, some will give only a warning message, and some compilers will not object at all. Even if the compiler does allow you to use the previous assignment, it will give `intVariable` the `int` value `2`, not the value `3`. Since you cannot count on your compiler accepting the previous assignment, you should not assign a `double` value to a variable of type `int`.

Even if the compiler will allow you to mix types in an assignment statement, in most cases you should not. Doing so makes your program less portable, and it can be confusing.

assigning int values to double variables

There are some special cases in which it is permitted to assign a value of one type to a variable of another type. It is acceptable to assign a value of an integer type, such as `int`, to a variable of a floating-point type, such as type `double`. For example, the following is both legal and acceptable style:

```
double doubleVariable;
doubleVariable = 2;
```

The style shown will set the value of the variable named `doubleVariable` equal to `2.0`.

Although it is usually a bad idea to do so, you can store an `int` value such as `65` in a variable of type `char` and you can store a letter such as `'Z'` in a variable of type `int`. For many purposes, the C language considers characters to be small integers, and perhaps unfortunately, C++ inherited this from C. The reason for allowing this is that variables of type `char` consume less memory than variables of type `int`; thus, doing arithmetic with variables of type `char` can save some memory. However, it is clearer to use the type `int` when you are dealing with integers and to use the type `char` when you are dealing with characters.

mixing types

The general rule is that you cannot place a value of one type in a variable of another type—though it may seem that there are more exceptions to the rule than there are cases that follow the rule. Even if the compiler does not enforce this rule strictly, it is a good rule to follow. Placing data of one type in a variable of another type can cause problems because the value must be changed to a value of the appropriate type and that value may not be what you would expect.

integers and Booleans

Values of type `bool` can be assigned to variables of an integer type (`short`, `int`, `long`), and integers can be assigned to variables of type `bool`. However, it is poor style to do this. For completeness and to help you read other people's code, here are the details: When assigned to a variable of type `bool`, any nonzero integer will be stored as the value `true`. Zero will be stored as the value `false`. When assigning a `bool` value to an integer variable, `true` will be stored as `1`, and `false` will be stored as `0`.

Literals

literal constant

A **literal** is a name for one specific value. Literals are often called **constants** in contrast to variables. Literals or constants do not change value; variables can change their values. Integer constants are written in the way you are used to writing numbers. Constants of type `int` (or any other integer type) must not contain a decimal point. Constants of type `double` may be written in either of two forms. The simple form for `double` constants is like the everyday way of writing decimal fractions. When written in this form a `double` constant must contain a decimal point. No number constant (either integer or floating-point) in C++ may contain a comma.

scientific notation or floating-point notation

A more complicated notation for constants of type `double` is called **scientific notation** or **floating-point notation** and is particularly handy for writing very large numbers and very small fractions. For instance, 3.67×10^{17}, which is the same as

```
367000000000000000.0
```

is best expressed in C++ by the constant `3.67e17`. The number 5.89×10^{-6}, which is the same as `0.00000589`, is best expressed in C++ by the constant `5.89e-6`. The `e`

stands for *exponent* and means "multiply by 10 to the power that follows." The e may be either uppercase or lowercase.

Think of the number after the e as telling you the direction and number of digits to move the decimal point. For example, to change 3.49e4 to a numeral without an e, you move the decimal point four places to the right to obtain 34900.0, which is another way of writing the same number. If the number after the e is negative, you move the decimal point the indicated number of spaces to the left, inserting extra zeros if need be. So, 3.49e-2 is the same as 0.0349.

The number before the e may contain a decimal point, although it is not required. However, the exponent after the e definitely must *not* contain a decimal point.

What Is Doubled?

Why is the type for numbers with a fractional part called double? Is there a type called "single" that is half as big? No, but something like that is true. Many programming languages traditionally used two types for numbers with a fractional part. One type used less storage and was very imprecise (that is, it did not allow very many significant digits). The second type used *double* the amount of storage and so was much more precise; it also allowed numbers that were larger (although programmers tend to care more about precision than about size). The kinds of numbers that used twice as much storage were called *double-precision* numbers; those that used less storage were called *single precision*. Following this tradition, the type that (more or less) corresponds to this double-precision type was named double in C++. The type that corresponds to single precision in C++ was called float. C++ also has a third type for numbers with a fractional part, which is called long double.

Constants of type char are expressed by placing the character in single quotes, as illustrated in what follows:

```
char symbol = 'Z';
```

Note that the left and right single quote symbols are the same symbol.

Constants for strings of characters are given in double quotes, as illustrated by the following line taken from Display 1.1:

```
cout << "How many programming languages have you used? ";
```

quotes

Be sure to notice that string constants are placed inside double quotes, while constants of type char are placed inside single quotes. The two kinds of quotes mean different things. In particular, 'A' and "A" mean different things. 'A' is a value of type char and can be stored in a variable of type char. "A" is a string of characters. The fact that the string happens to contain only one character does *not* make "A" a value of type char. Also notice that for both strings and characters, the left and right quotes are the same.

C-string

Strings in double quotes, like "Hello", are often called **C-strings**. A C-string is not the same as the string class introduced earlier although both are used to store sequences of text and we sometimes use the two interchangeably. The difference is explained in detail in Chapter 9. Experts recommend you use the string class when possible instead of a C-string for purposes of security and flexibility.

The type `bool` has two constants, `true` and `false`. These two constants may be assigned to a variable of type `bool` or used anyplace else an expression of type `bool` is allowed. They must be spelled with all lowercase letters.

Escape Sequences

escape sequence

A backslash, `\`, preceding a character tells the compiler that the sequence following the backslash does not have the same meaning as the character appearing by itself. Such a sequence is called an **escape sequence**. The sequence is typed in as two characters with no space between the symbols. Several escape sequences are defined in C++

If you want to put a backslash, `\`, or a quote symbol, `"`, into a string constant, you must escape the ability of the `"` to terminate a string constant by using `\"`, or the ability of the `\` to escape, by using `\\`. The `\\` tells the compiler you mean a real backslash, `\`, not an escape sequence; the `\"` tells it you mean a real quote, not the end of a string constant.

A stray `\`, say `\z`, in a string constant will have different effects on different compilers. One compiler may simply give back a `z`; another might produce an error. The ANSI/ISO standard states that unspecified escape sequences have undefined behavior. This means a compiler can do anything its author finds convenient. The consequence is that code that uses undefined escape sequences is not portable. You should not use any escape sequences other than those provided by the C++ standard. These C++ control characters are listed in Display 1.4.

Display 1.4 Some Escape Sequences

SEQUENCE	MEANING
`\n`	New line
`\r`	Carriage return (Positions the cursor at the start of the current line. You are not likely to use this very much.)
`\t`	(Horizontal) Tab (Advances the cursor to the next tab stop.)
\a	Alert (Sounds the alert noise, typically a bell.)
`\\`	Backslash (Allows you to place a backslash in a quoted expression.)
`\'`	Single quote (Mostly used to place a single quote inside single quotes.)
`\"`	Double quote (Mostly used to place a double quote inside a quoted string.)
The following are not as commonly used, but we include them for completeness:	
`\v`	Vertical tab
`\b`	Backspace
`\f`	Form feed
`\?`	Question mark

Raw String Literals

C++11 introduced a new feature, **raw string literals**, that can make it easier to insert escape sequences. Raw string literals are especially helpful if you want to encode a string with backslashes. For example, say that you literally want to encode the exact string "\t\\t\n". If we try to do it like this:

```
string s = "\t\\t\n";
```

then because the escape character is \, C++ interprets the first "\t" as a tab, the next "\\" as a backslash, the "t" as a char t, and the "\n" as a newline. The string stored in variable s is then "tab" + "\t" + newline. To store the literal string "\t\\t\n", we need to use "\\" for every occurrence of the backslash. The result is

```
string s = "\\t\\\\t\\n";
```

An easier way to encode this string is to use raw string literals. The format is an uppercase R followed by double quotes and then the desired literal string in parentheses. The following would store "\t\\t\n" in variable s:

```
string s = R"(\t\\t\n)";
```

The variable s now contains the exact string "\t\\t\n". Raw string literals are handy when you need to reference a pathname to a file or when you use regular expressions, which allow the programmer to specify a pattern for matching characters.

Naming Constants

Numbers in a computer program pose two problems. The first is that they carry no mnemonic value. For example, when the number 10 is encountered in a program, it gives no hint of its significance. If the program is a banking program, it might be the number of branch offices or the number of teller windows at the main office. To understand the program, you need to know the significance of each constant. The second problem is that when a program needs to have some numbers changed, the changing tends to introduce errors. Suppose that 10 occurs twelve times in a banking program—four of the times it represents the number of branch offices, and eight of the times it represents the number of teller windows at the main office. When the bank opens a new branch and the program needs to be updated, there is a good chance that some of the 10s that should be changed to 11 will not be, or some that should not be changed will be. The way to avoid these problems is to name each number and use the name instead of the number within your program. For example, a banking program might have two constants with the names BRANCH_COUNT and WINDOW_COUNT.

Both these numbers might have a value of 10, but when the bank opens a new branch, all you need do to update the program is change the definition of BRANCH_COUNT.

How do you name a number in a C++ program? One way to name a number is to initialize a variable to that number value, as in the following example:

```
int BRANCH_COUNT = 10;
int WINDOW_COUNT = 10;
```

There is, however, one problem with this method of naming number constants: You might inadvertently change the value of one of these variables. C++ provides a way of marking an initialized variable so that it cannot be changed. If your program tries to change one of these variables, it produces an error condition. To mark a variable declaration so that the value of the variable cannot be changed, precede the declaration with the word `const` (which is an abbreviation of *constant*). For example,

const

```
const int BRANCH_COUNT = 10;
const int WINDOW_COUNT = 10;
```

If the variables are of the same type, it is possible to combine the previous two lines into one declaration, as follows:

```
const int BRANCH_COUNT = 10, WINDOW_COUNT = 10;
```

However, most programmers find that placing each name definition on a separate line is clearer. The word `const` is often called a **modifier**, because it modifies (restricts) the variables being declared.

modifier

A variable declared using the `const` modifier is often called a **declared constant**. Writing declared constants in all uppercase letters is not required by the C++ language, but it is standard practice among C++ programmers.

declared constant

Once a number has been named in this way, the name can then be used anywhere the number is allowed, and it will have exactly the same meaning as the number it names. To change a named constant, you need only change the initializing value in the `const` variable declaration. The meaning of all occurrences of `BRANCH_COUNT`, for instance, can be changed from `10` to `11` simply by changing the initializing value of `10` in the declaration of `BRANCH_COUNT`.

Display 1.5 contains a simple program that illustrates the use of the declaration modifier `const`.

Arithmetic Operators and Expressions

As in most other languages, C++ allows you to form expressions using variables, constants, and the arithmetic operators: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (modulo, remainder). These expressions can be used anyplace it is legal to use a value of the type produced by the expression.

mixing types

All the arithmetic operators can be used with numbers of type `int`, numbers of type `double`, and even with one number of each type. However, the type of the value produced and the exact value of the result depend on the types of the numbers being combined. If both operands (that is, both numbers) are of type `int`, then the result of combining them with an arithmetic operator is of type `int`. If one or both of the operands are of type `double`, then the result is of type `double`. For example, if the

variables `baseAmount` and `increase` are of type `int`, then the number produced by the following expression is of type `int`:

```
baseAmount + increase
```

However, if one or both of the two variables are of type `double`, then the result is of type `double`. This is also true if you replace the operator + with any of the operators −, *, or /.

More generally, you can combine any of the arithmetic types in expressions. If all the types are integer types, the result will be the integer type. If at least one of the subexpressions is of a floating-point type, the result will be a floating-point type. C++ tries its best to make the type of an expression either `int` or `double`, but if the value produced by the expression is not of one of these types because of the value's size, a suitable different integer or floating-point type will be produced.

precedence rules

You can specify the order of operations in an arithmetic expression by inserting parentheses. If you omit parentheses, the computer will follow rules called **precedence rules** that determine the order in which the operations, such as addition and multiplication, are performed. These precedence rules are similar to rules used in algebra and other mathematics classes. For example,

```
x + y * z
```

is evaluated by first doing the multiplication and then the addition. Except in some standard cases, such as a string of additions or a simple multiplication embedded inside

Display 1.5 Named Constant

```
1  #include <iostream>
2  using namespace std;
3
4  int main( )
5  {
6      const double RATE = 6.9;
7      double deposit;

8      cout << "Enter the amount of your deposit $";
9      cin >> deposit;

10     double newBalance;
11     newBalance = deposit + deposit*(RATE/100);
12     cout << "In one year, that deposit will grow to\n"
13          << "$" << newBalance << " an amount worth waiting for.\n";

14     return 0;
15 }
```

Sample Dialogue

```
Enter the amount of your deposit $100
In one year, that deposit will grow to
$106.9 an amount worth waiting for.
```

Naming Constants with the `const` Modifier

When you initialize a variable inside a declaration, you can mark the variable so that the program is not allowed to change its value. To do this, place the word `const` in front of the declaration, as described here:

SYNTAX

```
const  Type_Name Variable_Name = Constant;
```

EXAMPLES

```
const int MAX_TRIES = 3;
const double PI = 3.14159;
```

an addition, it is usually best to include the parentheses, even if the intended order of operations is the one dictated by the precedence rules. The parentheses make the expression easier to read and less prone to programmer error. A complete set of C++ precedence rules is given in Appendix 2.

Integer and Floating-Point Division

integer division

When used with one or both operands of type `double`, the division operator, `/`, behaves as you might expect. However, when used with two operands of type `int`, the division operator yields the integer part resulting from division. In other words, integer division discards the part after the decimal point. So, `10/3` is `3` (not `3.3333`…), `5/2` is `2` (not `2.5`), and `11/3` is `3` (not `3.6666`…). Notice that the number *is not rounded*; the part after the decimal point is discarded no matter how large it is.

the % operator

The operator `%` can be used with operands of type `int` to recover the information lost when you use `/` to do division with numbers of type `int`. When used with values of type `int`, the two operators `/` and `%` yield the two numbers produced when you perform the long division algorithm you learned in grade school. For example, `17` divided by `5` yields `3` with a remainder of `2`. The `/` operation yields the number of times one number "goes into" another. The `%` operation gives the remainder. For example, the statements

```
cout << "17 divided by 5 is " << (17 / 5) << "\n";
cout << "with a remainder of " << (17 % 5) << "\n";
```

yield the following output:

```
17 divided by 5 is 3
with a remainder of 2
```

negative integers in division

When used with negative values of type `int`, the result of the operators `/` and `%` can be different for different implementations of C++. Thus, you should use `/` and `%` with `int` values only when you know that both values are nonnegative.

PITFALL: Division with Whole Numbers

When you use the division operator / on two integers, the result is an integer. This can be a problem if you expect a fraction. Moreover, the problem can easily go unnoticed, resulting in a program that looks fine but is producing incorrect output without you even being aware of the problem. For example, suppose you are a landscape architect who charges $5,000 per mile to landscape a highway, and suppose you know the length of the highway you are working on in feet. The price you charge can easily be calculated by the following C++ statement:

```
totalPrice = 5000 * (feet/5280.0);
```

This works because there are 5,280 feet in a mile. If the stretch of highway you are landscaping is 15,000 feet long, this formula will tell you that the total price is

```
5000 * (15000/5280.0)
```

Your C++ program obtains the final value as follows: `15000/5280.0` is computed as `2.84`. Then the program multiplies `5000` by `2.84` to produce the value `14200.00`. With the aid of your C++ program, you know that you should charge $14,200 for the project.

Now suppose the variable `feet` is of type `int`, and you forget to put in the decimal point and the zero, so that the assignment statement in your program reads

```
totalPrice = 5000 * (feet/5280);
```

It still looks fine, but will cause serious problems. If you use this second form of the assignment statement, you are dividing two values of type `int`, so the result of the division `feet/5280` is `15000/5280`, which is the `int` value `2` (instead of the value `2.84` that you think you are getting). The value assigned to `totalPrice` is thus `5000*2`, or `10000.00`. If you forget the decimal point, you will charge $10,000. However, as we have already seen, the correct value is $14,200. A missing decimal point has cost you $4,200. Note that this will be true whether the type of `totalPrice` is `int` or `double`; the damage is done before the value is assigned to `totalPrice`. ■

Self-Test Exercises

4. Convert each of the following mathematical formulas to a C++ expression.

$$3x \quad 3x+y \quad \frac{x+y}{7} \quad \frac{3x+y}{z+2}$$

5. What is the output of the following program lines when they are embedded in a correct program that declares all variables to be of type `char`?

```
a = 'b';
b = 'c';
c = a;
cout << a << b << c << 'c';
```

(continued)

Self-Test Exercises (continued)

6. What is the output of the following program lines when they are embedded in a correct program that declares `number` to be of type `int`?

```
number = (1/3) * 3;
cout << "(1/3) * 3 is equal to " << number;
```

7. Write a complete C++ program that reads two whole numbers into two variables of type `int` and then outputs both the whole number part and the remainder when the first number is divided by the second. This can be done using the operators `/` and `%`.

8. Given the following fragment that purports to convert from degrees Celsius to degrees Fahrenheit, answer the following questions:

```
double c = 20;
double f;
f = (9/5) * c + 32.0;
```

   a. What value is assigned to `f`?
   b. Explain what is actually happening, and what the programmer likely wanted.
   c. Rewrite the code as the programmer intended.

Type Casting

type cast

A **type cast** is a way of changing a value of one type to a value of another type. A type cast is a kind of function that takes a value of one type and produces a value of another type that is C++'s best guess of an equivalent value. C++ has four to six different kinds of casts, depending on how you count them. There is an older form of type cast that has two notations for expressing it, and there are four new kinds of type casts introduced with the latest standard. The new kinds of type casts were designed as replacements for the older form; in this book, we will use the newer kinds. However, C++ retains the older kind(s) of cast along with the newer kinds, so we will briefly describe the older kind as well.

Let's start with the newer kinds of type casts. Consider the expression `9/2`. In C++ this expression evaluates to `4` because when both operands are of an integer type, C++ performs integer division. In some situations, you might want the answer to be the `double` value `4.5`. You can get a result of `4.5` by using the "equivalent" floating-point value `2.0` in place of the integer value `2`, as in `9/2.0`, which evaluates to `4.5`. But what if the `9` and the `2` are the values of variables of type `int` named `n` and `m`? Then, `n/m` yields `4`. If you want floating-point division in this case, you must do a type cast from `int` to `double` (or another floating-point type), such as in the following:

```
double ans = n/static_cast<double>(m);
```

The expression

```
static_cast<double>(m)
```

is a type cast. The expression `static_cast<double>` is like a function that takes an `int` argument (actually, an argument of almost any type) and returns an "equivalent" value of type `double`. So, if the value of `m` is `2`, the expression `static_cast<double>(m)` returns the `double` value `2.0`.

Note that `static_cast<double>(n)` does not change the value of the variable `n`. If `n` has the value `2` before this expression is evaluated, then `n` still has the value `2` after the expression is evaluated. (If you know what a function is in mathematics or in some programming language, you can think of `static_cast<double>` as a function that returns an "equivalent" value of type `double`.)

You may use any type name in place of `double` to obtain a type cast to another type. We said this produces an "equivalent" value of the target type. The word equivalent is in quotes because there is no clear notion of equivalent that applies to any two types. In the case of a type cast from an integer type to a floating-point type, the effect is to add a decimal point and a zero. The type cast in the other direction, from a floating-point type to an integer type, simply deletes the decimal point and all digits after the decimal point. Note that when type casting from a floating-point type to an integer type, the number is truncated, not rounded. `static_cast<int>(2.9)` is `2`; it is not `3`.

This `static_cast` is the most common kind of type cast and the only one we will use for some time. For completeness and reference value, we list all four kinds of type casts. Some may not make sense until you reach the relevant topics. If some or all of the remaining three kinds do not make sense to you at this point, do not worry. The four kinds of type cast are as follows:

```
static_cast<Type>(Expression)
const_cast<Type>(Expression)
dynamic_cast<Type>(Expression)
reinterpret_cast<Type>(Expression)
```

We have already discussed `static_cast`. It is a general-purpose type cast that applies in most "ordinary" situations. The `const_cast` is used to cast away constantness. The `dynamic_cast` is used for safe downcasting from one type to a descendent type in an inheritance hierarchy. The `reinterpret_cast` is an implementation-dependent cast that we will not discuss in this book and that you are unlikely to need. (These descriptions may not make sense until you cover the appropriate topics, where they will be discussed further. For now, we only use `static_cast`.)

The older form of type casting is approximately equivalent to the `static_cast` kind of type casting but uses a different notation. One of the two notations uses a type name as if it were a function name. For example, `int(9.3)` returns the `int` value `9`; `double(42)` returns the value `42.0`. The second, equivalent, notation for the older form of type casting would write `(double)42` instead of `double(42)`. Either notation can be used with variables or other more complicated expressions instead of just with constants.

Although C++ retains this older form of type casting, you are encouraged to use the newer form of type casting. (Someday, the older form may go away, although there is, as yet, no such plan for its elimination.)

As we noted earlier, you can always assign a value of an integer type to a variable of a floating-point type, as in

```
double d = 5;
```

In such cases C++ performs an automatic type cast, converting the `5` to `5.0` and placing `5.0` in the variable `d`. You cannot store the `5` as the value of `d` without a type cast, but sometimes C++ does the type cast for you. Such an automatic conversion is sometimes called a **type coercion**.

type coercion

Increment and Decrement Operators

The `++` in the name of the C++ language comes from the increment operator, `++`. The **increment operator** adds 1 to the value of a variable. The **decrement operator,** `--`, subtracts 1 from the value of a variable. They are usually used with variables of type `int`, but they can be used with any numeric type. If `n` is a variable of a numeric type, then `n++` increases the value of `n` by 1 and `n--` decreases the value of `n` by 1. So `n++` and `n--` (when followed by a semicolon) are executable statements. For example, the statements

increment operator

decrement operator

```
int n = 1, m = 7;
n++;
cout << "The value of n is changed to " << n << "\n";
m--;
cout << "The value of m is changed to " << m << "\n";
```

yield the following output:

```
The value of n is changed to 2
The value of m is changed to 6
```

An expression like `n++` returns a value as well as changing the value of the variable `n`, so `n++` can be used in an arithmetic expression such as

```
2*(n++)
```

The expression `n++` first returns the value of the variable `n`, and *then* the value of `n` is increased by 1. For example, consider the following code:

```
int n = 2;
int valueProduced = 2*(n++);
cout << valueProduced << "\n";
cout << n << "\n";
```

This code will produce the output

```
4
3
```

Notice the expression `2*(n++)`. When C++ evaluates this expression, it uses the value that `number` has *before* it is incremented, not the value that it has after it is incremented. Thus, the value produced by the expression `n++` is `2`, even though the increment operator changes the value of `n` to `3`. This may seem strange, but sometimes it is just

what you want. And, as you are about to see, if you want an expression that behaves differently, you can have it.

v++ versus ++v

The expression `n++` evaluates to the value of the variable `n`, and *then* the value of the variable `n` is incremented by 1. If you reverse the order and place the `++` in front of the variable, the order of these two actions is reversed. The expression `++n` first increments the value of the variable `n` and then returns this increased value of `n`. For example, consider the following code:

```
int n = 2;
int valueProduced = 2*(++n);
cout << valueProduced << "\n";
cout << n << "\n";
```

This code is the same as the previous piece of code except that the `++` is before the variable, so this code will produce the following output:

```
6
3
```

Notice that the two increment operators in `n++` and `++n` have the same effect on a variable `n`: They both increase the value of `n` by 1. But the two expressions evaluate to different values. Remember, if the `++` is *before* the variable, the incrementing is done *before* the value is returned; if the `++` is *after* the variable, the incrementing is done *after* the value is returned.

Everything we said about the increment operator applies to the decrement operator as well, except that the value of the variable is decreased by 1 rather than increased by 1. For example, consider the following code:

```
int n = 8;
int valueProduced = n--;
cout << valueProduced << "\n";
cout << n << "\n";
```

This produces the output

```
8
7
```

On the other hand, the code

```
int n = 8;
int valueProduced = --n;
cout << valueProduced << "\n";
cout << n << "\n";
```

produces the output

```
7
7
```

`n--` returns the value of `n` and then decrements `n`; on the other hand, `--n` first decrements `n` and then returns the value of `n`.

You cannot apply the increment and decrement operators to anything other than a single variable. Expressions such as `(x + y)++`, `--(x + y)`, `5++`, and so forth, are all illegal in C++.

The increment and decrement operators can be dangerous when used inside more complicated expressions, as explained in the following Pitfall.

PITFALL: Order of Evaluation

For most operators, the order of evaluation of subexpressions is not guaranteed. In particular, you normally cannot assume that the order of evaluation is left to right. For example, consider the following expression:

```
n + (++n)
```

Suppose `n` has the value `2` before the expression is evaluated. Then, if the first expression is evaluated first, the result is `2 + 3`. If the second expression is evaluated first, the result is `3 + 3`. Since C++ does not guarantee the order of evaluation, the expression could evaluate to either `5` or `6`. The moral is that you should not program in a way that depends on order of evaluation, except for the operators discussed in the next paragraph.

Some operators do guarantee that their order of evaluation of subexpressions is left to right. For the operators `&&` (and), `||` (or), and the comma operator (which is discussed in Chapter 2), C++ guarantees that the order of evaluations is left to right. Fortunately, these are the operators for which you are most likely to want a predictable order of evaluation. For example, consider

```
(n <= 2) && (++n > 2)
```

Suppose `n` has the value `2`, before the expression is evaluated. In this case you know that the subexpression `(n <= 2)` is evaluated before the value of `n` is incremented.

You thus know that `(n <= 2)` will evaluate to `true` and so the entire expression will evaluate to `true`.

Do not confuse order of operations (by precedence rules) with order of evaluation. For example,

```
(n + 2) * (++n) + 5
```

always means

```
((n + 2) * (++n)) + 5
```

However, it is not clear whether the `++n` is evaluated before or after the `n + 2`. Either one could be evaluated first.

Now you know why we said that it is usually a bad idea to use the increment (`++`) and decrement (`--`) operators as subexpressions of larger expressions.

If this is too confusing, just follow the simple rule of not writing code that depends on the order of evaluation of subexpressions. ■

1.3 Console Input/Output

Garbage in means garbage out.

Programmers' saying/Unattributed

Simple console input is done with the objects cin, cout, and cerr, all of which are defined in the library iostream. In order to use this library, your program should contain the following near the start of the file containing your code:

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

Output Using cout

cout

The values of variables as well as strings of text may be output to the screen using cout. Any combination of variables and strings can be output. For example, consider the following from the program in Display 1.1:

```
cout << "Hello reader.\n"
     << "Welcome to C++.\n";
```

This statement outputs two strings, one per line. Using cout, you can output any number of items, each either a string, a variable, or a more complicated expression. Simply insert a << before each thing to be output.

As another example, consider the following:

```
cout << numberOfGames << "games played.";
```

This statement tells the computer to output two items: the value of the variable numberOfGames and the quoted string "games played.".

Notice that you do not need a separate copy of the object cout for each item output. You can simply list all the items to be output, preceding each item to be output with the arrow symbols <<. The previous single cout statement is equivalent to the following two cout statements:

```
cout << numberOfGames;
cout << " games played.";
```

expression in a cout statement

You can include arithmetic expressions in a cout statement, as shown by the following example, where price and tax are variables:

```
cout << "The total cost is $" << (price + tax);
```

Parentheses around arithmetic expressions, such as price + tax, are required by some compilers, so it is best to include them.

The two < symbols should be typed without any space between them. The arrow notation << is often called the **insertion operator**. The entire cout statement ends with a semicolon.

spaces in output

Notice the spaces inside the quotes in our examples. The computer does not insert any extra space before or after the items output by a `cout` statement, which is why the quoted strings in the examples often start or end with a blank. The blanks keep the various strings and numbers from running together. If all you need is a space and there is no quoted string where you want to insert the space, then use a string that contains only a space, as in the following:

```
cout << firstNumber << " " << secondNumber;
```

Similarly, if you place the '+' symbol between two variables of type `string` then this operator concatenates (i.e. joins) the two strings together to create one longer string. For example, the code

```
string day1 = "Monday", day2="Tuesday";
cout << day1 + day2;
```

outputs the concatenated string of

```
"MondayTuesday"
```

New Lines in Output

As noted in the subsection on escape sequences, `\n` tells the computer to start a new line of output. Unless you tell the computer to go to the next line, it will put all the output on the same line. Depending on how your screen is set up, this can produce anything from arbitrary line breaks to output that runs off the screen. Notice that the `\n` goes inside the quotes. In C++, going to the next line is considered to be a special character, and the way you spell this special character inside a quoted string is `\n`, with no space between the two symbols in `\n`. Although it is typed as two symbols, C++ considers `\n` to be a single character that is called the **newline character**.

newline character

If you wish to insert a blank line in the output, you can output the newline character `\n` by itself:

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

Another way to output a blank line is to use `endl`, which means essentially the same thing as `"\n"`. So you can also output a blank line as follows:

```
cout << endl;
```

Although `"\n"` and `endl` mean the same thing, they are used slightly differently; `\n` must always be inside quotes, and `endl` should not be placed in quotes.

deciding between `\n` and `endl`

A good rule for deciding whether to use `\n` or `endl` is the following: If you can include the `\n` at the end of a longer string, then use `\n`, as in the following:

```
cout << "Fuel efficiency is "
     << mpg << " miles per gallon\n";
```

On the other hand, if the `\n` would appear by itself as the short string `"\n"`, then use `endl` instead:

```
cout << "You entered " << number << endl;
```

Starting New Lines in Output

To start a new output line, you can include \n in a quoted string, as in the following example:

```
cout << "You have definitely won\n"
     << "one of the following prizes:\n";
```

Recall that \n is typed as two symbols with no space in between the two symbols. Alternatively, you can start a new line by outputting endl. An equivalent way to write the previous cout statement is as follows:

```
cout << "You have definitely won" << endl
     << "one of the following prizes:" << endl;
```

TIP: End Each Program with \n or endl

It is a good idea to output a newline instruction at the end of every program. If the last item to be output is a string, then include a \n at the end of the string; if not, output an endl as the last output action in your program. This serves two purposes. Some compilers will not output the last line of your program unless you include a newline instruction at the end. On other systems, your program may work fine without this final newline instruction, but the next program that is run will have its first line of output mixed with the last line of the previous program. Even if neither of these problems occurs on your system, putting a newline instruction at the end will make your programs more portable. ■

Formatting for Numbers with a Decimal Point

format for double values

When the computer outputs a value of type double, the format may not be what you would like. For example, the following simple cout statement can produce any of a wide range of outputs:

```
cout << "The price is $" << price << endl;
```

If price has the value 78.5, the output might be

```
The price is $78.500000
```

or it might be

```
The price is $78.5
```

or it might be output in the following notation (which was explained in the subsection entitled “Literals”):

```
The price is $7.850000e01
```

It is extremely unlikely that the output will be the following, however, even though this is the format that makes the most sense:

```
The price is $78.50
```

To ensure that the output is in the form you want, your program should contain some sort of instructions that tell the computer how to output the numbers.

magic formula

There is a "magic formula" that you can insert in your program to cause numbers that contain a decimal point, such as numbers of type `double`, to be output in everyday notation with the exact number of digits after the decimal point that you specify. If you want two digits after the decimal point, use the following magic formula:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

outputting money amounts

If you insert the preceding three statements in your program, then any `cout` statements that follow these statements will output values of any floating-point type in ordinary notation, with exactly two digits after the decimal point. For example, suppose the following `cout` statement appears somewhere after this magic formula and suppose the value of `price` is `78.5`.

```
cout << "The price is $" << price << endl;
```

The output will then be as follows:

```
The price is $78.50
```

You may use any other nonnegative whole number in place of `2` to specify a different number of digits after the decimal point. You can even use a variable of type `int` in place of the `2`.

We will explain this magic formula in detail in Chapter 12. For now, you should think of this magic formula as one long instruction that tells the computer how you want it to output numbers that contain a decimal point.

If you wish to change the number of digits after the decimal point so that different values in your program are output with different numbers of digits, you can repeat the magic formula with some other number in place of `2`. However, when you repeat the magic formula, you only need to repeat the last line of the formula. If the magic formula has already occurred once in your program, then the following line will change the number of digits after the decimal point to five for all subsequent values of any floating-point type that are output:

```
cout.precision(5);
```

Outputting Values of Type `double`

If you insert the following "magic formula" in your program, then all numbers of type `double` (or any other type of floating-point number) will be output in ordinary notation with two digits after the decimal point:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

You can use any other nonnegative whole number in place of the `2` to specify a different number of digits after the decimal point. You can even use a variable of type `int` in place of the `2`.

Output with `cerr`

`cerr`

The object `cerr` is used in the same way as `cout`. The object `cerr` sends its output to the standard error output stream, which normally is the console screen. This gives you a way to distinguish two kinds of output: `cout` for regular output, and `cerr` for error message output. If you do nothing special to change things, then `cout` and `cerr` will both send their output to the console screen, so there is no difference between them.

On some systems you can redirect output from your program to a file. This is an operating system instruction, not a C++ instruction, but it can be useful. On systems that allow for output redirection, `cout` and `cerr` may be redirected to different files.

Input Using `cin`

`cin`

You use `cin` for input more or less the same way you use `cout` for output. The syntax is similar, except that `cin` is used in place of `cout` and the arrows point in the opposite direction. For example, in the program in Display 1.1, the variable `numberOfLanguages` was filled by the following `cin` statement:

```
cin >> numberOfLanguages;
```

You can list more than one variable in a single `cin` statement, as illustrated by the following:

```
cout << "Enter the number of dragons\n"
     << "followed by the number of trolls.\n";
cin >> dragons >> trolls;
```

If you prefer, the above `cin` statement can be written on two lines, as follows:

```
cin >> dragons
    >> trolls;
```

Notice that, as with the `cout` statement, there is just one semicolon for each occurrence of `cin`.

how cin works

When a program reaches a cin statement, it waits for input to be entered from the keyboard. It sets the first variable equal to the first value typed at the keyboard, the second variable equal to the second value typed, and so forth. However, the program does not read the input until the user presses the Return key. This allows the user to backspace and correct mistakes when entering a line of input. Display 1.6 illustrates reading an int and a string in the same program along with a simple calculation.

separate numbers with spaces

whitespace

Numbers in the input must be separated by one or more spaces or by a line break. These delimiting characters are called **whitespace**. When you use cin statements, the computer will skip over any number of blanks or line breaks until it finds the next input value. Thus, it does not matter whether input numbers are separated by one space or several spaces or even a line break. This same behavior holds when you are reading data into a string. This means that you cannot input a string that contains spaces. This may sometimes cause errors, as indicated in Display 1.6, Sample Dialogue 2. In this case, the user intends to enter "Mr. Bojangles" as the name of the pet, but the string is only read up to "Mr." since the next character is a space. The "Bojangles" string is ignored by this program but would be read next if there was another cin statement. Chapter 9 describes a technique to input a string that may include spaces.

You can read in integers, floating-point numbers, characters, or strings using cin. Later in this book we will discuss the reading in of other kinds of data using cin.

Display 1.6 Using cin and cout with a string (part 1 of 2)

```
1  //Program to demonstrate cin and cout with strings
2  #include <iostream>
3  #include <string>    <-- Needed to access the string class.

4  using namespace std;
5  int main( )
6  {
7    string dogName;
8     int actualAge;
9     int humanAge;

10    cout << "How many years old is your dog?" << endl;
11    cin >> actualAge;
12    humanAge = actualAge * 7;

13    cout << "What is your dog's name?" << endl;
14    cin >> dogName;

15    cout << dogName << "'s age is approximately " <<
16           "equivalent to a " << humanAge << " year old human."
17           << endl;

18     return 0;
19  }
```

Display 1.6 Using `cin` and `cout` with a string (part 2 of 2) (continued)

Sample Dialogue 1

```
How many years old is your dog?
5
What is your dog's name?
Rex
Rex's age is approximately equivalent to a 35 year old human.
```

Sample Dialogue 2

```
How many years old is your dog?
10
What is your dog's name?
Mr. Bojangles
Mr.'s age is approximately equivalent to a 70 year old human.
```

"Bojangles" is not read into `dogName` *because* `cin` *stops input at the space.*

`cin` Statements

A `cin` statement sets variables equal to values typed in at the keyboard.

SYNTAX

`cin >>` *Variable_1* >> *Variable_2* >> . . .;

EXAMPLES

```
cin >> number >> size;
cin >> timeLeft
    >> pointsNeeded;
```

Self-Test Exercises

9. Give an output statement that will produce the following message on the screen.

```
The answer to the question of
Life, the Universe, and Everything is 42.
```

10. Give an input statement that will fill the variable `theNumber` (of type `int`) with a number typed in at the keyboard. Precede the input statement with a prompt statement asking the user to enter a whole number.

(continued)

Self-Test Exercises (continued)

11. What statements should you include in your program to ensure that when a number of type `double` is output, it will be output in ordinary notation with three digits after the decimal point?

12. Write a complete C++ program that writes the phrase `Hello world` to the screen. The program does nothing else.

13. Give an output statement that produces the letter `'A'`, followed by the newline character, followed by the letter `'B'`, followed by the tab character, followed by the letter `'C'`.

14. The following code intends to input a user's first name, last name, and age. However, it has an error. Fix the code.

```
string fullName;
int age;
cout << "Enter your first and last name." << endl;
cin >> fullName;
cout << "Enter your age." << endl;
cin >> age;
cout << "You are " << age << " years old, " << fullName << endl;
```

15. What will the following code output?

```
string s1 = "5";
string s2 = "3";
string s3 = s1 + s2;
cout << s3 << endl;
```

TIP: Line Breaks in I/O

It is possible to keep output and input on the same line, and sometimes it can produce a nicer interface for the user. If you simply omit a `\n` or `endl` at the end of the last prompt line, then the user's input will appear on the same line as the prompt. For example, suppose you use the following prompt and input statements:

```
cout << "Enter the cost per person: $";
cin >> costPerPerson;
```

When the `cout` statement is executed, the following will appear on the screen:

```
Enter the cost per person: $
```

When the user types in the input, it will appear on the same line, like this:

```
Enter the cost per person: $1.25
```

1.4 Program Style

In matters of grave importance, style, not sincerity, is the vital thing.

OSCAR WILDE, *The Importance of Being Earnest. Act III. London, 1895*

C++ programming style is similar to that used in other languages. The goal is to make your code easy to read and easy to modify. We will say a bit about indenting in the next chapter. We have already discussed defined constants. Most, if not all, literals in a program should be defined constants. Choice of variable names and careful indenting should eliminate the need for very many comments, but any points that still remain unclear deserve a comment.

Comments

There are two ways to insert comments in a C++ program. In C++, two slashes, `//`, are used to indicate the start of a comment. All the text between the `//` and the end of the line is a comment. The compiler simply ignores anything that follows `//` on a line. If you want a comment that covers more than one line, place a `//` on each line of the comment. The symbols `//` do not have a space between them.

Another way to insert comments in a C++ program is to use the symbol pairs `/*` and `*/`. Text between these symbols is considered a comment and is ignored by the compiler. Unlike the `//` comments, which require an additional `//` on each line, the `/*`-to-`*/` comments can span several lines, like so:

```
/*This is a comment that spans
three lines. Note that there is no comment
symbol of any kind on the second line.*/
```

Comments of the `/* */` type may be inserted anywhere in a program that a space or line break is allowed. However, they should not be inserted anywhere except where they are easy to read and do not distract from the layout of the program. Usually, comments are placed at the ends of lines or on separate lines by themselves.

Opinions differ regarding which kind of comment is best to use. Either variety (the `//` kind or the `/* */` kind) can be effective if used with care. One approach is to use the `//` comments in final code and reserve the `/**/`-style comments for temporarily commenting out code while debugging.

when to comment

It is difficult to say just how many comments a program should contain. The only correct answer is "just enough," which of course conveys little to the novice programmer. It will take some experience to get a feel for when it is best to include a comment. Whenever something is important and not obvious, it merits a comment. However, too many comments are as bad as too few. A program that has a comment on each line will be so buried in comments that the structure of the program is hidden in a sea of obvious observations. Comments like the following contribute nothing to understanding and should not appear in a program:

```
distance = speed * time; //Computes the distance traveled.
```

1.5 Libraries and Namespaces

C++ comes with a number of standard libraries. These libraries place their definitions in a *namespace*, which is simply a name given to a collection of definitions. The techniques for including libraries and dealing with namespaces will be discussed in detail later in this book. This section discusses enough details to allow you to use the standard C++ libraries.

Libraries and `include` Directives

`#include`

C++ includes a number of standard libraries. In fact, it is almost impossible to write a C++ program without using at least one of these libraries. The normal way to make a library available to your program is with an `include` directive. An `include` directive for a standard library has the form

```
#include <Library_Name>
```

For example, the library for console I/O is `iostream`. So, most of our demonstration programs will begin

```
#include <iostream>
```

Compilers (preprocessors) can be very fussy about spacing in `include` directives. Thus, it is safest to type an `include` directive with no extra space: no space before the `#`, no space after the `#`, and no spaces inside the `<>`.

An `include` directive is simply an instruction to include the text found in a file at the location of the `include` directive. A library name is simply the name of a file that includes all the definitions of items in the library. We will eventually discuss using `include` directives for things other than standard libraries, but for now we only need `include` directives for standard C++ libraries. A list of some standard C++ libraries is given in Appendix 4.

preprocessor

C++ has a **preprocessor** that handles some simple textual manipulation before the text of your program is given to the compiler. Some people will tell you that `include` directives are not processed by the compiler but are processed by a preprocessor. They're right, but the difference is more of a word game than anything that need concern you. On almost all compilers, the preprocessor is called automatically when you compile your program.

Technically speaking, only part of the library definition is given in the header file. However, at this stage, that is not an important distinction, since using the `include` directive with the header file for a library will (on almost all systems) cause C++ to automatically add the rest of the library definition.

Namespaces

namespace

A **namespace** is a collection of name definitions. One name, such as a function name, can be given different definitions in two namespaces. A program can then use one of these namespaces in one place and the other in another location. We will discuss

namespaces in detail later in this book. For now, we only need to discuss the namespace `std`. All the standard libraries we will be using are defined in the `std` (standard) namespace. To use any of these definitions in your program, you must insert the following `using` directive:

using namespace

```
using namespace std;
```

Thus, a simple program that uses console I/O would begin

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

If you want to make some, but not all, names in a namespace available to your program, there is a form of the `using` directive that makes just one name available. For example, if you only want to make the name `cin` from the `std` namespace available to your program, you could use the following `using` directive:

```
using std::cin;
```

Thus, if the only names from the `std` namespace that your program uses are `cin`, `count`, and `endl`, you might start your program with

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
```

instead of

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

Older C++ header files for libraries did not place their definitions in the `std` namespace, so if you look at older C++ code, you will probably see that the header file names are spelled slightly differently and the code does not contain any `using` directive. This is allowed for backward compatibility. However, you should use the newer library header files and the `std` namespace directive.

PITFALL: Problems with Library Names

The C++ language is constantly in transition. If you are using a compiler that has not yet been revised to meet the new standard, then you will need to use different library names.

If the following does not work

```
#include <iostream>
```

use

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

(continued)

PITFALL: (continued)

Similarly, other library names are different for older compilers. Appendix 5 gives the correspondence between older and newer library names. This book always uses the new compiler names. If a library name does not work with your compiler, try the corresponding older library name. In all probability, either all the new library names will work or you will need to use all old library names. It is unlikely that only some of the library names have been made up to date on your system.

If you use the older library names (the ones that end in `.h`), you do *not* need the `using` directive

```
using namespace std;
```

Chapter Summary

- C++ is *case sensitive*. For example, `count` and `COUNT` are two different identifiers.
- Use meaningful names for variables.
- Variables must be declared before they are used. Other than following this rule, a variable declaration may appear anyplace.
- Be sure that variables are initialized before the program attempts to use their value. This can be done when the variable is declared or with an *assignment statement* before the variable is first used.
- You can assign a value of an integer type, like `int`, to a variable of a floating-point type, like `double`, but not vice versa.
- C++11 introduced new fixed-width integer data types.
- Almost all number constants in a program should be given meaningful names that can be used in place of the numbers. This can be done by using the modifier `const` in a variable declaration.
- Use enough parentheses in arithmetic expressions to make the order of operations clear.
- The object `cout` is used for console output.
- A `\n` in a quoted string or an `endl` sent to console output starts a new line of output.
- The object `cerr` is used for error messages. In a typical environment, `cerr` behaves the same as `cout`.
- The object `cin` is used for console input.

- In order to use `cin`, `cout`, or `cerr`, you should place the following directives near the beginning of the file with your program:

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

- There are two forms of comments in C++: Everything following `//` on the same line is a comment, and anything enclosed in `/*` and `*/` is a comment.
- Do not over comment.

Answers to Self-Test Exercises

1.
```
int feet = 0, inches = 0;
int feet(0), inches(0);
```

2.
```
int count = 0;
double distance = 1.5;
int count(0);
double distance(1.5);
   public static void main(String[] args)
```

3. The actual output from a program such as this is dependent on the system and the history of the use of the system.

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

int main( )
{
   int first, second, third, fourth, fifth;
   cout << first << " " << second << " " << third
        << " " << fourth << " " << fifth << "\n";
   return 0;
}
```

4.
```
3*x
3*x + y
(x + y)/7 Note that x + y/7 is not correct.
(3*x + y)/(z + 2)
```

5. `bcbc`

6. `(1/3) * 3 is equal to 0`

   Since `1` and `3` are of type `int`, the `/` operator performs integer division, which discards the remainder, so the value of `1/3` is `0`, not `0.3333...`. This makes the value of the entire expression `0 * 3`, which of course is `0`.

7.
```
#include <iostream>
using namespace std;
int main( )
{
    int number1, number2;
    cout << "Enter two whole numbers: ";
    cin >> number1 >> number2;
    cout << number1 << " divided by " << number2
         << " equals " << (number1/number2) << "\n"
         << "with a remainder of " << (number1%number2)
         << "\n";
    return 0;
}
```

8. a. `52.0`

   b. `9/5` has `int` value `1`. Since the numerator and denominator are both `int`, integer division is done; the fractional part is discarded. The programmer probably wanted floating-point division, which does not discard the part after the decimal point.

   c.
```
f = (9.0/5) * c + 32.0;
or
f = 1.8 * c + 32.0;
```

9.
```
cout << "The answer to the question of\n"
     << "Life, the Universe, and Everything is 42.\n";
```

10.
```
cout << "Enter a whole number and press Return: ";
cin  >> theNumber;
```

11.
```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(3);
```

12.
```
#include <iostream>
using namespace std;

int main( )
{
    cout << "Hello world\n";
    return 0;
}
```

13.
```
cout << 'A' << endl << 'B' << '\t' << 'C';
```
Other answers are also correct. For example, the letters could be in double quotes instead of single quotes. Another possible answer is the following:
```
cout << "A\nB\tC";
```

14. cin only reads up to the next whitespace, so the first and last name cannot be read into a single string as written if there is a space between the first name and last name. For now the easiest solution is to read the first and last name into two separate variables:

```
string first, last;
int age;
cout << "Enter your first and last name." << endl;
cin >> first >> last;
cout << "Enter your age." << endl;
cin >> age;
cout << "You are " << age << " years old, " << first <<
     " " << last << endl;
```

15. The + operator concatenates two string operands. The result is `s3 = "53"`. If s1 and s2 were numeric data types then the values would be added.

Programming Projects

1. One gallon is 3785.41 milliliters. Write a program that will read the volume of a package of milk in milliliters, and output the volume in gallons, as well as the number of packets needed to yield one gallon of milk.
2. A government research lab has concluded that an artificial sweetener commonly used in diet soda will cause death in laboratory mice. A friend of yours is desperate to lose weight but cannot give up soda. Your friend wants to know how much diet soda it is possible to drink without dying as a result. Write a program to supply the answer. The input to the program is the amount of artificial sweetener needed to kill a mouse, the weight of the mouse, and the weight of the dieter. To ensure the safety of your friend, be sure the program requests the weight at which the dieter will stop dieting, rather than the dieter's current weight. Assume that diet soda contains one-tenth of 1% artificial sweetener. Use a variable declaration with the modifier const to give a name to this fraction. You may want to express the percentage as the double value 0.001.
3. An electronics company sees a 13% increase in sales from the previous year, making 27% more profit than the previous work. Write a program that takes the company's previous year's sales and profit in millions as input and outputs the increase in sales and profit in millions from the previous year. Use a variable declaration with the modifier const to express the sales and profit increase.
4. Negotiating a consumer loan is not always straightforward. One form of loan is the discount installment loan, which works as follows. Suppose a loan has a face value of $1,000, the interest rate is 15%, and the duration is 18 months. The interest is computed by multiplying the face value of $1,000 by 0.15, yielding $150. That figure is then multiplied by the loan period of 1.5 years to yield $225 as the total interest owed. That amount is immediately deducted from the face value, leaving

the consumer with only $775. Repayment is made in equal monthly installments based on the face value. So the monthly loan payment will be $1,000 divided by 18, which is $55.56. This method of calculation may not be too bad if the consumer needs $775, but the calculation is a bit more complicated if the consumer needs $1,000. Write a program that will take three inputs: the amount the consumer needs to receive, the interest rate, and the duration of the loan in months. The program should then calculate the face value required in order for the consumer to receive the amount needed. It should also calculate the monthly payment.

5. Write a program that determines whether a truck is loaded beyond its rated capacity. The program will read in the truck's maximum load capacity (in metric tons), the number of boxes it's carrying, and the average weight of a box. If the total weight of the boxes is less than or equal to the truck's capacity, the program announces that the truck can legally carry all boxes and tells how many additional boxes can be added. If the number of boxes exceeds the maximum capacity, the program announces that the truck is overloaded and tells how many boxes must be removed.

6. A city cab service provider charges customers based on distance and time. The service charges $2 for the first two kilometers. After the first two kilometers, the service charges $0.50 per kilometer for the next six kilometers. After six kilometers, the customer is charged $1 per kilometer. At the end of the ride, a fee of $0.2 per minute is charged based on the total travel duration. Write a program that reads the distance traveled in kilometers and the time taken in minutes for a cab ride and computes the fare based on the service's charges.

7. A person runs on a treadmill for fifteen minutes a day as part of their exercise routine. They start the workout at a speed of 4 MPH for the first three minutes, and then increase it by 2 MPH after every four minutes. Write a program to calculate the total calories burned during this workout. The number of calories burned may be estimated using the formula

$$\text{Calories/Minute} = 0.0175 \times (\text{Number of metabolic equivalents (MET)}) \times (\text{Weight in kilograms})$$

Running at 6 MPH expends 10 METS. Your program should input the person's weight in kilograms and output the calories burned by them.

8. The Babylonian algorithm to compute the square root of a positive number **n** is as follows:

    1. Make a **guess** at the answer (you can pick n/2 as your initial guess).
    2. Compute ***r*** = **n / guess**.
    3. Set **guess** = **(guess + r) / 2**.
    4. Go back to step 2 for as many iterations as necessary. The more steps 2 and 3 are repeated, the closer **guess** will become to the square root of **n**.

Write a program that inputs a double for **n**, iterates through the Babylonian algorithm five times, and outputs the answer as a `double` to two decimal places. Your answer will be most accurate for small values of **n**.

9. The video game machines at your local arcade output coupons depending on how well you play the game. You can redeem 10 coupons for a candy bar or 3 coupons for a gumball. You prefer candy bars to gumballs. Write a program that inputs the number of coupons you win and outputs how many candy bars and gumballs you can get if you spend all of your coupons on candy bars first and any remaining coupons on gumballs.

10. Write a program that reads the speed of an airplane in miles per hour and the travel time in hours. The program should display the distance traveled by the airplane in that time in miles, and its speed in meters per second. Use the equation

$$\text{Speed in m/s} = (\text{Speed in MPH}/3600) \times 1609.344$$

VideoNote **Solution to Programming Project 1.11**

11. Write a program that inputs an integer that represents a length of time in seconds. The program should then output the number of hours, minutes, and seconds that corresponds to that number of seconds. For example, if the user inputs 50391 total seconds then the program should output 13 hours, 59 minutes, and 51 seconds.

12. A simple rule to estimate your ideal body weight is to allow 110 pounds for the first 5 feet of height and 5 pounds for each additional inch. Write a program with a variable for the height of a person in feet and another variable for the additional inches and input values for these variables from the keyboard. Assume the person is at least 5 feet tall. For example, a person that is 6 feet and 3 inches tall would be represented with a variable that stores the number 6 and another variable that stores the number 3. Based on these values calculate and output the ideal body weight.

13. Scientists estimate that consuming roughly 10 grams of caffeine at once is a lethal overdose. Write a program that inputs the number of milligrams of caffeine in a drink and outputs how many of those drinks it would take to kill a person. A 12-ounce can of cola has approximately 34 mg of caffeine, while a 16-ounce cup of coffee has approximately 160 mg of caffeine.

This page intentionally left blank

Flow of Control 2

2.1 BOOLEAN EXPRESSIONS 76
Building Boolean Expressions 76
Pitfall: Strings of Inequalities 77
Evaluating Boolean Expressions 78
Precedence Rules 80
Pitfall: Integer Values Can Be Used as Boolean Values 84

2.2 BRANCHING MECHANISMS 86
`if-else` Statements 86
Compound Statements 88
Pitfall: Using `=` in Place of `==` 89
Omitting the `else` 91
Nested Statements 91
Multiway `if-else` Statement 91
The `switch` Statement 92
Pitfall: Forgetting a `break` in a `switch` Statement 95
Tip: Use `switch` Statements for Menus 95
Enumeration Types 95
The Conditional Operator 97

2.3 LOOPS 97
The `while` and `do-while` Statements 98
Increment and Decrement Operators Revisited 101
The Comma Operator 102
The `for` Statement 104
Tip: Repeat-*N*-Times Loops 106
Pitfall: Extra Semicolon in a `for` Statement 107
Pitfall: Infinite Loops 107
The `break` and `continue` Statements 110
Nested Loops 113

2.4 INTRODUCTION TO FILE INPUT 113
Reading from a Text File Using `ifstream` 114

Chapter Summary 117
Answers to Self-Test Exercises 117
Programming Projects 123

2 Flow of Control

"Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to," said the Cat.

LEWIS CARROLL, *Alice's Adventures in Wonderland. London: Macmillan and Co., 1865*

Introduction

As in most programming languages, C++ handles flow of control with branching and looping statements. C++ branching and looping statements are similar to branching and looping statements in other languages. They are the same as in the C language and very similar to what they are in the Java programming language. Exception handling is also a way to handle flow of control. Exception handling is covered in Chapter 18.

2.1 Boolean Expressions

He who would distinguish the true from the false must have an adequate idea of what is true and false.

BENEDICT SPINOZA, *Ethics, Demonstrated in Geometrical Order. 1677*

Boolean expression

Most branching statements are controlled by Boolean expressions. A **Boolean expression** is any expression that is either true or false. The simplest form for a Boolean expression consists of two expressions, such as numbers or variables, which are compared with one of the comparison operators shown in Display 2.1. Notice that some of the operators are spelled with two symbols, for example, `= =`, `!=`, `<=`, or `>=`. Be sure to notice that you use a double equal `= =` for the equal sign and that you use the two symbols `!=` for not equal. Such two-symbol operators should not have any space between the two symbols.

Building Boolean Expressions

&& means "and"

You can combine two comparisons using the "and" operator, which is spelled `&&` in C++. For example, the following Boolean expression is true provided `x` is greater than `2` *and* `x` is less than `7`:

```
(2 < x) && (x < 7)
```

When two comparisons are connected using an `&&`, the entire expression is true, provided both of the comparisons are true; otherwise, the entire expression is false.

The “and” Operator, &&

You can form a more elaborate Boolean expression by combining two simpler Boolean expressions using the “and” operator, &&.

SYNTAX FOR A BOOLEAN EXPRESSION USING &&

```
(Boolean_Exp_1) && (Boolean_Exp_2)
```

SYNTAX (WITHIN AN if-else STATEMENT)

```
if ( (score > 0) && (score < 10))
    cout << "score is between 0 and 10.\n";
else
    cout << "score is not between 0 and 10.\n";
```

If the value of `score` is greater than `0` and the value of `score` is also less than `10`, then the first `cout` statement will be executed; otherwise, the second `cout` statement will be executed. (`if-else` statements are covered a bit later in this chapter, but the meaning of this simple example should be intuitively clear.)

|| means “or”

You can also combine two comparisons using the “or” operator, which is spelled `||` in C++. For example, the following is true provided `y` is less than `0` *or* `y` is greater than `12`:

```
(y < 0)||(y > 12)
```

When two comparisons are connected using a `||`, the entire expression is true provided that one or both of the comparisons are true; otherwise, the entire expression is false.

You can negate any Boolean expression using the `!` operator. If you want to negate a Boolean expression, place the expression in parentheses and place the `!` operator in front of it. For example, `!(x < y)` means “`x` is *not* less than `y`.” The `!` operator can usually be avoided. For example, `!(x < y)` is equivalent to `x >= y`. In some cases you can safely omit the parentheses, but the parentheses never do any harm. The exact details on omitting parentheses are given in the subsection entitled “**Precedence Rules**.”

PITFALL: Strings of Inequalities

Do not use a string of inequalities such as `x < z < y`. If you do, your program will probably compile and run, but it will undoubtedly give incorrect output. Instead, you must use two inequalities connected with an `&&`, as follows:

```
(x < z) && (z < y)
```

Display 2.1 Comparison Operators

MATH SYMBOL	ENGLISH	C++ NOTATION	C++ SAMPLE	MATH EQUIVALENT
=	Equal to	==	`x + 7 == 2*y`	$x + 7 = 2y$
≠	Not equal to	!=	`ans != 'n'`	$ans \neq$ 'n'
<	Less than	<	`count < m + 3`	$count < m + 3$
≤	Less than or equal to	<=	`time <= limit`	$time \leq limit$
>	Greater than	>	`time > limit`	$time > limit$
≥	Greater than or equal to	>=	`age >= 21`	$age \geq 21$

The "or" Operator, ||

You can form a more elaborate Boolean expression by combining two simpler Boolean expressions using the "or" operator, ||.

SYNTAX FOR A BOOLEAN EXPRESSION USING ||

```
(Boolean_Exp_1) || (Boolean_Exp_2)
```

EXAMPLE WITHIN AN `if-else` STATEMENT

```
if ( (x == 1) || (x == y))
    cout << "x is 1 or x equals y.\n";
else
    cout << "x is neither 1 nor equal to y.\n";
```

If the value of `x` is equal to `1` or the value of `x` is equal to the value of `y` (or both), then the first `cout` statement will be executed; otherwise, the second `cout` statement will be executed. (`if-else` statements are covered a bit later in this chapter, but the meaning of this simple example should be intuitively clear.)

Evaluating Boolean Expressions

As you will see in the next two sections of this chapter, Boolean expressions are used to control branching and looping statements. However, a Boolean expression has an independent identity apart from any branching or looping statement you might use it

in. A variable of type `bool` can store either of the values `true` or `false`. Thus, you can set a variable of type `bool` equal to a Boolean expression. For example,

```
bool result = (x < z) && (z < y);
```

A Boolean expression can be evaluated in the same way that an arithmetic expression is evaluated. The only difference is that an arithmetic expression uses operations such as `+`, `*`, and / and produces a number as the final result, whereas a Boolean expression uses relational operations such as `= =` and `<` and Boolean operations such as `&&`, `||` , and `!` and produces one of the two values `true` or `false` as the final result. Note that `=`, `!=`, `<`, `<=`, and so forth, operate on pairs of any built-in type to produce a Boolean value `true` or `false`.

First let's review evaluating an arithmetic expression. The same technique will work to evaluate Boolean expressions. Consider the following arithmetic expression:

```
(x + 1) * (x + 3)
```

Assume that the variable `x` has the value `2`. To evaluate this arithmetic expression, you evaluate the two sums to obtain the numbers `3` and `5`, and then you combine these two numbers `3` and `5` using the `*` operator to obtain `15` as the final value. Notice that in performing this evaluation, you do not multiply the expressions `(x + 1)` and `(x + 3)`. Instead, you multiply the values of these expressions. You use `3`; you do not use `(x + 1)`. You use `5`; you do not use `(x + 3)`.

truth tables

The computer evaluates Boolean expressions the same way. Subexpressions are evaluated to obtain values, each of which is either `true` or `false`. These individual values of `true` or `false` are then combined according to the rules in the tables shown in Display 2.2. For example, consider the Boolean expression

```
!( (y < 3) || (y > 7) )
```

which might be the controlling expression for an `if-else` statement. Suppose the value of `y` is `8`. In this case `(y < 3)` evaluates to `false` and `(y > 7)` evaluates to `true`, so the previous Boolean expression is equivalent to

```
!( false || true )
```

Consulting the tables for `||` (which is labeled OR), the computer sees that the expression inside the parentheses evaluates to `true`. Thus, the computer sees that the entire expression is equivalent to

```
!(true)
```

Consulting the tables again, the computer sees that `!(true)` evaluates to `false`, and so it concludes that false is the value of the original Boolean expression.

Display 2.2 Truth Tables

AND

Exp_1	*Exp_2*	*Exp_1 && Exp_2*
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

OR

Exp_1	*Exp_2*	*Exp_1 || Exp_2*
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

NOT

Exp	*!(Exp)*
`true`	`false`
`false`	`true`

The Boolean (`bool`) Values Are `true` and `false`

`true` and `false` are predefined constants of type `bool`. (They must be written in lowercase.) In C++, a Boolean expression evaluates to the `bool` value `true` when it is satisfied and to the `bool` value `false` when it is not satisfied.

Precedence Rules

parentheses

Boolean expressions (and arithmetic expressions) need not be fully parenthesized. If you omit parentheses, the default precedence is as follows: Perform `!` first, then perform relational operations such as `<`, then `&&`, and then `||`. However, it is a good practice to include most parentheses to make the expression easier to understand. One place where parentheses can safely be omitted is a simple string of `&&`'s or `||`'s (but not a mixture of the two). The following expression is acceptable in terms of both the C++ compiler and readability:

```
(temperature > 90) && (humidity > 0.90) && (poolGate == OPEN)
```

Since the relational operations > and == are performed before the && operation, you could omit the parentheses in the previous expression and it would have the same meaning, but including some parentheses makes the expression easier to read.

precedence rules

When parentheses are omitted from an expression, the compiler groups items according to rules known as **precedence rules**. Most of the precedence rules for C++ are given in Display 2.3. The table includes a number of operators that are not discussed until later in this book, but they are included for completeness and for those who may already know about them.

Display 2.3 **Precedence of Operators (part 1 of 2)**

Operator	Description
::	Scope resolution operator
.	Dot operator
->	Member selection
[]	Array indexing
( )	Function call
++	Postfix increment operator (placed after the variable)
--	Postfix decrement operator (placed after the variable)
++	Prefix increment operator (placed before the variable)
--	Prefix decrement operator (placed before the variable)
!	Not
-	Unary minus
+	Unary plus
*	Dereference
&	Address of
new	Create (allocate memory)
delete	Destroy (deallocate)
delete []	Destroy array (deallocate)
sizeof	Size of object
( )	Type cast
*	Multiply
/	Divide
%	Remainder (modulo)
+	Addition
-	Subtraction
<<	Insertion operator (console output)
>>	Extraction operator (console input)

Highest precedence (done first)

Lower precedence (done later)

(continued)

Display 2.3 **Precedence of Operators** (part 2 of 2)

All operators in part 2 are of lower precedence than those in part 1.

<table>
<tr><td><
>
<=
>=</td><td>Less than
Greater than
Less than or equal to
Greater than or equal to</td></tr>
<tr><td>==
!=</td><td>Equal
Not equal</td></tr>
<tr><td>&&</td><td>And</td></tr>
<tr><td>||</td><td>Or</td></tr>
<tr><td>=
+=
-=
*=
/=
%=</td><td>Assignment
Add and assign
Subtract and assign
Multiply and assign
Divide and assign
Modulo and assign</td></tr>
<tr><td>? :</td><td>Conditional operator</td></tr>
<tr><td>throw</td><td>Throw an exception</td></tr>
<tr><td>,</td><td>Comma operator</td></tr>
</table>

Lowest precedence (done last)

higher precedence

If one operation is performed before another, the operation that is performed first is said to have **higher precedence**. All the operators in a given box in Display 2.3 have the same precedence. Operators in higher boxes have higher precedence than operators in lower boxes.

When operators have the same precedences and the order is not determined by parentheses, then unary operations are done right to left. The assignment operations are also done right to left. For example, `x = y = z` means `x = (y = z)`. Other binary

operations that have the same precedences are done left to right. For example, `x + y + z` means `(x + y) + z`.

Notice that the precedence rules include both arithmetic operators such as `+` and `*` as well as Boolean operators such as `&&` and `||`. This is because many expressions combine arithmetic and Boolean operations, as in the following simple example:

```
(x + 1) > 2 || (x + 1) < -3
```

If you check the precedence rules given in Display 2.3, you will see that this expression is equivalent to

```
((x + 1) > 2) || ((x + 1) < -3)
```

because `>` and `<` have higher precedence than `||`. In fact, you could omit all the parentheses in the previous expression and it would have the same meaning, although it would be harder to read. Although we do not advocate omitting all the parentheses, it might be instructive to see how such an expression is interpreted using the precedence rules. Here is the expression without any parentheses:

```
x + 1 > 2 || x + 1 < -3
```

The precedences rules say first apply the unary `-`, then apply the `+`'s, then the `>` and the `<`, and finally apply the `||`, which is exactly what the fully parenthesized version says to do.

The previous description of how a Boolean expression is evaluated is basically correct, but in C++, the computer actually takes an occasional shortcut when evaluating a Boolean expression. Notice that in many cases you need to evaluate only the first of two subexpressions in a Boolean expression. For example, consider the following:

```
(x >= 0) && (y > 1)
```

If `x` is negative, then `(x >= 0)` is `false`. As you can see in the tables in Display 2.2, when one subexpression in an `&&` expression is `false`, then the whole expression is `false`, no matter whether the other expression is `true` or `false`. Thus, if we know that the first expression is `false`, there is no need to evaluate the second expression. A similar thing happens with `||` expressions. If the first of two expressions joined with the `||` operator is `true`, then you know the entire expression is `true`, no matter whether the second expression is `true` or `false`. The C++ language uses this fact to sometimes save itself the trouble of evaluating the second subexpression in a logical expression connected with an `&&` or `||`. C++ first evaluates the leftmost of the two expressions joined by an `&&` or `||`. If that gives it enough information to determine the final value of the expression (independent of the value of the second expression), then C++ does not bother to evaluate the second expression. This method of evaluation is called **short-circuit evaluation**.

short-circuit evaluation

Some languages other than C++ use **complete evaluation**. In complete evaluation, when two expressions are joined by an `&&` or `||`, both subexpressions are always evaluated and then the truth tables are used to obtain the value of the final expression.

complete evaluation

PITFALL: Integer Values Can Be Used as Boolean Values

integers convert to bool

C++ sometimes uses integers as if they were Boolean values and `bool` values as if they were integers. In particular, C++ converts the integer `1` to `true` and converts the integer `0` to `false`, and vice versa. The situation is even a bit more complicated than simply using `1` for `true` and `0` for `false`. The compiler will treat any nonzero number as if it were the value `true` and will treat `0` as if it were the value `false`. As long as you make no mistakes in writing Boolean expressions, this conversion causes no problems. However, when you are debugging, it might help to know that the compiler is happy to combine integers using the Boolean operators `&&`, `||`, and `!`.

For example, suppose you want a Boolean expression that is `true` provided that time has not yet run out (in some game or process). You might use the following:

```
!time > limit
```

This sounds right if you read it out loud: "not `time` greater than `limit`." The Boolean expression is wrong, however, and unfortunately, the compiler will not give you an error message. The compiler will apply the precedence rules from Display 2.3 and interpret your Boolean expression as the following:

```
(!time) > limit
```

This looks like nonsense, and intuitively it is nonsense. If the value of `time` is, for example, `36`, what could possibly be the meaning of `(!time)`? After all, that is equivalent to "not `36`." But in C++, any nonzero integer converts to `true` and `0` is converted to `false`. Thus, `!36` is interpreted as "not `true`" and so it evaluates to `false`, which is in turn converted back to `0` because we are comparing to an `int`.

What we want as the value of this Boolean expression and what C++ gives us are not the same. If `time` has a value of `36` and `limit` has a value of `60`, you want the previously displayed Boolean expression to evaluate to `true` (because it is *not* `true` that `time > limit`). Unfortunately, the Boolean expression instead evaluates as follows: `(!time)` evaluates to `false`, which is converted to `0`, so the entire Boolean expression is equivalent to

```
0 > limit
```

That in turn is equivalent to `0 > 60`, because `60` is the value of `limit`, and that evaluates to `false`. Thus, the above logical expression evaluates to `false`, when you want it to evaluate to `true`.

There are two ways to correct this problem. One way is to use the `!` operator correctly. When using the `!` operator, be sure to include parentheses around the argument. The correct way to write the above Boolean expression is

```
!(time > limit)
```

PITFALL: (continued)

Another way to correct this problem is to completely avoid using the ! operator. For example, the following is also correct and easier to read:

```
if (time <= limit)
```

You can almost always avoid using the ! operator, and some programmers advocate avoiding it as much as possible. ■

Both short-circuit evaluation and complete evaluation give the same answer, so why should you care that C++ uses short-circuit evaluation? Most of the time you need not care. As long as both subexpressions joined by the && or the || have a value, the two methods yield the same result. However, if the second subexpression is undefined, you might be happy to know that C++ uses short-circuit evaluation. Let's look at an example that illustrates this point. Consider the following statement:

```
if ( (kids != 0) && ((pieces/kids) >= 2) )
    cout << "Each child may have two pieces!";
```

If the value of `kids` is not zero, this statement involves no subtleties. However, suppose the value of `kids` is zero; consider how short-circuit evaluation handles this case. The expression `(kids!=0)` evaluates to `false`, so there would be no need to evaluate the second expression. Using short-circuit evaluation, C++ says that the entire expression is `false`, without bothering to evaluate the second expression. This prevents a run-time error, since evaluating the second expression would involve dividing by zero.

Self-Test Exercises

1. Determine the value, `true` or `false`, of each of the following Boolean expressions, assuming that the value of the variable `count` is `0` and the value of the variable `limit` is `10`. Give your answer as one of the values `true` or `false`.

   a. `(count == 0) && (limit < 20)`
   b. `count == 0 && limit < 20`
   c. `(limit > 20) || (count < 5)`
   d. `!(count == 12)`
   e. `(count == 1) && (x < y)`
   f. `(count < 10) || (x < y)`
   g. `!( ((count < 10) || (x < y)) && (count >= 0) )`
   h. `((limit / count) > 7) || (limit < 20)`
   i. `(limit < 20) || ((limit / count) > 7)`
   j. `((limit / count) > 7) && (limit < 0)`
   k. `(limit < 0) && ((limit / count) > 7)`
   l. `(5 && 7) + (!6)`

(continued)

Self-Test Exercises (continued)

2. You sometimes see numeric intervals given as

```
2 < x < 3
```

In C++ this interval does not have the meaning you may expect. Explain and give the correct C++ Boolean expression that specifies that x lies between 2 and 3.

3. Consider a quadratic expression, say

$x^2 - x - 2$

Describing where this quadratic is positive (that is, greater than 0) involves describing a set of numbers that are either less than the smaller root (which is –1) or greater than the larger root (which is 2). Write a C++ Boolean expression that is true when this formula has positive values.

4. Consider the quadratic expression

$x^2 - 4x + 3$

Describing where this quadratic is negative involves describing a set of numbers that are simultaneously greater than the smaller root (1) and less than the larger root (3). Write a C++ Boolean expression that is true when the value of this quadratic is negative.

2.2 Branching Mechanisms

When you come to a fork in the road, take it.

Attributed to YOGI BERRA

`if-else` Statements

`if-else` statement

An **`if-else` statement** chooses between two alternative statements based on the value of a Boolean expression. For example, suppose you want to design a program to compute a week's salary for an hourly employee. Assume the firm pays an overtime rate of one-and-one-half times the regular rate for all hours after the first 40 hours worked. When the employee works 40 or more hours, the pay is then equal to

```
rate * 40 + 1.5 * rate * (hours - 40)
```

`if-else` Statement

The `if-else` statement chooses between two alternative actions based on the value of a Boolean expression. The syntax is shown next. Be sure to note that the Boolean expression must be enclosed in parentheses.

SYNTAX: A SINGLE STATEMENT FOR EACH ALTERNATIVE

```
if (Boolean_Expression)
    Yes_Statement
else
    No_Statement
```

If the *`Boolean_Expression`* evaluates to `true`, then the *`Yes_Statement`* is executed. If the *`Boolean_Expression`* evaluates to `false`, then the *`No_Statement`* is executed.

SYNTAX: A SEQUENCE OF STATEMENTS FOR EACH ALTERNATIVE

```
if (Boolean_Expression)
{
    Yes_Statement_1
    Yes_Statement_2
       ...
    Yes_Statement_Last
}
else
{
    No_Statement_1
    No_Statement_2
       ...
    No_Statement_Last
}
```

EXAMPLE

```
if (myScore > yourScore)
{
    cout << "I win!\n";
    wager = wager + 100;
}
else
{
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```

However, if the employee works less than 40 hours, the correct pay formula is simply

```
rate * hours
```

The following `if-else` statement computes the correct pay for an employee whether the employee works less than 40 hours or works 40 or more hours,

```
if (hours > 40)
    grossPay = rate * 40 + 1.5 * rate * (hours - 40);
else
    grossPay = rate * hours;
```

The syntax for an `if-else` statement is given in the accompanying box. If the Boolean expression in parentheses (after the `if`) evaluates to `true`, then the statement before the `else` is executed. If the Boolean expression evaluates to `false`, the statement after the `else` is executed.

Notice that an `if-else` statement has smaller statements embedded in it. Most of the statement forms in C++ allow you to make larger statements out of smaller statements by combining the smaller statements in certain ways.

parentheses

Remember that when you use a Boolean expression in an `if-else` statement, the Boolean expression must be enclosed in parentheses.

Compound Statements

`if-else` with multiple statements

You will often want the branches of an `if-else` statement to execute more than one statement each. To accomplish this, enclose the statements for each branch between a pair of braces, { and }, as indicated in the second syntax template in the box entitled "`if-else` Statement." A list of statements enclosed in a pair of braces is called a **compound statement**. A compound statement is treated as a single statement by C++ and may be used anywhere that a single statement may be used. (Thus, the second syntax template in the box entitled "`if-else` Statement" is really just a special case of the first one.)

compound statement

There are two commonly used ways of indenting and placing braces in `if-else` statements, which are illustrated here:

```
if (myScore > yourScore)
{
    cout << "I win!\n";
    wager = wager + 100;
}
else
{
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```

and

```
if (myScore > yourScore){
    cout << "I win!\n";
    wager = wager + 100;
} else {
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```

The only differences are the placement of braces. We find the first form easier to read and therefore prefer it. The second form saves lines, so some programmers prefer the second form or some minor variant of it.

PITFALL: Using = in Place of = =

Unfortunately, you can write many things in C++ that you would think are incorrectly formed C++ statements but which turn out to have some obscure meaning. This means that if you mistakenly write something that you would expect to produce an error message, you may find that the program compiles and runs with no error messages but gives incorrect output. Since you may not realize you wrote something incorrectly, this can cause serious problems. For example, consider an `if-else` statement that begins as follows:

```
if (x = 12)
    Do_Something
else
    Do_Something_Else
```

Suppose you wanted to test to see if the value of `x` is equal to `12`, so that you really meant to use `= =` rather than `=`. You might think the compiler would catch your mistake. The expression

```
x = 12
```

is not something that is satisfied or not. It is an assignment statement, so surely the compiler will give an error message. Unfortunately, that is not the case. In C++ the expression `x = 12` is an expression that returns a value, just like `x + 12` or `2 + 3`. An assignment expression's value is the value transferred to the variable on the left. For example, the value of `x = 12` is `12`. We saw in our discussion of Boolean value compatibility that nonzero `int` values are converted to `true`. If you use `x = 12` as the Boolean expression in an `if-else` statement, the Boolean expression will always evaluate to `true`.

This error is very hard to find, because it looks right. The compiler can find the error without any special instructions if you put the `12` on the left side of the comparison: `12 == x` will produce no error message, but `12 = x` will generate an error message.

Self-Test Exercises

5. Does the following sequence produce division by zero?

```
j = -1;
if ((j > 0) && (1/(j + 1) > 10))
   cout << i << endl;
```

6. Write an `if-else` statement that outputs the word `High` if the value of the variable `score` is greater than `100` and `Low` if the value of `score` is at most `100`. The variable `score` is of type `int`.

7. Suppose `savings` and `expenses` are variables of type `double` that have been given values. Write an `if-else` statement that outputs the word `Solvent`, decreases the value of `savings` by the value of `expenses`, and sets the value of `expenses` to zero provided that `savings` is at least as large as `expenses`. If, however, `savings` is less than `expenses`, the `if-else` statement simply outputs the word `Bankrupt` and does not change the value of any variables.

8. Write an `if-else` statement that outputs the word `Passed` provided the value of the variable `exam` is greater than or equal to `60` and also the value of the variable `programsDone` is greater than or equal to `10`. Otherwise, the `if-else` statement outputs the word `Failed`. The variables `exam` and `programsDone` are both of type `int`.

9. Write an `if-else` statement that outputs the word `Warning` provided that either the value of the variable `temperature` is greater than or equal to `100`, or the value of the variable `pressure` is greater than or equal to `200`, or both. Otherwise, the `if-else` statement outputs the word `OK`. The variables `temperature` and `pressure` are both of type `int`.

10. What is the output of the following? Explain your answers.

a.
```
if(0)
  cout << "0 is true";
else
  cout << "0 is false";
cout << endl;
```

b.
```
if(1)
  cout << "1 is true";
else
  cout << "1 is false";
cout << endl;
```

c.
```
if(-1)
  cout << "-1 is true";
else
  cout << "-1 is false";
cout << endl;
```

Note: This is an exercise only. This is *not* intended to illustrate programming style you should follow.

Omitting the `else`

Sometimes you want one of the two alternatives in an `if-else` statement to do nothing at all. In C++ this can be accomplished by omitting the `else` part. These sorts of statements are referred to as **`if` statements** to distinguish them from `if-else` statements. For example, the first of the following two statements is an `if` statement:

if statement

```
if (sales >= minimum)
    salary = salary + bonus;
cout << "salary = $" << salary;
```

If the value of `sales` is greater than or equal to the value of `minimum`, the assignment statement is executed and then the following `cout` statement is executed. On the other hand, if the value of `sales` is less than `minimum`, then the embedded assignment statement is not executed. Thus, the `if` statement causes no change (that is, no bonus is added to the base salary), and the program proceeds directly to the `cout` statement.

Nested Statements

As you have seen, `if-else` statements and `if` statements contain smaller statements within them. Thus far we have used compound statements and simple statements such as assignment statements as these smaller substatements, but there are other possibilities. In fact, any statement at all can be used as a subpart of an `if-else` statement or of other statements that have one or more statements within them.

indenting

When nesting statements, you normally indent each level of nested substatements, although there are some special situations (such as a multiway `if-else` branch) where this rule is not followed.

Multiway `if-else` Statement

multiway `if-else`

The multiway `if-else` statement is not really a different kind of C++ statement. It is simply an ordinary `if-else` statement nested inside `if-else` statements, but it is thought of as a kind of statement and is indented differently from other nested statements so as to reflect this thinking.

The syntax for a multiway `if-else` statement and a simple example are given in the accompanying box. Note that the Boolean expressions are aligned with one another, and their corresponding actions are also aligned with each other. This makes it easy to see the correspondence between Boolean expressions and actions. The Boolean expressions are evaluated in order until a `true` Boolean expression is found. At that point the evaluation of Boolean expressions stops, and the action corresponding to the first `true` Boolean expression is executed. The final `else` is optional. If there is a final `else` and all the Boolean expressions are `false`, the final action is executed. If there is no final `else` and all the Boolean expressions are `false`, then no action is taken.

Multiway if-else Statement

SYNTAX

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
        .
        .
        .
else if (Boolean_Expression_n)
    Statement_n
else
    Statement_For_All_Other_Possibilities
```

EXAMPLE

```
if ((temperature < -10) && (day == SUNDAY))
    cout << "Stay home.";
else if (temperature < -10) // and day != SUNDAY
    cout << "Stay home, but call work.";
else if (temperature <= 0) // and temperature >= -10
    cout << "Dress warm.";
else // temperature > 0
    cout << "Work hard and play hard.";
```

The Boolean expressions are checked in order until the first `true` Boolean expression is encountered, and then the corresponding statement is executed. If none of the Boolean expressions is `true`, then the *`Statement_For_All_Other_Possibilities`* is executed.

The `switch` Statement

switch statement

The **`switch` statement** is the only other kind of C++ statement that implements multiway branches. Syntax for a `switch` statement and a simple example are shown in the accompanying box.

controlling expression

When a `switch` statement is executed, one of a number of different branches is executed. The choice of which branch to execute is determined by a **controlling expression** given in parentheses after the keyword `switch`. The controlling expression for a `switch` statement must always return either a `bool` value, an `enum` constant (discussed later in this chapter), one of the integer types, or a character. When the `switch` statement is executed, this controlling expression is evaluated and the computer looks at the constant values given after the various occurrences of the `case` identifiers. If it finds a constant that equals the value of the controlling expression, it executes the code for that `case`. You cannot have two occurrences of `case` with the same constant value after them because that would create an ambiguous instruction.

switch Statement

SYNTAX

```
switch (Controlling_Expression)
{
    case Constant_1:
        Statement_Sequence_1
        break;
    case Constant_2:
        Statement_Sequence_2
        break;
                    .
                    .
                    .
    case Constant_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
}
```

*You need not place a **break** statement in each case. If you omit a **break**, that case continues until a **break** (or the end of the **switch** statement) is reached.*

EXAMPLE

```
int vehicleClass;
double toll;
cout << "Enter vehicle class: ";
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break;
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
}
```

*If you forget this **break**, then passenger cars will pay $ 1.50.*

break statement

The switch statement ends when either a break statement is encountered or the end of the switch statement is reached. A **break statement** consists of the keyword break followed by a semicolon. When the computer executes the statements after a case label, it continues until it reaches a break statement. When the computer encounters a break statement, the switch statement ends. If you omit the break statements, then after executing the code for one case, the computer will go on to execute the code for the next case.

Note that you can have two case labels for the same section of code, as in the following portion of a switch statement:

```
case 'A':
case 'a':
    cout << "Excellent. "
         << "You need not take the final.\n";
    break;
```

Since the first case has no break statement (in fact, no statement at all), the effect is the same as having two labels for one case, but C++ syntax requires one keyword case for each label, such as 'A' and 'a'.

default

If no case label has a constant that matches the value of the controlling expression, then the statements following the default label are executed. You need not have a default section. If there is no default section and no match is found for the value of the controlling expression, then nothing happens when the switch statement is executed. However, it is safest to always have a default section. If you think your case labels list all possible outcomes, then you can put an error message in the default section.

Self-Test Exercises

11. What output will be produced by the following code?

```
int x = 2;
cout << "Start\n";
if (x <= 3)
    if (x != 0)
        cout << "Hello from the second if.\n";
    else
        cout << "Hello from the else.\n";
cout << "End\n";

cout << "Start again\n";
if (x > 3)
    if (x != 0)
        cout << "Hello from the second if.\n";
    else
        cout << "Hello from the else.\n";
cout << "End again\n";
```

Self-Test Exercises (continued)

12. What output will be produced by the following code?

```
int extra = 2;
if (extra < 0)
    cout << "small";
else if (extra == 0)
    cout << "medium";
else
    cout << "large";
```

13. What would be the output in Self-Test Exercise 12 if the assignment were changed to the following?

```
int extra = -37;
```

14. What would be the output in Self-Test Exercise 12 if the assignment were changed to the following?

```
int extra = 0;
```

15. Write a multiway `if-else` statement that classifies the value of an `int` variable `n` into one of the following categories and writes out an appropriate message.

```
n < 0 or 0 ≤ n ≤ 100 or n > 100
```

PITFALL: Forgetting a `break` in a `switch` Statement

If you forget a `break` in a `switch` statement, the compiler will not issue an error message. You will have written a syntactically correct `switch` statement, but it will not do what you intended it to do. Notice the annotation in the example in the box entitled "`switch` Statement." ■

TIP: Use `switch` Statements for Menus

The multiway `if-else` statement is more versatile than the `switch` statement, and you can use a multiway `if-else` statement anywhere you can use a `switch` statement. However, sometimes the `switch` statement is clearer. For example, the `switch` statement is perfect for implementing menus. Each branch of the `switch` statement can be one menu choice. ■

Enumeration Types

enumeration type

An **enumeration type** is a type whose values are defined by a list of constants of type `int`. An enumeration type is very much like a list of declared constants. Enumeration types can be handy for defining a list of identifiers to use as the `case` labels in a `switch` statement.

When defining an enumeration type, you can use any `int` values and can define any number of constants. For example, the following enumeration type defines a constant for the length of each month:

```
enum MonthLength { JAN_LENGTH = 31, FEB_LENGTH = 28,
    MAR_LENGTH = 31, APR_LENGTH = 30, MAY_LENGTH = 31,
    JUN_LENGTH = 30, JUL_LENGTH = 31, AUG_LENGTH = 31,
    SEP_LENGTH = 30, OCT_LENGTH = 31, NOV_LENGTH = 30,
    DEC_LENGTH = 31 };
```

As this example shows, two or more named constants in an enumeration type can receive the same `int` value.

If you do not specify any numeric values, the identifiers in an enumeration type definition are assigned consecutive values beginning with 0. For example, the type-definition

```
enum Direction { NORTH = 0, SOUTH = 1, EAST = 2, WEST = 3 };
```

is equivalent to

```
enum Direction { NORTH, SOUTH, EAST, WEST };
```

The form that does not explicitly list the `int` values is normally used when you just want a list of names and do not care about what values they have.

Suppose you initialize an enumeration constant to some value, say

```
enum MyEnum { ONE = 17, TWO, THREE, FOUR = -3, FIVE };
```

then `ONE` takes the value 17; `TWO` takes the next `int` value, 18; `THREE` takes the next value, 19; `FOUR` takes –3; and `FIVE` takes the next value, –2. In short, the default for the first enumeration constant is 0. The rest increase by 1 unless you set one or more of the enumeration constants.

Although the constants in an enumeration type are given as `int` values and can be used as integers in many contexts, remember that an enumeration type is a separate type and treat it as a type different from the type `int`. Use enumeration types as labels and avoid doing arithmetic with variables of an enumeration type.

C++11 introduced a new version of enumerations called **strong enums** or **enum classes** that avoids some of the problems with conventional enums. For example, you may not want an enum to act as an integer, since this opens up the possibility of storing a value not associated with a declared constant in the enumeration type variable. Additionally, enums are global in scope so you can't have the same enum value twice. To define a strong enum, add the word `class` after `enum`. You can qualify an enum value by providing the enum name followed by two colons followed by the value. For example:

```
enum class Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
enum class Weather { Rain, Sun };
Days d = Days::Tue;
Weather w = Weather::Sun;
```

The variables `d` and `w` are not integers so we can't treat them as such. For example, it would be illegal to check `if (d == 0)`, whereas this is legal in a traditional enum. It is legal to check `if (d == Days::Sun)`.

The Conditional Operator

conditional operator

It is possible to embed a conditional inside an expression by using a ternary operator known as the **conditional operator** (also called the *ternary operator* or *arithmetic* `if`). Its use is reminiscent of an older programming style, and we do not advise using it. It is included here for the sake of completeness (and in case you disagree with our programming style).

The conditional operator is a notational variant on certain forms of the `if-else` statement. This variant is illustrated as follows. Consider the statement

```
if (n1 > n2)
    max = n1;
else
    max = n2;
```

This can be expressed using the conditional operator as follows:

```
max = (n1 > n2) ? n1 : n2;
```

conditional operator expression

The expression on the right-hand side of the assignment statement is the **conditional operator expression**:

```
(n1 > n2) ? n1 : n2
```

The `?` and `:` together form a ternary operator known as the conditional operator. A conditional operator expression starts with a Boolean expression followed by a `?` and then followed by two expressions separated with a colon. If the Boolean expression is `true`, then the first of the two expressions is returned; otherwise, the second of the two expressions is returned.

Self-Test Exercises

16. Given the following declaration and output statement, assume that this has been embedded in a correct program and is run. What is the output?

```
enum Direction { N, S, E, W };
// ...
cout << W << " " << E << " " << S << " " << N << endl;
```

17. Given the following declaration and output statement, assume that this has been embedded in a correct program and is run. What is the output?

```
enum Direction { N = 5, S = 7, E = 1, W };
// ...
cout << W << " " << E << " " << S << " " << N << endl;
```

2.3 Loops

Few tasks are more like the torture of Sisyphus than housework, with its endless repetition: the clean becomes soiled, the soiled is made clean, over and over, day after day.

SIMONE DE BEAUVOIR, *The Second Sex. Trans. Constance Borde and Sheila Malovany-Chevalier. 1949*

loop body
iteration

Looping mechanisms in C++ are similar to those in other high-level languages. The three C++ loop statements are the `while` statement, the `do-while` statement, and the `for` statement. The same terminology is used with C++ as with other languages. The code that is repeated in a loop is called the **loop body**. Each repetition of the loop body is called an **iteration** of the loop.

The `while` and `do-while` Statements

`while` and `do-while` compared

The syntax for the `while` statement and its variant, the `do-while` statement, is given in the accompanying box. In both cases, the multistatement body syntax is a special case of the syntax for a loop with a single-statement body. The multistatement body is a single compound statement. Examples of a `while` statement and a `do-while` statement are given in Displays 2.4 and 2.5.

Syntax for `while` and `do-while` Statements

A `while` STATEMENT WITH A SINGLE-STATEMENT BODY

```
while (Boolean_Expression)
    Statement
```

A `while` STATEMENT WITH A MULTISTATEMENT BODY

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
      .
      .
      .
    Statement_Last
}
```

A `do-while` STATEMENT WITH A SINGLE-STATEMENT BODY

```
do
    Statement
while (Boolean_Expression);
```

A `do-while` STATEMENT WITH A MULTISTATEMENT BODY

```
do
{
    Statement_1
    Statement_2
      .
      .
      .
    Statement_Last
} while (Boolean_Expression);
```

Do not forget the final semicolon.

Display 2.4 Example of a `while` Statement

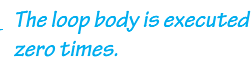
```
1   #include <iostream>
2   using namespace std;

3   int main( )

4   {
5       int countDown;

6       cout << "How many greetings do you want? ";
7       cin >> countDown;

8       while (countDown > 0)
9       {
10          cout << "Hello ";
11          countDown = countDown - 1;
12      }

13      cout << endl;
14      cout << "That's all!\n";

15      return 0;
16  }
```

Sample Dialogue 1

```
How many greetings do you want? 3
Hello Hello Hello
That's all!
```

Sample Dialogue 2

```
How many greetings do you want? 0
That's all!
```



The important difference between the `while` and `do-while` loops involves *when* the controlling Boolean expression is checked. With a `while` statement, the Boolean expression is checked *before* the loop body is executed. If the Boolean expression evaluates to `false`, the body is not executed at all. With a `do-while` statement, the body of the loop is executed first and the Boolean expression is checked *after* the loop body is executed. Thus, the `do-while` statement always executes the loop body at least once. After this start-up, the `while` loop and the `do-while` loop behave the same. After each iteration of the loop body, the Boolean expression is again checked; if it is `true`, the loop is iterated again. If it has changed from `true` to `false`, the loop statement ends.

Display 2.5 Example of a `do-while` Statement

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

3  int main( )
4  {
5      int countDown;

6      cout << "How many greetings do you want? ";
7      cin >> countDown;

8      do
9      {
10         cout << "Hello ";
11         countDown = countDown - 1;
12     } while (countDown > 0);

13     cout << endl;
14     cout << "That's all!\n";

15     return 0;
16 }
```

Sample Dialogue 1

```
How many greetings do you want? 3
Hello Hello Hello
That's all!
```

Sample Dialogue 2

```
How many greetings do you want? 0
Hello
That's all!
```

The loop body is always executed at least once.

executing the body zero times

The first thing that happens when a `while` loop is executed is that the controlling Boolean expression is evaluated. If the Boolean expression evaluates to `false` at that point, the body of the loop is never executed. It may seem pointless to execute the body of a loop zero times, but that is sometimes the desired action. For example, a `while` loop is often used to sum a list of numbers, but the list could be empty. To be more specific, a checkbook balancing program might use a `while` loop to sum the values of all the checks you have written in a month–but you might take a month's vacation and write no checks at all. In that case, there are zero numbers to sum and so the loop is iterated zero times.