

GLOBAL  
EDITION



# Problem Solving with C++

TENTH EDITION

Walter Savitch



Pearson



# Digital Resources for Students

Your new textbook provides 12-month access to digital resources that may include VideoNotes (step-by-step video tutorials on programming concepts), source code, web chapters, quizzes, and more. Refer to the preface in the textbook for a detailed list of resources.

Follow the instructions below to register for the Companion Website for Walter Savitch's *Problem Solving with C++*, Tenth Edition, Global Edition.

1. Go to [www.pearsonglobaleditions.com/savitch](http://www.pearsonglobaleditions.com/savitch).
2. Enter the title of your textbook or browse by author name.
3. Click Companion Website.
4. Click Register and follow the on-screen instructions to create a login name and password.

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

Use the login name and password you created during registration to start using the online resources that accompany your textbook.

## **IMPORTANT:**

This prepaid subscription does not include access to MyProgrammingLab, which is available at [www.myprogramminglab.com](http://www.myprogramminglab.com) for purchase.

This access code can only be used once. This subscription is valid for 12 months upon activation and is not transferable. If the access code has already been revealed it may no longer be valid.

For technical support go to <https://support.pearson.com/getsupport/>

This page intentionally left blank

# PROBLEM SOLVING with C++

---

This page intentionally left blank

Tenth Edition  
Global Edition

**PROBLEM SOLVING**

with

**C++**

**Walter J. Savitch**

*UNIVERSITY OF CALIFORNIA, SAN DIEGO*

CONTRIBUTOR

**Kenrick Mock**

*UNIVERSITY OF ALASKA, ANCHORAGE*



330 Hudson Street, New York, NY 10013

Senior Vice President Courseware Portfolio Management:	Marcia J. Horton
Director, Portfolio Management: Engineering,	
Computer Science & Global Editions:	Julian Partridge
Portfolio Manager:	Matt Goldstein
Assistant Acquisitions Editor, Global Edition:	Aditee Agarwal
Portfolio Management Assistant:	Kristy Alaura
Field Marketing Manager:	Demetrius Hall
Product Marketing Manager:	Yvonne Vannatta
Managing Producer, ECS and Math:	Scott Disanno
Content Producer:	Sandra L. Rodriguez
Project Editor, Global Edition:	K.K. Neelakantan
Senior Manufacturing Controller, Global Edition:	Angela Hawksbee
Manager, Media Production, Global Edition:	Vikram Kumar
Cover Designer:	Lumina Datamatics, Inc.
Cover Photo:	Iana Chyrva/Shutterstock

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

Pearson Education Limited

KAO Two

KAO Park

Harlow

CM17 9NA

United Kingdom

and Associated Companies throughout the world

Visit us on the World Wide Web at: [www.pearsonglobaleditions.com](http://www.pearsonglobaleditions.com)

© Pearson Education Limited 2018

The rights 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 Problem Solving with C++, 10th Edition, ISBN 978-0-13-444828-2 by Walter Savitch published by Pearson Education © 2018.*

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: 1-292-22282-4

ISBN 13: 978-1-292-22282-0

Typeset by iEnergizer Aptara®, Ltd.

Printed and bound in Malaysia

# Preface

---

This book is meant to be used in a first course in programming and computer science using the C++ language. It assumes no previous programming experience and no mathematics beyond high school algebra.

If you have used the previous edition of this book, you should read the following section that explains the changes to this tenth edition and then you can skip the rest of this preface. If you are new to this book, the rest of this preface will give you an overview of the book.

## Changes to the Tenth Edition

This tenth edition presents the same programming philosophy as the ninth edition. All of the material from the ninth edition remains, but with the following enhancements:

- Consistent use of camelCase notation rather than underscore\_case throughout the text.
- Discussion in Chapter 10 of shallow vs. deep copy.
- Additional material in Chapter 12 and 17 on compiling templates with header files.
- Additional material in Chapter 18 on the `std::array` class, regular expressions, threads, and smart pointers in C++11.
- Correction of errata and edits for clarity such as indicating preferred methods for file I/O, naming of terminology, better definition of encapsulation, and removing material that is now standard in C++11 and higher.
- Ten 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.

If you are an instructor already using the ninth edition, you can continue to teach your course almost without change.

## Flexibility in Topic Ordering

This book was written to allow instructors wide latitude in reordering the material. To illustrate this flexibility, we suggest two alternative ways to order



the topics. There is no loss of continuity when the book is read in either of these ways. To ensure this continuity when you rearrange material, you may need to move sections rather than entire chapters. However, only large sections in convenient locations are moved. To help customize a particular order for any class's needs, the end of this preface contains a dependency chart, and each chapter has a "Prerequisites" section that explains what material needs to be covered before each section in that chapter.

### **Reordering 1: Earlier Classes**

To effectively design classes, a student needs some basic tools such as control structures and function definitions. This basic material is covered in Chapters 1 through 6. After completing Chapter 6, students can begin to write their own classes. One possible reordering of chapters that allows for such early coverage of classes is the following:

*Basics:* Chapters 1, 2, 3, 4, 5, and 6. This material covers all control structures, function definitions, and basic file I/O. Chapter 3, which covers additional control structures, could be deferred if you wish to cover classes as early as possible.

*Classes and namespaces:* Chapter 10, Sections 11.1 and 11.2 of Chapter 11, and Chapter 12. This material covers defining classes, friends, overloaded operators, and namespaces.

*Arrays, strings and vectors:* Chapters 7 and 8

*Pointers and dynamic arrays:* Chapter 9

*Arrays in classes:* Sections 11.3 and 11.4 of Chapter 11

*Inheritance:* Chapter 15

*Recursion:* Chapter 14. (Alternately, recursion may be moved to later in the course.)

*Pointers and linked lists:* Chapter 13

Any subset of the following chapters may also be used:

*Exception handling:* Chapter 16

*Templates:* Chapter 17

*Standard Template Library:* Chapter 18

### **Reordering 2: Classes Slightly Later but Still Early**

This version covers all control structures and the basic material on arrays before doing classes, but classes are covered later than the previous ordering and slightly earlier than the default ordering.

*Basics:* Chapters 1, 2, 3, 4, 5, and 6. This material covers all control structures, function definitions, and the basic file I/O.

*Arrays and strings:* Chapter 7, Sections 8.1 and 8.2 of Chapter 8

*Classes and namespaces:* Chapter 10, Sections 11.1 and 11.2 of Chapter 11, and Chapter 12. This material covers defining classes, friends, overloaded operators, and namespaces.

*Pointers and dynamic arrays:* Chapter 9

*Arrays in classes:* Sections 11.3 and 11.4 of Chapter 11

*Inheritance:* Chapter 15

*Recursion:* Chapter 14. (Alternately, recursion may be moved to later in the course.)

*Vectors:* Chapter 8.3

*Pointers and linked lists:* Chapter 13

Any subset of the following chapters may also be used:

*Exception handling:* Chapter 16

*Templates:* Chapter 17

*Standard Template Library:* Chapter 18

## Accessibility 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 to be clear and correct when read by an instructor or other experienced programmer. The material needs to be presented in a way that is accessible to beginning students. In this introductory textbook, I have endeavored to write in a way that students find clear and friendly. Reports from the many students who have used the earlier editions of this book confirm that this style makes the material clear and often even enjoyable to students.

## ANSI/ISO C++ Standard

This edition is fully compatible with compilers that meet the latest ANSI/ISO C++ standard. At the time of this writing the latest standard is C++14.

## Advanced Topics

Many “advanced topics” are becoming part of a standard CS1 course. Even if they are not part of a course, it is good to have them available in the text as enrichment material. This book offers a number of advanced topics that can be integrated into a course or left as enrichment topics. It gives thorough coverage of C++ templates, inheritance (including virtual functions), exception handling, the STL (Standard Template Library), threads, regular expressions, and smart pointers. Although this book uses libraries and teaches students the importance of libraries, it does not require any nonstandard libraries. This book uses only libraries that are provided with essentially all C++ implementations.

## Dependency Chart

The dependency chart on the next page shows possible orderings of chapters and subsections. A line joining two boxes means that the upper box must be covered before the lower box. Any ordering that is consistent with this partial ordering can be read without loss of continuity. If a box contains a section number or numbers, then the box refers only to those sections and not to the entire chapter.

## Summary Boxes

Each major point is summarized in a boxed section. These boxed sections are spread throughout each chapter.

## Self-Test Exercises

Each chapter contains numerous Self-Test Exercises at strategic points. Complete answers for all the Self-Test Exercises are given at the end of each chapter.



VideoNote

## VideoNotes

VideoNotes are designed for teaching students key programming concepts and techniques. These short step-by-step videos demonstrate how to solve problems from design through coding. VideoNotes allow for self-paced instruction with easy navigation including the ability to select, play, rewind, fast-forward, and stop within each VideoNote exercise.

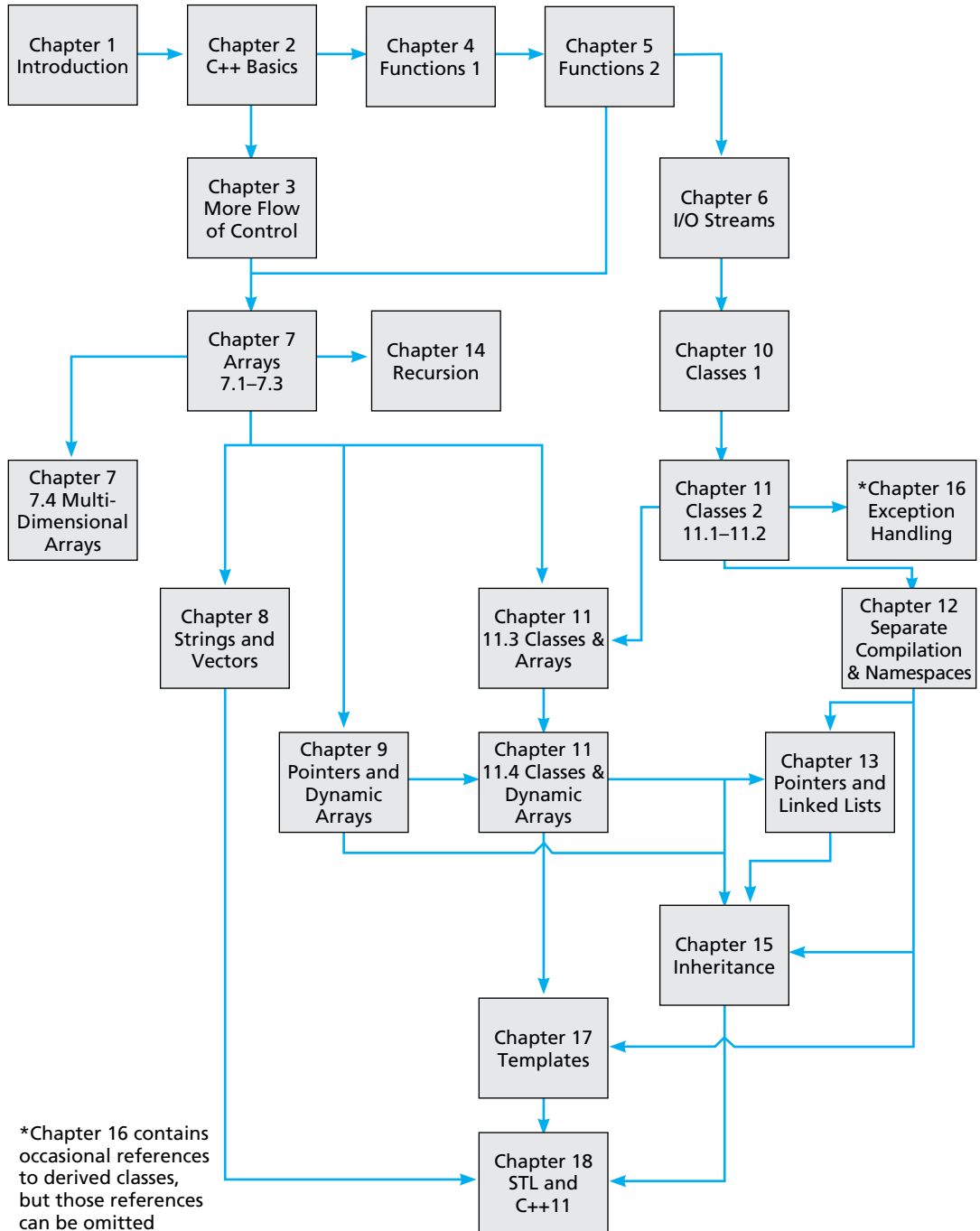
## Online Practice and Assessment with MyProgrammingLab

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.

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

# DISPLAY P.1 Dependency Chart



## Support Material

There is support material available to all users of this book and additional material available only to qualified instructors.

### Materials Available to All Users of this Book

- Source Code from the book
- PowerPoint slides
- VideoNotes
- To access these materials, go to: [www.pearsonglobaleditions.com/savitch](http://www.pearsonglobaleditions.com/savitch)

### Resources Available to Qualified Instructors Only

Visit Pearson Education's instructor resource center at [www.pearsonglobaleditions.com/savitch](http://www.pearsonglobaleditions.com/savitch) to access the following instructor resources:

- Instructor's Resource Guide—including chapter-by-chapter teaching hints, quiz questions with solutions, and solutions to many programming projects
- Test Bank and Test Generator
- PowerPoint Lectures—including programs and art from the text
- Lab Manual

## Contact Us

Your comments, suggestions, questions, and corrections are always welcome. Please e-mail them to [savitch.programming.cpp@gmail.com](mailto:savitch.programming.cpp@gmail.com)

## Acknowledgments

Numerous individuals and groups have provided me with suggestions, discussions, and other help in preparing this textbook. Much of the first edition of this book was written while I was visiting the Computer Science Department at the University of Colorado in Boulder. The remainder of the writing on the first edition and the work on subsequent editions was done in the Computer Science and Engineering Department at the University of California, San Diego (UCSD). I am grateful to these institutions for providing a conducive environment for teaching this material and writing this book.

I extend a special thanks to all the individuals who have contributed critiques or programming projects for this or earlier editions and drafts of this book. In alphabetical order, they are: Alex Feldman, Amber Settle, Andrew Burt, Andrew Haas, Anne Marchant, Barney MacCabe, Bob Holloway, Bob Matthews, Brian R. King, Bruce Johnston, Carol Roberts, Charles Dowling, Claire Bono, Cynthia Martincic, David Feinstein, David Teague, Dennis Heckman, Donald Needham, Doug Cosman, Dung Nguyen, Edward Carr, Eitan M. Gurari, Ethan Munson, Firooz Khosraviyani, Frank Moore, Gilliean Lee, Huzefa Kagdi, James Stepleton, Jeff Roach, Jeffrey Watson, Jennifer Perkins,

Jerry Weltman, Joe Faletti, Joel Cohen, John J. Westman, John Marsaglia, John Russo, Joseph Allen, Joseph D. Oldham, Jerrold Grossman, Jesse Morehouse, Karla Chaveau, Ken Rockwood, Larry Johnson, Len Garrett, Linda F. Wilson, Mal Gunasekera, Marianne Lepp, Matt Johnson, Michael Keenan, Michael Main, Michal Sramka, Naomi Shapiro, Nat Martin, Noah Aydin, Nisar Hundewale, Paul J. Kaiser, Paul Kube, Paulo Franca, Richard Borie, Scot Drysdale, Scott Strong, Sheila Foster, Steve Mahaney, Susanne Sherba, Thomas Judson, Walter A. Manrique, Wei Lian Chen, and Wojciech Komornicki.

I extend a special thanks to the many instructors who used early editions of this book. Their comments provided some of the most helpful reviewing that the book received.

Finally, I thank Kenrick Mock who implemented the changes in this edition. He had the almost impossible task of pleasing me, my editor, and his own sensibilities, and he did a superb job of it.

*Walter Savitch*

## Acknowledgments for the Global Edition

Pearson would like to thank and acknowledge Bradford Heap, University of New South Wales, for contributing to the Global Edition, and Kaushik Goswami, St. Xavier's College Kolkata, Ela Kashyap, and Sandeep Singh, Jaypee Institute of Technology, for reviewing the Global Edition.



This page intentionally left blank

# MyProgrammingLab™

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

## PROGRAMMING PRACTICE

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

## IMMEDIATE, PERSONALIZED FEEDBACK

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

## GRADUATED COMPLEXITY

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

## DYNAMIC ROSTER

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

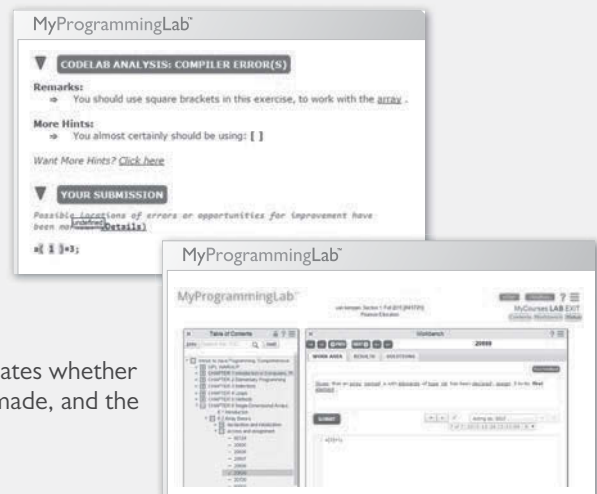
## PEARSON eTEXT

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

## STEP-BY-STEP VIDEONOTE TUTORIALS

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

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



This page intentionally left blank

# Brief Contents

---

<b>Chapter 1</b>	<b>Introduction to Computers and C++ Programming</b>	<b>33</b>
<b>Chapter 2</b>	<b>C++ Basics</b>	<b>71</b>
<b>Chapter 3</b>	<b>More Flow of Control</b>	<b>143</b>
<b>Chapter 4</b>	<b>Procedural Abstraction and Functions That Return a Value</b>	<b>213</b>
<b>Chapter 5</b>	<b>Functions for All Subtasks</b>	<b>283</b>
<b>Chapter 6</b>	<b>I/O Streams as an Introduction to Objects and Classes</b>	<b>339</b>
<b>Chapter 7</b>	<b>Arrays</b>	<b>411</b>
<b>Chapter 8</b>	<b>Strings and Vectors</b>	<b>485</b>
<b>Chapter 9</b>	<b>Pointers and Dynamic Arrays</b>	<b>541</b>
<b>Chapter 10</b>	<b>Defining Classes</b>	<b>575</b>
<b>Chapter 11</b>	<b>Friends, Overloaded Operators, and Arrays in Classes</b>	<b>653</b>
<b>Chapter 12</b>	<b>Separate Compilation and Namespaces</b>	<b>737</b>

<b>Chapter 13</b>	<b>Pointers and Linked Lists</b>	773
<b>Chapter 14</b>	<b>Recursion</b>	823
<b>Chapter 15</b>	<b>Inheritance</b>	867
<b>Chapter 16</b>	<b>Exception Handling</b>	927
<b>Chapter 17</b>	<b>Templates</b>	959
<b>Chapter 18</b>	<b>Standard Template Library and C++11</b>	991
<b>Appendices</b>		
<b>1</b>	<b>C++ Keywords</b>	1067
<b>2</b>	<b>Precedence of Operators</b>	1068
<b>3</b>	<b>The ASCII Character Set</b>	1070
<b>4</b>	<b>Some Library Functions</b>	1071
<b>5</b>	<b>Inline Functions</b>	1078
<b>6</b>	<b>Overloading the Array Index Square Brackets</b>	1079
<b>7</b>	<b>The this Pointer</b>	1081
<b>8</b>	<b>Overloading Operators as Member Operators</b>	1084
<b>Credits</b>		1086
<b>Index</b>		1089

# Contents

---

## Chapter 1 Introduction to Computers and C++ Programming 33

### 1.1 COMPUTER SYSTEMS 34

Hardware 34

Software 39

High-Level Languages 40

Compilers 41

*History Note* 44

### 1.2 PROGRAMMING AND PROBLEM-SOLVING 44

Algorithms 44

Program Design 47

Object-Oriented Programming 48

The Software Life Cycle 49

### 1.3 INTRODUCTION TO C++ 50

Origins of the C++ Language 50

A Sample C++ Program 51

*Pitfall:* Using the Wrong Slash in `\n` 55

*Programming Tip:* Input and Output Syntax 55

Layout of a Simple C++ Program 56

*Pitfall:* Putting a Space Before the `include` File Name 58

Compiling and Running a C++ Program 58

*Pitfall:* Compiling a C++11 Program 59

*Programming Tip:* Getting Your Program to Run 59

### 1.4 TESTING AND DEBUGGING 61

Kinds of Program Errors 62

*Pitfall:* Assuming Your Program Is Correct 63

Chapter Summary 64

Answers to Self-Test Exercises 65

Practice Programs 67

Programming Projects 68



## Chapter 2 C++ Basics 71

### 2.1 VARIABLES AND ASSIGNMENTS 72

Variables 72

Names: Identifiers 74

Variable Declarations 77

Assignment Statements 77

*Pitfall:* Uninitialized Variables 79

*Programming Tip:* Use Meaningful Names 81

### 2.2 INPUT AND OUTPUT 82

Output Using `cout` 82

Include Directives and Namespaces 84

Escape Sequences 85

*Programming Tip:* End Each Program with a `\n` or `endl` 87

Formatting for Numbers with a Decimal Point 87

Input Using `cin` 88

Designing Input and Output 90

*Programming Tip:* Line Breaks in I/O 90

### 2.3 DATA TYPES AND EXPRESSIONS 92

The Types `int` and `double` 92

Other Number Types 94

C++11 Types 95

The Type `char` 96

The Type `bool` 98

Introduction to the Class `string` 98

Type Compatibilities 100

Arithmetic Operators and Expressions 101

*Pitfall:* Whole Numbers in Division 104

More Assignment Statements 106

### 2.4 SIMPLE FLOW OF CONTROL 106

A Simple Branching Mechanism 107

*Pitfall:* Strings of Inequalities 112

*Pitfall:* Using `=` in place of `==` 113

Compound Statements 114

Simple Loop Mechanisms 116

Increment and Decrement Operators 119

*Programming Example:* Charge Card Balance 121

*Pitfall:* Infinite Loops 122

## 2.5 PROGRAM STYLE 125

Indenting 125

Comments 125

Naming Constants 127

Chapter Summary 130

Answers to Self-Test Exercises 130

Practice Programs 135

Programming Projects 137

## Chapter 3 More Flow of Control 143

### 3.1 USING BOOLEAN EXPRESSIONS 144

Evaluating Boolean Expressions 144

*Pitfall:* Boolean Expressions Convert to *int* Values 148

Enumeration Types (*Optional*) 151

### 3.2 MULTIWAY BRANCHES 152

Nested Statements 152

*Programming Tip:* Use Braces in Nested Statements 153

Multiway *if-else* Statements 155

*Programming Example:* State Income Tax 157

The *switch* Statement 160

*Pitfall:* Forgetting a *break* in a *switch* Statement 164

Using *switch* Statements for Menus 165

Blocks 167

*Pitfall:* Inadvertent Local Variables 170

### 3.3 MORE ABOUT C++ LOOP STATEMENTS 171

The *while* Statements Reviewed 171

Increment and Decrement Operators Revisited 173

The *for* Statement 176

*Pitfall:* Extra Semicolon in a *for* Statement 181

What Kind of Loop to Use 182

*Pitfall:* Uninitialized Variables and Infinite Loops 184

The *break* Statement 185

*Pitfall:* The *break* Statement in Nested Loops 186

### 3.4 DESIGNING LOOPS 187

Loops for Sums and Products 187

Ending a Loop 189

Nested Loops	192
Debugging Loops	194
Chapter Summary	197
Answers to Self-Test Exercises	198
Practice Programs	204
Programming Projects	206

## **Chapter 4** Procedural Abstraction and Functions That Return a Value 213

### **4.1 TOP-DOWN DESIGN 214**

### **4.2 PREDEFINED FUNCTIONS 215**

Using Predefined Functions	215
Random Number Generation	220
Type Casting	222
Older Form of Type Casting	224
<i>Pitfall: Integer Division Drops the Fractional Part</i>	224

### **4.3 PROGRAMMER-DEFINED FUNCTIONS 225**

Function Definitions	225
Functions That Return a Boolean Value	231
Alternate Form for Function Declarations	231
<i>Pitfall: Arguments in the Wrong Order</i>	232
Function Definition–Syntax Summary	233
More About Placement of Function Definitions	234
<i>Programming Tip: Use Function Calls in Branching Statements</i>	235

### **4.4 PROCEDURAL ABSTRACTION 236**

The Black-Box Analogy	236
<i>Programming Tip: Choosing Formal Parameter Names</i>	239
<i>Programming Tip: Nested Loops</i>	240
<i>Case Study: Buying Pizza</i>	243
<i>Programming Tip: Use Pseudocode</i>	249

### **4.5 SCOPE AND LOCAL VARIABLES 250**

The Small Program Analogy	250
<i>Programming Example: Experimental Pea Patch</i>	253
Global Constants and Global Variables	253
Call-by-Value Formal Parameters Are Local Variables	256
Block Scope	258

Namespaces Revisited	259
<i>Programming Example: The Factorial Function</i>	262
<b>4.6 OVERLOADING FUNCTION NAMES</b>	<b>264</b>
Introduction to Overloading	264
<i>Programming Example: Revised Pizza-Buying Program</i>	267
Automatic Type Conversion	270
Chapter Summary	272
Answers to Self-Test Exercises	272
Practice Programs	277
Programming Projects	279

## **Chapter 5** Functions for All Subtasks 283

<b>5.1 VOID FUNCTIONS</b>	<b>284</b>
Definitions of <i>void</i> Functions	284
<i>Programming Example: Converting Temperatures</i>	287
return Statements in <i>void</i> Functions	287
<b>5.2 CALL-BY-REFERENCE PARAMETERS</b>	<b>291</b>
A First View of Call-by-Reference	291
Call-by-Reference in Detail	294
<i>Programming Example: The swapValues Function</i>	299
Mixed Parameter Lists	300
<i>Programming Tip: What Kind of Parameter to Use</i>	301
<i>Pitfall: Inadvertent Local Variables</i>	302
<b>5.3 USING PROCEDURAL ABSTRACTION</b>	<b>305</b>
Functions Calling Functions	305
Preconditions and Postconditions	307
<i>Case Study: Supermarket Pricing</i>	308
<b>5.4 TESTING AND DEBUGGING FUNCTIONS</b>	<b>313</b>
Stubs and Drivers	314
<b>5.5 GENERAL DEBUGGING TECHNIQUES</b>	<b>319</b>
Keep an Open Mind	319
Check Common Errors	319
Localize the Error	320
The <code>assert</code> Macro	322

Chapter Summary	324
Answers to Self-Test Exercises	325
Practice Programs	328
Programming Projects	331

## Chapter 6 I/O Streams as an Introduction to Objects and Classes 339

### 6.1 STREAMS AND BASIC FILE I/O 340

Why Use Files for I/O?	341
File I/O	342
Introduction to Classes and Objects	346
<i>Programming Tip: Check Whether a File Was Opened Successfully</i>	348
Techniques for File I/O	350
Appending to a File ( <i>Optional</i> )	354
File Names as Input ( <i>Optional</i> )	355

### 6.2 TOOLS FOR STREAM I/O 357

Formatting Output with Stream Functions	357
Manipulators	363
Streams as Arguments to Functions	366
<i>Programming Tip: Checking for the End of a File</i>	366
A Note on Namespaces	369
<i>Programming Example: Cleaning Up a File Format</i>	370

### 6.3 CHARACTER I/O 372

The Member Functions <code>get</code> and <code>put</code>	372
The <code>putback</code> Member Function ( <i>Optional</i> )	376
<i>Programming Example: Checking Input</i>	377
<i>Pitfall: Unexpected '\n' in Input</i>	379
<i>Programming Example: Another <code>newLine</code> Function</i>	381
Default Arguments for Functions ( <i>Optional</i> )	382
The <code>eof</code> Member Function	387
<i>Programming Example: Editing a Text File</i>	389
Predefined Character Functions	390
<i>Pitfall: <code>toupper</code> and <code>tolower</code> Return Values</i>	392
Chapter Summary	394
Answers to Self-Test Exercises	395
Practice Programs	402
Programming Projects	404

## Chapter 7 Arrays 411

### 7.1 INTRODUCTION TO ARRAYS 412

Declaring and Referencing Arrays 412

*Programming Tip:* Use *for* Loops with Arrays 414

*Pitfall:* Array Indexes Always Start with Zero 414

*Programming Tip:* Use a Defined *Constant* for the Size of  
an Array 414

Arrays in Memory 416

*Pitfall:* Array Index Out of Range 417

Initializing Arrays 420

*Programming Tip:* C++11 Range-Based *for* Statement 420

### 7.2 ARRAYS IN FUNCTIONS 423

Indexed Variables as Function Arguments 423

Entire Arrays as Function Arguments 425

The *const* Parameter Modifier 428

*Pitfall:* Inconsistent Use of *const* Parameters 431

Functions That Return an Array 431

*Case Study:* Production Graph 432

### 7.3 PROGRAMMING WITH ARRAYS 445

Partially Filled Arrays 445

*Programming Tip:* Do Not Skimp on Formal Parameters 448

*Programming Example:* Searching an Array 448

*Programming Example:* Sorting an Array 451

*Programming Example:* Bubble Sort 455

### 7.4 MULTIDIMENSIONAL ARRAYS 458

Multidimensional Array Basics 459

Multidimensional Array Parameters 459

*Programming Example:* Two-Dimensional Grading  
Program 461

*Pitfall:* Using Commas Between Array Indexes 465

Chapter Summary 466

Answers to Self-Test Exercises 467

Practice Programs 471

Programming Projects 473



## Chapter 8 Strings and Vectors 485

### 8.1 AN ARRAY TYPE FOR STRINGS 487

C-String Values and C-String Variables 487

*Pitfall:* Using = and == with C Strings 490

Other Functions in `<cstring>` 492

*Pitfall:* Copying past the end of a C-string using `strcpy` 495

C-String Input and Output 498

C-String-to-Number Conversions and Robust Input 500

### 8.2 THE STANDARD STRING CLASS 506

Introduction to the Standard Class `string` 506

I/O with the Class `string` 509

*Programming Tip:* More Versions of `getline` 512

*Pitfall:* Mixing `cin >> variable;` and `getline` 513

String Processing with the Class `string` 514

*Programming Example:* Palindrome Testing 518

Converting between `string` Objects and C Strings 521

Converting Between Strings and Numbers 522

### 8.3 VECTORS 523

Vector Basics 523

*Pitfall:* Using Square Brackets Beyond the Vector Size 526

*Programming Tip:* Vector Assignment Is Well Behaved 527

Efficiency Issues 527

Chapter Summary 529

Answers to Self-Test Exercises 529

Practice Programs 531

Programming Projects 532

## Chapter 9 Pointers and Dynamic Arrays 541

### 9.1 POINTERS 542

Pointer Variables 543

Basic Memory Management 550

*Pitfall:* Dangling Pointers 551

Static Variables and Automatic Variables 552

*Programming Tip:* Define Pointer Types 552

## 9.2 DYNAMIC ARRAYS 555

- Array Variables and Pointer Variables 555
- Creating and Using Dynamic Arrays 556
- Pointer Arithmetic (*Optional*) 562
- Multidimensional Dynamic Arrays (*Optional*) 564
- Chapter Summary 566
- Answers to Self-Test Exercises 566
- Practice Programs 567
- Programming Projects 568

## Chapter 10 Defining Classes 575

### 10.1 STRUCTURES 576

- Structures for Diverse Data 576
- Pitfall*: Forgetting a Semicolon in a Structure Definition 581
- Structures as Function Arguments 582
- Programming Tip*: Use Hierarchical Structures 583
- Initializing Structures 585

### 10.2 CLASSES 588

- Defining Classes and Member Functions 588
- Public and Private Members 593
- Programming Tip*: Make All Member Variables Private 601
- Programming Tip*: Define Accessor and Mutator Functions 601
- Programming Tip*: Use the Assignment Operator with Objects 603
- Programming Example*: BankAccount Class—Version 1 604
- Summary of Some Properties of Classes 608
- Constructors for Initialization 610
- Programming Tip*: Always Include a Default Constructor 618
- Pitfall*: Constructors with No Arguments 619
- Member Initializers and Constructor Delegation in C++11 621

### 10.3 ABSTRACT DATA TYPES 622

- Classes to Produce Abstract Data Types 623
- Programming Example*: Alternative Implementation of a Class 627

### 10.4 INTRODUCTION TO INHERITANCE 632

- Derived Classes 633
- Defining Derived Classes 634

Chapter Summary	638
Answers to Self-Test Exercises	639
Practice Programs	645
Programming Projects	646

## **Chapter 11** Friends, Overloaded Operators, and Arrays in Classes 653

### **11.1 FRIEND FUNCTIONS 654**

<i>Programming Example: An Equality Function</i>	654
Friend Functions	658
<i>Programming Tip: Define Both Accessor Functions and Friend Functions</i>	660
<i>Programming Tip: Use Both Member and Nonmember Functions</i>	662
<i>Programming Example: Money Class (Version 1)</i>	662
Implementation of <code>digitToInt</code> ( <i>Optional</i> )	669
<i>Pitfall: Leading Zeros in Number Constants</i>	670
The <i>const</i> Parameter Modifier	672
<i>Pitfall: Inconsistent Use of const</i>	673

### **11.2 OVERLOADING OPERATORS 677**

Overloading Operators	678
Constructors for Automatic Type Conversion	681
Overloading Unary Operators	683
Overloading <code>&gt;&gt;</code> and <code>&lt;&lt;</code>	684

### **11.3 ARRAYS AND CLASSES 694**

Arrays of Classes	694
Arrays as Class Members	698
<i>Programming Example: A Class for a Partially Filled Array</i>	699

### **11.4 CLASSES AND DYNAMIC ARRAYS 701**

<i>Programming Example: A String Variable Class</i>	702
Destructors	705
<i>Pitfall: Pointers as Call-by-Value Parameters</i>	708
Copy Constructors	709
Overloading the Assignment Operator	714
Chapter Summary	717
Answers to Self-Test Exercises	717
Practice Programs	727
Programming Projects	728

## Chapter 12 Separate Compilation and Namespaces 737

### 12.1 SEPARATE COMPILATION 738

ADTs Reviewed 739

*Case Study:* DigitalTime—A Class Compiled Separately 740

Using `#ifndef` 749

*Programming Tip:* Defining Other Libraries 752

### 12.2 NAMESPACES 753

Namespaces and *using* Directives 754

Creating a Namespace 755

Qualifying Names 758

A Subtle Point About Namespaces (*Optional*) 759

Unnamed Namespaces 760

*Programming Tip:* Choosing a Name for a Namespace 765

*Pitfall:* Confusing the Global Namespace and the Unnamed Namespace 766

Chapter Summary 767

Answers to Self-Test Exercises 768

Practice Programs 770

Programming Projects 772

## Chapter 13 Pointers and Linked Lists 773

### 13.1 NODES AND LINKED LISTS 774

Nodes 774

`nullptr` 779

Linked Lists 780

Inserting a Node at the Head of a List 781

*Pitfall:* Losing Nodes 784

Searching a Linked List 785

Pointers as Iterators 789

Inserting and Removing Nodes Inside a List 789

*Pitfall:* Using the Assignment Operator with Dynamic Data Structures 791

Variations on Linked Lists 794

Linked Lists of Classes 796

### 13.2 STACKS AND QUEUES 799

Stacks 799

*Programming Examples:* A Stack Class 800

Queues 805

*Programming Examples:* A Queue Class 806

Chapter Summary	810
Answers to Self-Test Exercises	810
Practice Programs	813
Programming Projects	814

## Chapter 14 Recursion 823

### 14.1 RECURSIVE FUNCTIONS FOR TASKS 825

<i>Case Study:</i> Vertical Numbers	825
A Closer Look at Recursion	831
<i>Pitfall:</i> Infinite Recursion	833
Stacks for Recursion	834
<i>Pitfall:</i> Stack Overflow	836
Recursion Versus Iteration	836

### 14.2 RECURSIVE FUNCTIONS FOR VALUES 838

General Form for a Recursive Function That Returns a Value	838
<i>Programming Example:</i> Another Powers Function	838

### 14.3 THINKING RECURSIVELY 843

Recursive Design Techniques	843
<i>Case Study:</i> Binary Search—An Example of Recursive Thinking	844
<i>Programming Example:</i> A Recursive Member Function	852

Chapter Summary	856
Answers to Self-Test Exercises	856
Practice Programs	861
Programming Projects	861

## Chapter 15 Inheritance 867

### 15.1 INHERITANCE BASICS 868

Derived Classes	871
Constructors in Derived Classes	879
<i>Pitfall:</i> Use of Private Member Variables from the Base Class	882
<i>Pitfall:</i> Private Member Functions Are Effectively Not Inherited	884
The <i>protected</i> Qualifier	884
Redefinition of Member Functions	887
Redefining Versus Overloading	890
Access to a Redefined Base Function	892

### 15.2 INHERITANCE DETAILS 893

Functions That Are Not Inherited	893
----------------------------------	-----

Assignment Operators and Copy Constructors in Derived Classes	894
Destructors in Derived Classes	895
<b>15.3 POLYMORPHISM</b>	<b>896</b>
Late Binding	897
Virtual Functions in C++	898
Virtual Functions and Extended Type Compatibility	903
<i>Pitfall</i> : The Slicing Problem	907
<i>Pitfall</i> : Not Using Virtual Member Functions	908
<i>Pitfall</i> : Attempting to Compile Class Definitions Without Definitions for Every Virtual Member Function	909
<i>Programming Tip</i> : Make Destructors Virtual	909
Chapter Summary	911
Answers to Self-Test Exercises	911
Practice Programs	915
Programming Projects	918
<b>Chapter 16 Exception Handling</b>	<b>927</b>
<b>16.1 EXCEPTION-HANDLING BASICS</b>	<b>929</b>
A Toy Example of Exception Handling	929
Defining Your Own Exception Classes	938
Multiple Throws and Catches	938
<i>Pitfall</i> : Catch the More Specific Exception First	942
<i>Programming Tip</i> : Exception Classes Can Be Trivial	943
Throwing an Exception in a Function	943
Exception Specification	945
<i>Pitfall</i> : Exception Specification in Derived Classes	947
<b>16.2 PROGRAMMING TECHNIQUES FOR EXCEPTION HANDLING</b>	<b>948</b>
When to Throw an Exception	948
<i>Pitfall</i> : Uncaught Exceptions	950
<i>Pitfall</i> : Nested <i>try-catch</i> Blocks	950
<i>Pitfall</i> : Overuse of Exceptions	950
Exception Class Hierarchies	951
Testing for Available Memory	951
Rethrowing an Exception	952
Chapter Summary	952
Answers to Self-Test Exercises	952
Practice Programs	954
Programming Projects	955



## Chapter 17 Templates 959

### 17.1 TEMPLATES FOR ALGORITHM ABSTRACTION 960

Templates for Functions 961

*Pitfall:* Compiler Complications 965

*Programming Example:* A Generic Sorting Function 967

*Programming Tip:* How to Define Templates 971

*Pitfall:* Using a Template with an Inappropriate Type 972

### 17.2 TEMPLATES FOR DATA ABSTRACTION 973

Syntax for Class Templates 973

*Programming Example:* An Array Class 976

Chapter Summary 983

Answers to Self-Test Exercises 983

Practice Programs 987

Programming Projects 987

## Chapter 18 Standard Template Library and C++11 991

### 18.1 ITERATORS 993

using Declarations 993

Iterator Basics 994

*Programming Tip:* Use auto to Simplify Variable Declarations 998

*Pitfall:* Compiler Problems 998

Kinds of Iterators 1000

Constant and Mutable Iterators 1004

Reverse Iterators 1005

Other Kinds of Iterators 1006

### 18.2 CONTAINERS 1007

Sequential Containers 1008

*Pitfall:* Iterators and Removing Elements 1012

*Programming Tip:* Type Definitions in Containers 1013

Container Adapters `stack` and `queue` 1013

Associative Containers `set` and `map` 1017

*Programming Tip:* Use Initialization, Ranged `for`, and `auto` with Containers 1024

Efficiency 1024

### 18.3 GENERIC ALGORITHMS 1025

Running Times and Big-O Notation 1026

Container Access Running Times 1029

Nonmodifying Sequence Algorithms 1031  
Container Modifying Algorithms 1035  
Set Algorithms 1037  
Sorting Algorithms 1038

#### 18.4 C++ IS EVOLVING 1039

std::array 1039  
Regular Expressions 1040  
Threads 1045  
Smart Pointers 1051  
  
Chapter Summary 1057  
Answers to Self-Test Exercises 1058  
Practice Programs 1059  
Programming Projects 1061

#### APPENDICES

- 1 C++ Keywords 1067
- 2 Precedence of Operators 1068
- 3 The ASCII Character Set 1070
- 4 Some Library Functions 1071
- 5 Inline Functions 1078
- 6 Overloading the Array Index Square Brackets 1079
- 7 The this Pointer 1081
- 8 Overloading Operators as Member Operators 1084

CREDITS 1086

INDEX 1089

This page intentionally left blank

# Introduction to Computers and C++ Programming



## 1.1 COMPUTER SYSTEMS 34

Hardware 34

Software 39

High-Level Languages 40

Compilers 41

*History Note* 44

## 1.2 PROGRAMMING AND PROBLEM-SOLVING 44

Algorithms 44

Program Design 47

Object-Oriented Programming 48

The Software Life Cycle 49

## 1.3 INTRODUCTION TO C++ 50

Origins of the C++ Language 50

A Sample C++ Program 51

*Pitfall:* Using the Wrong Slash in \n 55

*Programming Tip:* Input and Output Syntax 55

Layout of a Simple C++ Program 56

*Pitfall:* Putting a Space Before the `include` File  
Name 58

Compiling and Running a C++ Program 58


*Pitfall:* Compiling a C++11 Program 59

*Programming Tip:* Getting Your Program  
to Run 59

## 1.4 TESTING AND DEBUGGING 61

Kinds of Program Errors 62

*Pitfall:* Assuming Your Program Is Correct 63



*The whole of the development and operation of analysis are now capable of being executed by machinery. . . . As soon as an Analytical Engine exists, it will necessarily guide the future course of science.*

CHARLES BABPAGE (1792–1871)

---

## INTRODUCTION

In this chapter we describe the basic components of a computer, as well as the basic technique for designing and writing a program. We then show you a sample C++ program and describe how it works.

### 1.1 COMPUTER SYSTEMS

A set of instructions for a computer to follow is called a program. The collection of programs used by a computer is referred to as the **software** for that computer. The actual physical machines that make up a computer installation are referred to as **hardware**. As we will see, the hardware for a computer is conceptually very simple. However, computers now come with a large array of software to aid in the task of programming. This software includes editors, translators, and managers of various sorts. The resulting environment is a complicated and powerful system. In this book we are concerned almost exclusively with software, but a brief overview of how the hardware is organized will be useful.

#### Hardware

There are three main classes of computers: *PCs*, *workstations*, and *mainframes*. A **PC (personal computer)** is a relatively small computer designed to be used by one person at a time. Most home computers are PCs, but PCs are also widely used in business, industry, and science. A **workstation** is essentially a larger and more powerful PC. You can think of it as an “industrial-strength” PC. A **mainframe** is an even larger computer that typically requires some support staff and generally is shared by more than one user. The distinctions between PCs, workstations, and mainframes are not precise, but the terms are commonly used and do convey some very general information about a computer.

A **network** consists of a number of computers connected so that they may share resources such as printers and may share information. A network might contain a number of workstations and one or more mainframes, as well as shared devices such as printers.

For our purposes in learning programming, it will not matter whether you are working on a PC, a mainframe, or a workstation. The basic configuration of the computer, as we will view it, is the same for all three types of computers.

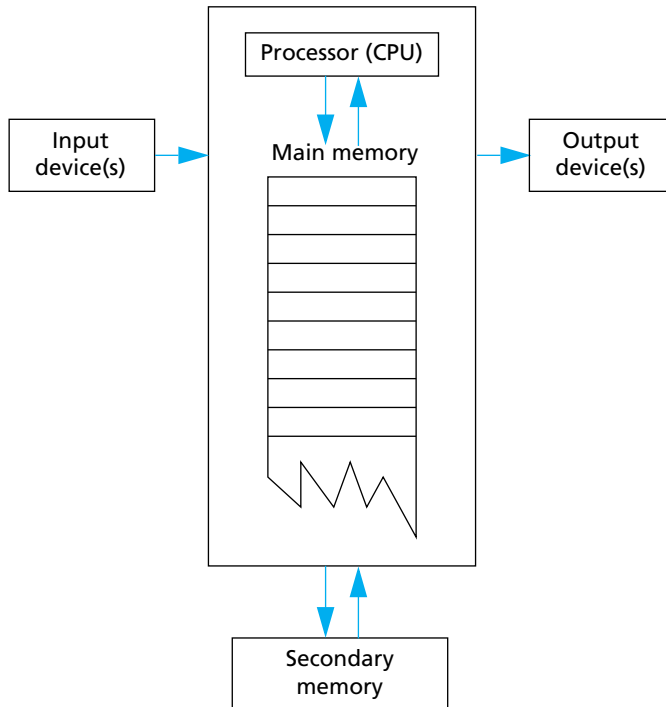
The hardware for most computer systems is organized as shown in Display 1.1. The computer can be thought of as having five main components: the *input device(s)*, the *output device(s)*, the *processor* (also called the *CPU*, for *central processing unit*), the *main memory*, and the *secondary memory*. The processor, main memory, and secondary memory are normally housed in a single cabinet. The processor and main memory form the heart of a computer and can be thought of as an integrated unit. Other components connect to the main memory and operate under the direction of the processor. The arrows in Display 1.1 indicate the direction of information flow.

An **input device** is any device that allows a person to communicate information to the computer. Your primary input devices are likely to be a keyboard and a mouse.

An **output device** is anything that allows the computer to communicate information to you. The most common output device is a display screen, referred to as a *monitor*. Quite often, there is more than one output device. For example, in addition to the monitor, your computer probably is connected to a printer for producing output on paper. The keyboard and monitor are sometimes thought of as a single unit called a *terminal*.

#### DISPLAY 1.1 Main Components of a Computer

---



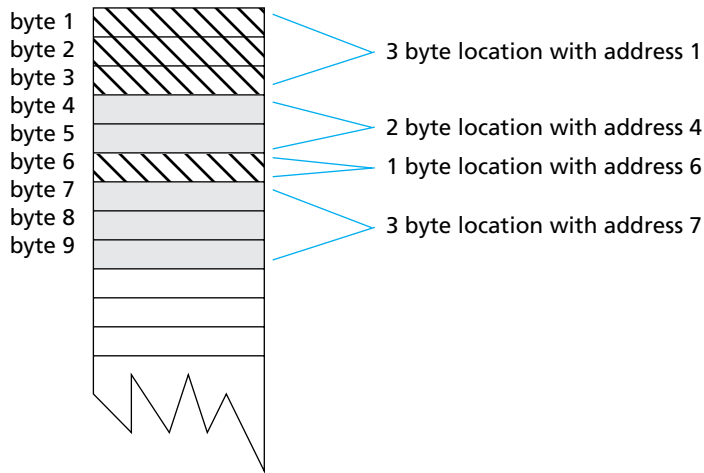
In order to store input and to have the equivalent of scratch paper for performing calculations, computers are provided with *memory*. The program that the computer executes is also stored in this memory. A computer has two forms of memory, called *main memory* and *secondary memory*. The program that is being executed is kept in main memory, and main memory is, as the name implies, the most important memory. **Main memory** consists of a long list of numbered locations called *memory locations*; the number of memory locations varies from one computer to another, ranging from a few thousand to many millions, and sometimes even into the billions. Each memory location contains a string of 0s and 1s. The contents of these locations can change. Hence, you can think of each memory location as a tiny blackboard on which the computer can write and erase. In most computers, all memory locations contain the same number of zero/one digits. A digit that can assume only the values 0 or 1 is called a **binary digit** or a **bit**. The memory locations in most computers contain eight bits (or some multiple of eight bits). An eight-bit portion of memory is called a **byte**, so we can refer to these numbered memory locations as *bytes*. To rephrase the situation, you can think of the computer's main memory as a long list of numbered memory locations called *bytes*. The number that identifies a byte is called its **address**. A data item, such as a number or a letter, can be stored in one of these bytes, and the address of the byte is then used to find the data item when it is needed.

If the computer needs to deal with a data item (such as a large number) that is too large to fit in a single byte, it will use several adjacent bytes to hold the data item. In this case, the entire chunk of memory that holds the data item is still called a **memory location**. The address of the first of the bytes that make up this memory location is used as the address for this larger memory location. Thus, as a practical matter, you can think of the computer's main memory as a long list of memory locations of *varying sizes*. The size of each of these locations is expressed in bytes and the address of the first byte is used as the address (name) of that memory location. Display 1.2 shows a picture of a hypothetical computer's main memory. The sizes of the memory locations are not fixed, but can change when a new program is run on the computer.

### Bytes and Addresses

Main memory is divided into numbered locations called **bytes**. The number associated with a byte is called its **address**. A group of consecutive bytes is used as the location for a data item, such as a number or letter. The address of the first byte in the group is used as the address of this larger memory location.

The fact that the information in a computer's memory is represented as 0s and 1s need not be of great concern to you when programming in C++ (or in

**DISPLAY 1.2** Memory Locations and Bytes

most other programming languages). There is, however, one point about this use of 0s and 1s that will concern us as soon as we start to write programs. The computer needs to interpret these strings of 0s and 1s as numbers, letters, instructions, or other types of information. The computer performs these interpretations automatically according to certain coding schemes. A different code is used for each different type of item that is stored in the computer's memory: one code for letters, another for whole numbers, another for fractions, another for instructions, and so on. For example, in one commonly used set of codes, 01000001 is the code for the letter A and also for the number 65. In order to know what the string 01000001 in a particular location stands for, the computer must keep track of which code is currently being used for that location. Fortunately, the programmer seldom needs to be concerned with such codes and can safely reason as though the locations actually contained letters, numbers, or whatever is desired.

**Why Eight?**

A **byte** is a memory location that can hold eight bits. What is so special about eight? Why not ten bits? There are two reasons why eight is special. First, eight is a power of 2. (8 is  $2^3$ .) Since computers use bits, which have only two possible values, powers of 2 are more convenient than powers of 10. Second, it turns out that eight bits (one byte) are required to code a single character (such as a letter or other keyboard symbol).



The memory we have been discussing up until now is the main memory. Without its main memory, a computer can do nothing. However, main memory is only used while the computer is actually following the instructions in a program. The computer also has another form of memory called *secondary memory* or *secondary storage*. (The words *memory* and *storage* are exact synonyms in this context.) **Secondary memory** is the memory that is used for keeping a permanent record of information after (and before) the computer is used. Some alternative terms that are commonly used to refer to secondary memory are *auxiliary memory*, *auxiliary storage*, *external memory*, and *external storage*.

Information in secondary storage is kept in units called **files**, which can be as large or as small as you like. A program, for example, is stored in a file in secondary storage and copied into main memory when the program is run. You can store a program, a letter, an inventory list, or any other unit of information in a file.

Several different kinds of secondary memory can be attached to a single computer. The most common forms of secondary memory are *hard disks*, *diskettes*, *CDs*, *DVDs*, and *removable flash memory drives*. (**Diskettes** are also sometimes referred to as *floppy disks*.) **CDs** (compact discs) used on computers are basically the same as those used to record and play music, while **DVDs** (digital versatile discs) are the same as those used to play videos. CDs and DVDs for computers can be read-only so that your computer can read, but cannot change, the data on the disc; CDs and DVDs for computers can also be read/write, which can have their data changed by the computer. **Hard disks** are fixed in place and are normally not removed from the disk drive. Diskettes and CDs can be easily removed from the disk drive and carried to another computer. Diskettes and CDs have the advantages of being inexpensive and portable, but hard disks hold more data and operate faster. **Flash drives** have largely replaced diskettes today and store data using a type of memory called flash memory. Unlike main memory, flash memory does not require power to maintain the information stored on the device. Other forms of secondary memory are also available, but this list covers most forms that you are likely to encounter.

Main memory is often referred to as **RAM** or **random access memory**. It is called *random access* because the computer can immediately access the data in any memory location. Secondary memory often requires **sequential access**, which means that the computer must look through all (or at least very many) memory locations until it finds the item it needs.

The **processor** (also known as the **central processing unit**, or **CPU**) is the “brain” of the computer. When a computer is advertised, the computer company tells you what *chip* it contains. The **chip** is the processor. The processor follows the instructions in a program and performs the calculations specified by the program. The processor is, however, a very simple brain. All it can do is follow a set of simple instructions provided by the programmer. Typical processor instructions say things like “Interpret the 0s and 1s as numbers, and then add the number in memory location 37 to the number in memory location 59,

and put the answer in location 43,” or “Read a letter of input, convert it to its code as a string of 0s and 1s, and place it in memory location 1298.” The processor can add, subtract, multiply, and divide and can move things from one memory location to another. It can interpret strings of 0s and 1s as letters and send the letters to an output device. The processor also has some primitive ability to rearrange the order of instructions. Processor instructions vary somewhat from one computer to another. The processor of a modern computer can have as many as several hundred available instructions. However, these instructions are typically all about as simple as those we have just described.

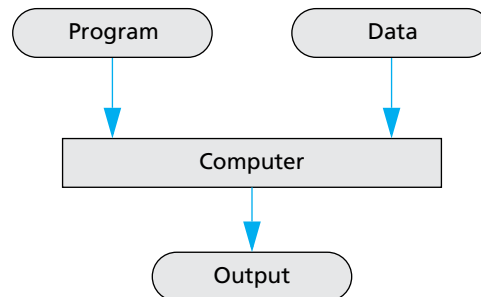
## Software

You do not normally talk directly to the computer, but communicate with it through an *operating system*. The **operating system** allocates the computer’s resources to the different tasks that the computer must accomplish. The operating system is actually a program, but it is perhaps better to think of it as your chief servant. It is in charge of all your other servant programs, and it delivers your requests to them. If you want to run a program, you tell the operating system the name of the file that contains it, and the operating system runs the program. If you want to edit a file, you tell the operating system the name of the file and it starts up the editor to work on that file. To most users, the operating system is the computer. Most users never see the computer without its operating system. The names of some common operating systems are *UNIX*, *DOS*, *Linux*, *Windows*, *Mac OS*, *iOS*, and *Android*.

A **program** is a set of instructions for a computer to follow. As shown in Display 1.3, the input to a computer can be thought of as consisting of two parts, a program and some data. The computer follows the instructions in the program and in that way performs some process. The **data** is what we conceptualize as the input to the program. For example, if the program adds two numbers, then the two numbers are the data. In other words, the data is the input to the program, and both the program and the data are input to the computer (usually via the operating system). Whenever we give a computer

### DISPLAY 1.3 Simple View of Running a Program

---



both a program to follow and some data for the program, we are said to be **running the program** on the data, and the computer is said to **execute the program** on the data. The word *data* also has a much more general meaning than the one we have just given it. In its most general sense, it means any information available to the computer. The word is commonly used in both the narrow sense and the more general sense.

## High-Level Languages

There are many languages for writing programs. In this text we will discuss the C++ programming language and use it to write our programs. C++ is a high-level language, as are most of the other programming languages you are likely to have heard of, such as C, C#, Java, Python, PHP, Pascal, Visual Basic, FORTRAN, COBOL, Lisp, Scheme, and Ada. **High-level languages** resemble human languages in many ways. They are designed to be easy for human beings to write programs in and to be easy for human beings to read. A high-level language, such as C++, contains instructions that are much more complicated than the simple instructions a computer's processor (CPU) is capable of following.

The kind of language a computer can understand is called a **low-level language**. The exact details of low-level languages differ from one kind of computer to another. A typical low-level instruction might be the following:

```
ADD X Y Z
```

This instruction might mean "Add the number in the memory location called X to the number in the memory location called Y, and place the result in the memory location called Z." The above sample instruction is written in what is called **assembly language**. Although assembly language is almost the same as the language understood by the computer, it must undergo one simple translation before the computer can understand it. In order to get a computer to follow an assembly language instruction, the words need to be translated into strings of 0s and 1s. For example, the word ADD might translate to 0110, the X might translate to 1001, the Y to 1010, and the Z to 1011. The version of the instruction above that the computer ultimately follows would then be:

```
0110 1001 1010 1011
```

Assembly language instructions and their translation into 0s and 1s differ from machine to machine.

Programs written in the form of 0s and 1s are said to be written in **machine language**, because that is the version of the program that the computer (the machine) actually reads and follows. Assembly language and machine language are almost the same thing, and the distinction between them will not be important to us. The important distinction is that between

machine language and high-level languages like C++: Any high-level language program must be translated into machine language before the computer can understand and follow the program.

## Compilers

A program that translates a high-level language like C++ to a machine language is called a **compiler**. A compiler is thus a somewhat peculiar sort of program, in that its input or data is some other program, and its output is yet another program. To avoid confusion, the input program is usually called the **source program** or **source code**, and the translated version produced by the compiler is called the **object program** or **object code**. The word **code** is frequently used to mean a program or a part of a program, and this usage is particularly common when referring to object programs. Now, suppose you want to run a C++ program that you have written. In order to get the computer to follow your C++ instructions, proceed as follows. First, run the compiler using your C++ program as data. Notice that in this case, your C++ program is not being treated as a set of instructions. To the compiler, your C++ program is just a long string of characters. The output will be another long string of characters, which is the machine-language equivalent of your C++ program. Next, run this machine-language program on what we normally think of as the data for the C++ program. The output will be what we normally conceptualize as the output of the C++ program. The basic process is easier to visualize if you have two computers available, as diagrammed in Display 1.4. In reality, the entire process is accomplished by using one computer two times.

### Compiler

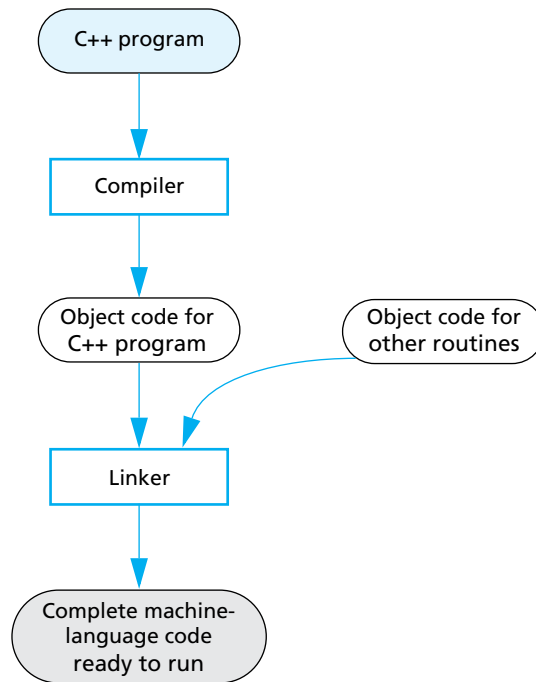
A **compiler** is a program that translates a high-level language program, such as a C++ program, into a machine-language program that the computer can directly understand and execute.

The complete process of translating and running a C++ program is a bit more complicated than what we show in Display 1.4. Any C++ program you write will use some operations (such as input and output routines) that have already been programmed for you. These items that are already programmed for you (like input and output routines) are already compiled and have their object code waiting to be combined with your program's object code to produce a complete machine-language program that can be run on the computer. Another program, called a **linker**, combines the object code for these program pieces with the object code that the compiler produced from your



**DISPLAY 1.5** Preparing a C++ Program for Running

---

**SELF-TEST EXERCISES**

1. What are the five main components of a computer?
2. What would be the data for a program to add two numbers?
3. What would be the data for a program that assigns letter grades to students in a class?
4. What is the difference between a machine-language program and a high-level language program?
5. What is the role of a compiler?
6. What is a source program? What is an object program?
7. What is an operating system?
8. What purpose does the operating system serve?

9. Name the operating system that runs on the computer you use to prepare programs for this course.
10. What is linking?
11. Find out whether linking is done automatically by the compiler you use for this course.

## 1.2 PROGRAMMING AND PROBLEM-SOLVING

*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, *Countess of Lovelace* (1815–1852)

### HISTORY NOTE **Charles Babbage, Ada Augusta**

The first truly programmable computer was designed by **Charles Babbage**, an English mathematician and physical scientist. Babbage began the project sometime before 1822 and worked on it for the rest of his life. Although he never completed the construction of his machine, the design was a conceptual milestone in the history of computing. Much of what we know about Charles Babbage and his computer design comes from the writings of his colleague **Ada Augusta**, the Countess of Lovelace and the daughter of the poet Byron. Ada Augusta is frequently given the title of the first computer programmer. Her comments, quoted in the opening of this section, still apply to the process of solving problems on a computer. Computers are not magic and do not, at least as yet, have the ability to formulate sophisticated solutions to all the problems we encounter. Computers simply do what the programmer orders them to do. The solutions to problems are carried out by the computer, but the solutions are formulated by the programmer. Our discussion of computer programming begins with a discussion of how a programmer formulates these solutions.

In this section we describe some general principles that you can use to design and write programs. These principles are not particular to C++. They apply no matter what programming language you are using.

### Algorithms

When learning your first programming language, it is easy to get the impression that the hard part of solving a problem on a computer is translating your ideas into the specific language that will be fed into the computer. This definitely is not the case. The most difficult part of solving a problem on a computer is discovering the method of solution. After you come up with a method of solution, it is routine to translate your method into the required language, be it C++ or some other programming language. It is therefore helpful to temporarily ignore the programming language and to concentrate instead on formulating the steps of the solution and writing them down in plain English, as if the instructions were to be given to a human being rather than a computer. A sequence of instructions expressed in this way is frequently referred to as an *algorithm*.

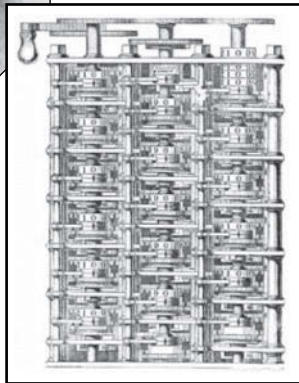
A sequence of precise instructions which leads to a solution is called an **algorithm**. Some approximately equivalent words are *recipe*, *method*,



▲ Ada Augusta,  
Countess of Lovelace and  
the first computer programmer



▲ Charles Babbage



◀ A model of  
Babbage's  
computer

*directions, procedure, and routine.* The instructions may be expressed in a programming language or a human language. Our algorithms will be expressed in English and in the programming language C++. A computer program is simply an algorithm expressed in a language that a computer can understand. Thus, the term *algorithm* is more general than the term *program*. However, when we say that a sequence of instructions is an algorithm, we usually mean that the instructions are expressed in English, since if they were expressed in a programming language we would use the more specific term *program*. An example may help to clarify the concept.

Display 1.6 contains an algorithm expressed in English. The algorithm determines the number of times a specified name occurs on a list of names. If the list contains the winners of each of last season's football games and the name is that of your favorite team, then the algorithm determines how many games your team won. The algorithm is short and simple but is otherwise very typical of the algorithms with which we will be dealing.



---

**DISPLAY 1.6** An Algorithm

---

**Algorithm that determines how many times a name occurs in a list of names:**

1. Get the list of names.
  2. Get the name being checked.
  3. Set a counter to zero.
  4. Do the following for each name on the list:  
Compare the name on the list to the name being checked,  
and if the names are the same, then add one to the counter.
  5. Announce that the answer is the number indicated by the counter.
- 

The instructions numbered 1 through 5 in our sample algorithm are meant to be carried out in the order they are listed. Unless otherwise specified, we will always assume that the instructions of an algorithm are carried out in the order in which they are given (written down). Most interesting algorithms do, however, specify some change of order, usually a repeating of some instruction again and again such as in instruction 4 of our sample algorithm.

The word *algorithm* has a long history. It derives from the name al-Khowarizmi, a ninth-century Persian mathematician and astronomer. He wrote a famous textbook on the manipulation of numbers and equations. The book was entitled *Kitab al-jabr w'almuqabala*, which can be translated as *Rules for Reuniting and Reducing*. The similar-sounding word *algebra* was derived from the Arabic word *al-jabr*, which appears in the title of the book and which is often translated as *reuniting* or *restoring*. The meanings of the words *algebra* and *algorithm* used to be much more intimately related than they are today. Indeed, until modern times, the word *algorithm* usually referred only to algebraic rules for solving numerical equations. Today, the word *algorithm* can be applied to a wide variety of kinds of instructions for manipulating symbolic as well as numeric data. The properties that qualify a set of instructions as an algorithm now are determined by the nature of the instructions rather than by the things manipulated by the instructions. To qualify as an algorithm, a set of instructions must completely and unambiguously specify the steps to be taken and the order in which they are taken. The person or machine carrying out the algorithm does exactly what the algorithm says, neither more nor less.

**Algorithm**

An **algorithm** is a sequence of precise instructions that leads to a solution.

## Program Design

Designing a program is often a difficult task. There is no complete set of rules, no algorithm to tell you how to write programs. Program design is a creative process. Still, there is the outline of a plan to follow. The outline is given in diagrammatic form in Display 1.7. As indicated there, the entire program design process can be divided into two phases, the *problem-solving phase* and the *implementation phase*. The result of the **problem-solving phase** is an algorithm, expressed in English, for solving the problem. To produce a program in a programming language such as C++, the algorithm is translated into the programming language. Producing the final program from the algorithm is called the **implementation phase**.

The first step is to be certain that the task—what you want your program to do—is completely and precisely specified. Do not take this step lightly. If you do not know exactly what you want as the output of your program, you may be surprised at what your program produces. Be certain that you know what the input to the program will be and exactly what information is supposed to be in the output, as well as what form that information should be in. For example, if the program is a bank accounting program, you must know not only the interest rate but also whether interest is to be compounded annually, monthly, daily, or whatever. If the program is supposed to write poetry, you need to determine whether the poems can be in free verse or must be in iambic pentameter or some other meter.

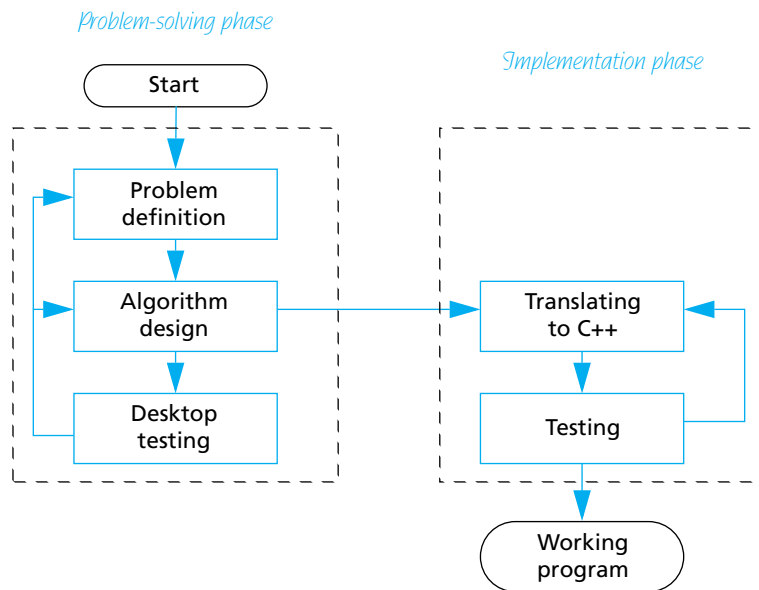
Many novice programmers do not understand the need to design an algorithm before writing a program in a programming language, such as C++, and so they try to short-circuit the process by omitting the problem-solving phase entirely, or by reducing it to just the problem-definition part. This seems reasonable. Why not “go for the mark” and save time? The answer is that *it does not save time!* Experience has shown that the two-phase process will produce a correctly working program faster. The two-phase process simplifies the algorithm design phase by isolating it from the detailed rules of a programming language such as C++. The result is that the algorithm design process becomes much less intricate and much less prone to error. For even a modest-size program, it can represent the difference between a half day of careful work and several frustrating days of looking for mistakes in a poorly understood program.

The implementation phase is not a trivial step. There are details to be concerned about, and occasionally some of these details can be subtle, but it is much simpler than you might at first think. Once you become familiar with C++ or any other programming language, the translation of an algorithm from English into the programming language becomes a routine task.

As indicated in Display 1.7, testing takes place in both phases. Before the program is written, the algorithm is tested, and if the algorithm is found to be deficient, then the algorithm is redesigned. That desktop testing is performed by mentally going through the algorithm and executing the steps yourself. For

**DISPLAY 1.7 Program Design Process**

---



large algorithms this will require a pencil and paper. The C++ program is tested by compiling it and running it on some sample input data. The compiler will give error messages for certain kinds of errors. To find other types of errors, you must somehow check to see whether the output is correct.

The process diagrammed in Display 1.7 is an idealized picture of the program design process. It is the basic picture you should have in mind, but reality is sometimes more complicated. In reality, mistakes and deficiencies are discovered at unexpected times, and you may have to back up and redo an earlier step. For example, as shown in Display 1.7, testing the algorithm might reveal that the definition of the problem was incomplete. In such a case you must back up and reformulate the definition. Occasionally, deficiencies in the definition or algorithm may not be observed until a program is tested. In that case you must back up and modify the problem definition or algorithm and all that follows them in the design process.

## Object-Oriented Programming

The program design process that we outlined in the previous section represents a program as an algorithm (set of instructions) for manipulating some data. That is a correct view, but not always the most productive view. Modern programs are usually designed using a method known as *object-oriented programming*, or **OOP**. In OOP, a program is viewed as a collection of interacting

objects. The methodology is easiest to understand when the program is a simulation program. For example, for a program to simulate a highway interchange, the objects might represent the automobiles and the lanes of the highway. Each object has algorithms that describe how it should behave in different situations. Programming in the OOP style consists of designing the objects and the algorithms they use. When programming in the OOP framework, the term *Algorithm design* in Display 1.7 would be replaced with the phrase *Designing the objects and their algorithms*.

The main characteristics of OOP are *encapsulation*, *inheritance*, and *polymorphism*. Encapsulation is usually described as a form of information hiding or abstraction. That description is correct, but perhaps an easier-to-understand characterization is to say that encapsulation is a form of simplification of the descriptions of objects. Inheritance has to do with writing reusable program 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 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.

## The Software Life Cycle

Designers of large software systems, such as compilers and operating systems, divide the software development process into six phases collectively known as the **software life cycle**. The six phases of this life cycle are:

1. Analysis and specification of the task (problem definition)
2. Design of the software (object and algorithm design)
3. Implementation (coding)
4. Testing
5. Maintenance and evolution of the system
6. Obsolescence

We did not mention the last two phases in our discussion of program design because they take place after the program is finished and put into service. However, they should always be kept in mind. You will not be able to add improvements or corrections to your program unless you design it to be easy to read and easy to change. Designing programs so that they can be easily modified is an important topic that we will discuss in detail when we have developed a bit more background and a few more programming techniques. The meaning of obsolescence is obvious, but it is not always easy to accept. When a program is not working as it should and cannot be fixed with a reasonable amount of effort, it should be discarded and replaced with a completely new program.

## SELF-TEST EXERCISES

12. An algorithm is approximately the same thing as a recipe, but some kinds of steps that would be allowed in a recipe are not allowed in an algorithm. Which steps in the following recipe would be allowed in an algorithm?

Place 2 teaspoons of sugar in mixing bowl.  
Add 1 egg to mixing bowl.  
Add 1 cup of milk to mixing bowl.  
Add 1 ounce of rum, if you are not driving.  
Add vanilla extract to taste.  
Beat until smooth.  
Pour into a pretty glass.  
Sprinkle with nutmeg.

13. What is the first step you should take when creating a program?
14. The program design process can be divided into two main phases. What are they?
15. Explain why the problem-solving phase should not be slighted.

## 1.3 INTRODUCTION TO C++

*Language is the only instrument of science . . .*

SAMUEL JOHNSON (1709–1784)

In this section we introduce you to the C++ programming language, which is the programming language used in this book.

### Origins of the C++ Language

The first thing that people notice about the C++ language is its unusual name. Is there a C programming language, you might ask? Is there a C– or a C – language? Are there programming languages named A and B? The answer to most of these questions is no. But the general thrust of the questions is on the mark. There is a B programming language; it was not derived from a language called A, but from a language called BCPL. The C language was derived from the B language, and C++ was derived from the C language. Why are there two pluses in the name C++? As you will see in the next chapter, ++ is an operation in the C and C++ languages, so using ++ produces a nice pun. The languages BCPL and B do not concern us. They are earlier versions of the C programming language. We will start our description of the C++ programming language with a description of the C language.

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 B, a language developed by Ken Thompson, who is 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 not limited to computers that use UNIX. However, despite its popularity, C is 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 many 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 to do *object-oriented programming*, which is a very powerful programming technique described earlier in this chapter. The C++ language continues to evolve. Major new features were added in 2011. This version is referred to as C++11. Minor features were added in 2014. At the time of writing this edition, C++17 is under development and will include additional features such as parallel algorithms.

## A Sample C++ Program

Display 1.8 contains a simple C++ program and the screen display that might be generated when a *user* runs and interacts with this program. The person who runs a program is called the **user**. The output when the program is run is shown in the Sample Dialogue. The text typed in by the user is shown in color to distinguish it from the text output by the program. On the actual screen both texts would look alike. The source code for the program is shown in lines 1–22. The line numbers are shown only for reference. You would not type in the line numbers when entering the program. Keywords with a predefined meaning in C++ are shown in color. These keywords are discussed in Chapter 2. The person who writes the program is called the **programmer**. Do not confuse the roles of the user and the programmer. The user and the programmer might or might not be the same person. For example, if you write and then run a program, you are both the programmer and the user. With professionally produced programs, the programmer (or programmers) and the user are usually different persons.

**DISPLAY 1.8 A Sample C++ Program**

---

```
1  #include <iostream>
2  using namespace std;
3  int main( )
4  {
5      int numberOfPods, peasPerPod, totalPeas;
6      cout << "Press return after entering a number.\n";
7      cout << "Enter the number of pods:\n";
8      cin >> numberOfPods;
9      cout << "Enter the number of peas in a pod:\n";
10     cin >> peasPerPod;
11     totalPeas = numberOfPods * peasPerPod;
12     cout << "If you have ";
13     cout << numberOfPods;
14     cout << " pea pods\n";
15     cout << "and ";
16     cout << peasPerPod;
17     cout << " peas in each pod, then\n";
18     cout << "you have ";
19     cout << totalPeas;
20     cout << " peas in all the pods.\n";
21     return 0;
22 }
```

---

**Sample Dialogue**

```
Press return after entering a number.
Enter the number of pods:
10
Enter the number of peas in a pod:
9
If you have 10 pea pods
and 9 peas in each pod, then
you have 90 peas in all the pods.
```

---

In the next chapter we will explain in detail all the C++ features you need to write programs like the one in Display 1.8, but to give you a feel for how a C++ program works, we will now provide a brief description of how this particular program works. If some of the details are a bit unclear, do not worry. In this section we just want to give you a feel for what a C++ program is.

The beginning and end of our sample program contain some details that need not concern us yet. The program begins with the following lines:

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

For now we will consider these lines to be a rather complicated way of saying “The program starts here.”

The program ends with the following two lines:

```
    return 0;
}
```

For a simple program, these two lines simply mean “The program ends here.”

The lines in between these beginning and ending lines are the heart of the program. We will briefly describe these lines, starting with the following line:

```
int numberOfPods, peasPerPod, totalPeas;
```

This line is called a **variable declaration**. This variable declaration tells the computer that `numberOfPods`, `peasPerPod`, and `totalPeas` will be used as names for three *variables*. Variables will be explained more precisely in the next chapter, but it is easy to understand how they are used in this program. In this program the **variables** are used to name numbers. The word that starts this line, `int`, is an abbreviation for the word *integer* and it tells the computer that the numbers named by these variables will be integers. An **integer** is a whole number, like 1, 2, -1, -7, 0, 205, -103, and so forth.

The remaining lines are all instructions that tell the computer to do something. These instructions are called **statements** or **executable statements**. In this program each statement fits on exactly one line. That need not be true, but for very simple programs, statements are usually listed one per line.

Most of the statements begin with either the word `cin` or `cout`. These statements are input statements and output statements. The word `cin`, which is pronounced “see-in,” is used for input. The statements that begin with `cin` tell the computer what to do when information is entered from the keyboard. The word `cout`, which is pronounced “see-out,” is used for output, that is, for sending information from the program to the terminal screen. The letter `c` is there because the language is C++. The arrows, written `<<` or `>>`, tell you the direction that data is moving. The arrows, `<<` and `>>`, are called ‘insert’ and ‘extract,’ or ‘put to’ and ‘get from,’ respectively. For example, consider the line:

```
cout << "Press return after entering a number.\n";
```

This line may be read, ‘put “Press...number.\n” to cout’ or simply ‘output “Press...number.\n”’. If you think of the word `cout` as a name for the screen (the output device), then the arrows tell the computer to send the string in quotes to the screen. As shown in the sample dialogue, this causes



the text contained in the quotes to be written to the screen. The `\n` at the end of the quoted string tells the computer to start a new line after writing out the text. Similarly, the next line of the program also begins with `cout`, and that program line causes the following line of text to be written to the screen:

```
Enter the number of pods:
```

The next program line starts with the word `cin`, so it is an input statement. Let's look at that line:

```
cin >> numberOfPods;
```

This line may be read, 'get `numberOfPods` from `cin`' or simply 'input `numberOfPods`'.

If you think of the word `cin` as standing for the keyboard (the input device), then the arrows say that input should be sent from the keyboard to the variable `numberOfPods`. Look again at the sample dialogue. The next line shown has a 10 written in bold. We use bold to indicate something typed in at the keyboard. If you type in the number 10, then the 10 appears on the screen. If you then press the Return key (which is also sometimes called the *Enter key*), that makes the 10 available to the program. The statement which begins with `cin` tells the computer to send that input value of 10 to the variable `numberOfPods`. From that point on, `numberOfPods` has the value 10; when we see `numberOfPods` later in the program, we can think of it as standing for the number 10.

Consider the next two program lines:

```
cout << "Enter the number of peas in a pod:\n";  
cin >> peasPerPod;
```

These lines are similar to the previous two lines. The first sends a message to the screen asking for a number. When you type in a number at the keyboard and press the Return key, that number becomes the value of the variable `peasPerPod`. In the sample dialogue, we assume that you type in the number 9. After you type in 9 and press the Return key, the value of the variable `peasPerPod` becomes 9.

The next nonblank program line, shown below, does all the computation that is done in this simple program:

```
totalPeas = numberOfPods * peasPerPod;
```

The asterisk symbol, `*`, is used for multiplication in C++. So this statement says to multiply `numberOfPods` and `peasPerPod`. In this case, 10 is multiplied by 9 to give a result of 90. The equal sign says that the variable `totalPeas` should be made equal to this result of 90. This is a special use of the equal sign; its meaning here is different than in other mathematical contexts. It gives the variable on the left-hand side a (possibly new) value; in this case it makes 90 the value of `totalPeas`.

The rest of the program is basically more of the same sort of output. Consider the next three nonblank lines:

```
cout << "If you have ";  
cout << numberOfPods;  
cout << " pea Pods\n";
```

These are just three more output statements that work basically the same as the previous statements that begin with `cout`. The only thing that is new is the second of these three statements, which says to output the variable `numberOfPods`. When a variable is output, it is the value of the variable that is output. So this statement causes a 10 to be output. (Remember that in this sample run of the program, the variable `numberOfPods` was set to 10 by the user who ran the program.) Thus, the output produced by these three lines is:

```
If you have 10 pea pods
```

Notice that the output is all on one line. A new line is not begun until the special instruction `\n` is sent as output.

The rest of the program contains nothing new, and if you understand what we have discussed so far, you should be able to understand the rest of the program.

### **PITFALL** Using the Wrong Slash in `\n`

When you use a `\n` in a `cout` statement be sure that you use the **backslash**, which is written `\`. If you make a mistake and use `/n` rather than `\n`, the compiler will not give you an error message. Your program will run, but the output will look peculiar. ■

### ■ **PROGRAMMING TIP** Input and Output Syntax

If you think of `cin` as a name for the keyboard or **input** device and think of `cout` as a name for the screen or the **output** device, then it is easy to remember the direction of the arrows `>>` and `<<`. They point in the direction that data moves. For example, consider the statement:

```
cin >> numberOfPods;
```

In the above statement, data moves from the keyboard to the variable `numberOfPods`, and so the arrow points from `cin` to the variable.

On the other hand, consider the output statement:

```
cout << numberOfPods;
```

In this statement the data moves from the variable `numberOfPods` to the screen, so the arrow points from the variable `numberOfPods` to `cout`. ■

## Layout of a Simple C++ Program

The general form of a simple C++ program is shown in Display 1.9. As far as the compiler is concerned, the **line breaks** and **spacing** need not be as shown there and in our examples. The compiler will accept any reasonable pattern of line breaks and indentation. In fact, the compiler will even accept most unreasonable patterns of line breaks and indentation. However, a program should always be laid out so that it is easy to read. Placing the opening brace, {, on a line by itself and also placing the closing brace, }, on a line by itself will make these punctuations easy to find. Indenting each statement and placing each statement on a separate line makes it easy to see what the program instructions are. Later on, some of our statements will be too long to fit on one line and then we will use a slight variant of this pattern for indenting and line breaks. You should follow the pattern set by the examples in this book, or follow the pattern specified by your instructor if you are in a class.

In Display 1.8, the variable declarations are on the line that begins with the word `int`. As we will see in the next chapter, you need not place all your variable declarations at the beginning of your program, but that is a good default location for them. Unless you have a reason to place them somewhere else, place them at the start of your program as shown in Display 1.9 and in the sample program in Display 1.8. The **statements** are the instructions that are followed by the computer. In Display 1.8, the statements are the lines that begin with `cout` or `cin` and the one line that begins with `totalPeas` followed by an equal sign. Statements are often called **executable statements**. We will use the terms *statement* and *executable statement* interchangeably. Notice that each of the statements we have seen ends with a semicolon. The semicolon in statements is used in more or less the same way that the period is used in English sentences; it marks the end of a statement.

---

### DISPLAY 1.9 Layout of a Simple C++ Program

---

```
1  #include <iostream>
2  using namespace std;
3
4  int  main()
5  {
6      Variable_Declarations
7
8      Statement_1
9      Statement_2
10     ...
11     Statement_Last
12
13     return 0;
14 }
```

---

For now you can view the first few lines as a funny way to say “this is the beginning of the program.” But we can explain them in a bit more detail. The first line

```
#include <iostream>
```

is called an **include directive**. It tells the compiler where to find information about certain items that are used in your program. In this case `iostream` is the name of a library that contains the definitions of the routines that handle input from the keyboard and output to the screen; `iostream` is a file that contains some basic information about this library. The linker program that we discussed earlier in this chapter combines the object code for the library `iostream` and the object code for the program you write. For the library `iostream` this will probably happen automatically on your system. You will eventually use other libraries as well, and when you use them, they will have to be named in directives at the start of your program. For other libraries, you may need to do more than just place an `include` directive in your program, but in order to use any library in your program, you will always need to at least place an `include` directive for that library in your program. Directives always begin with the symbol `#`. Some compilers require that directives have no spaces around the `#`, so it is always safest to place the `#` at the very start of the line and not include any space between the `#` and the word `include`.

The following line further explains the `include` directive that we just explained:

```
using namespace std;
```

This line says that the names defined in `iostream` are to be interpreted in the “standard way” (`std` is an abbreviation of *standard*). We will have more to say about this line a bit later in this book.

The third and fourth nonblank lines, shown next, simply say that the main part of the program starts here:

```
int main()
{
```

The correct term is *main function*, rather than *main part*, but the reason for that subtlety will not concern us until Chapter 4. The braces `{` and `}` mark the beginning and end of the main part of the program. They need not be on a line by themselves, but that is the way to make them easy to find and we will therefore always place each of them on a line by itself.

The next-to-last line

```
return 0;
```

says to “end the program when you get to here.” This line need not be the last thing in the program, but in a very simple program it makes no sense to place it anywhere else. Some compilers will allow you to omit this line and will figure out that the program ends when there are no more statements to execute.

However, other compilers will insist that you include this line, so it is best to get in the habit of including it, even if your compiler is happy without it. This line is called a **return statement** and is considered to be an executable statement because it tells the computer to do something; specifically, it tells the computer to end the program. The number 0 has no intuitive significance to us yet, but must be there; its meaning will become clear as you learn more about C++. Note that even though the return statement says to end the program, you still must add a closing brace, `}`, at the end of the main part of your program.

### **PITFALL** Putting a Space Before the `include` File Name

Be certain that you do not have any extra space between the `<` and the `iostream` file name (Display 1.9) or between the end of the file name and the closing `>`. The compiler `include` directive is not very smart: It will search for a file name that starts or ends with a space! The file name will not be found, producing an error that is quite difficult to locate. You should make this error deliberately in a small program, then compile it. Save the message that your compiler produces so you know what the error message means the next time you get that error message. ■



VideoNote  
Compiling and Running  
a C++ Program

## Compiling and Running a C++ Program

In the previous section you learned what would happen if you ran the C++ program shown in Display 1.8. But where is that program and how do you make it run?

You write a C++ program using a text editor in the same way that you write any other document—a term paper, a love letter, a shopping list, or whatever. The program is kept in a file just like any other document you prepare using a text editor. There are different text editors, and the details of how to use them will vary from one to another, so we cannot say too much more about your text editor. You should consult the documentation for your editor.

The way that you compile and run a C++ program also depends on the particular system you are using, so we will discuss these points in only a very general way. You need to learn how to give the commands to compile, link, and run a C++ program on your system. These commands can be found in the manuals for your system and by asking people who are already using C++ on your system. When you give the command to compile your program, this will produce a machine-language translation of your C++ program. This translated version is called the *object code* for your program. The object code must be linked (that is, combined) with the object code for routines (such as input and output routines) that are already written for you. It is likely that this linking will be done automatically, so you do not need to worry about linking. But on some systems, you may be required to make a separate call to the linker. Again, consult your manuals or a local expert. Finally, you give the command to run your program; how you give that command also depends on the system you are using, so check with the manuals or a local expert.

## **PITFALL** **Compiling a C++11 Program**

C++11 (formerly known as C++0x) is the most recent major version of the standard of the C++ programming language. It was approved on August 12, 2011 by the International Organization for Standardization. C++14 was released on December 15, 2014 and contains small extensions over C++11. We will not discuss these extensions in this book. A C++11 compiler is able to compile and run programs written for older versions of C++. However, the C++11 version includes 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 programs.

You may also need to specify whether or not to compile against the C++11 standard. For example, 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
```

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

## ■ **PROGRAMMING TIP** **Getting Your Program to Run**

Different compilers and different environments might require a slight variation in some details of how you set up a file with your C++ program. Obtain a copy of the program in Display 1.10. It is available for downloading over the Internet. (See the Preface for details.) Alternatively, *very carefully* type in the program yourself. Do not type in the line numbers. Compile the program. If you get an error message, check your typing, fix any typing mistakes, and recompile the file. Once the program compiles with no error messages, try running the program.

If you get the program to compile and run normally, you are all set. You do not need to do anything different from the examples shown in the book. If this program does not compile or does not run normally, then read on. In what follows we offer some hints for dealing with your C++ setup. Once you get this simple program to run normally, you will know what small changes to make to your C++ program files in order to get them to run on your system.

If your program seems to run, but you do not see the output line

```
Testing 1, 2, 3
```

then, in all likelihood, the program probably did give that output, but it disappeared before you could see it. Try adding the following to the end of your program, just before the line `return 0;` these lines should stop your program to allow you to read the output.

**DISPLAY 1.10 Testing Your C++ Setup**

```
1  #include <iostream>
2  using namespace std;
3
4  int main( )
5  {
6      cout << "Testing 1, 2, 3\n";
7      return 0;
8  }
9
```

*If you cannot compile and run this program, then see the programming tip entitled "Getting Your Program to Run." It suggests some things to do to get your C++ programs to run on your particular computer setup.*

**Sample Dialogue**

```
Testing 1, 2, 3
```

```
char letter;
cout << "Enter a letter to end the program:\n";
cin >> letter;
```

The part in braces should then read as follows:

```
cout << "Testing 1, 2, 3\n";
char letter;
cout << "Enter a letter to end the program:\n";
cin >> letter;
return 0;
```

For now you need not understand these added lines, but they will be clear to you by the end of Chapter 2.

If the program does not compile or run at all, then try changing

```
#include <iostream>
```

by adding `.h` to the end of `iostream`, so it reads as follows:

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

If your program requires `iostream.h` instead of `iostream`, then you have an old C++ compiler and should obtain a more recent compiler.

If your program still does not compile and run normally, try deleting

```
using namespace std;
```

If your program still does not compile and run, then check the documentation for your version of C++ to see if any more "directives" are needed for "console" input/output.

If all this fails, consult your instructor if you are in a course. If you are not in a course or you are not using the course computer, check the documentation

for your C++ compiler or check with a friend who has a similar computer setup. The necessary change is undoubtedly very small and, once you find out what it is, very easy.

## SELF-TEST EXERCISES

16. If the following statement were used in a C++ program, what would it cause to be written on the screen?

```
cout << "C++ is easy to understand.";
```

17. What is the meaning of `\n` as used in the following statement (which appears in Display 1.8)?

```
cout << "Enter the number of peas in a pod:\n";
```

18. What is the meaning of the following statement (which appears in Display 1.8)?

```
cin >> peasPerPod;
```

19. What is the meaning of the following statement (which appears in Display 1.8)?

```
totalPeas = numberOfPods * peasPerPod;
```

20. What is the meaning of this directive?

```
#include <iostream>
```

21. What, if anything, is wrong with the following `#include` directives?

- a. `#include <iostream >`
- b. `#include < iostream>`
- c. `#include <iostream>`

## 1.4 TESTING AND DEBUGGING

*"And if you take one from three hundred and sixty-five, what remains?"*

*"Three hundred and sixty-four, of course."*

*Humpty Dumpty looked doubtful. "I'd rather see that done on paper," he said.*

LEWIS CARROLL, *Through the Looking-Glass*

A mistake in a program is usually called a **bug**, and the process of eliminating bugs is called **debugging**. There is colorful history of how this term came into use. It occurred in the early days of computers, when computer hardware was



extremely sensitive and occupied an entire room. Rear Admiral Grace Murray Hopper (1906–1992) was “the third programmer on the world’s first large-scale digital computer.” (Denise W. Gurer, “Pioneering women in computer science” CACM 38(1):45–54, January 1995.) While Hopper was working on the Harvard Mark I computer under the command of Harvard professor Howard H. Aiken, an unfortunate moth caused a relay to fail. Hopper and the other programmers taped the deceased moth in the logbook with the note “First actual case of bug being found.” The logbook is currently on display at the Naval Museum in Dahlgren, Virginia. This was the first documented computer bug. Professor Aiken would come into the facility during a slack time and inquire if any numbers were being computed. The programmers would reply that they were debugging the computer. For more information about Admiral Hopper and other persons in computing, see Robert Slater, *Portraits in Silicon* (MIT Press, 1987). Today, a bug is a mistake in a program. In this section we describe the three main kinds of programming mistakes and give some hints on how to correct them.

## Kinds of Program Errors

The compiler will catch certain kinds of mistakes and will write out an error message when it finds a mistake. It will detect what are called **syntax errors**, because they are, by and large, violation of the syntax (that is, the grammar rules) of the programming language, such as omitting a semicolon.

If the compiler discovers that your program contains a syntax error, it will tell you where the error is likely to be and what kind of error it is likely to be. When the compiler says your program contains a syntax error, you can be confident that it does. However, the compiler may be incorrect about either the location or the nature of the error. It does a better job of determining the location of an error, to within a line or two, than it does of determining the source of the error. This is because the compiler is guessing at what you meant to write down and can easily guess wrong. After all, the compiler cannot read your mind. Error messages subsequent to the first one have a higher likelihood of being incorrect with respect to either the location or the nature of the error. Again, this is because the compiler must guess your meaning. If the compiler’s first guess was incorrect, this will affect its analysis of future mistakes, since the analysis will be based on a false assumption.

If your program contains something that is a direct violation of the syntax rules for your programming language, the compiler will give you an **error message**. However, sometimes the compiler will give you only a **warning message**, which indicates that you have done something that is not, technically speaking, a violation of the programming language syntax rules, but that is unusual enough to indicate a likely mistake. When you get a warning message, the compiler is saying, “Are you sure you mean this?” At this stage of your development, you should treat every warning as if it were an error until your instructor approves ignoring the warning.

There are certain kinds of errors that the computer system can detect only when a program is run. Appropriately enough, these are called **run-time errors**. Most computer systems will detect certain run-time errors and output an appropriate error message. Many run-time errors have to do with numeric calculations. For example, if the computer attempts to divide a number by zero, that is normally a run-time error.

If the compiler approved of your program and the program ran once with no run-time error messages, this does not guarantee that your program is correct. Remember, the compiler will only tell you if you wrote a syntactically (that is, grammatically) correct C++ program. It will not tell you whether the program does what you want it to do. Mistakes in the underlying algorithm or in translating the algorithm into the C++ language are called **logic errors**. For example, if you were to mistakenly use the addition sign + instead of the multiplication sign \* in the program in Display 1.8, that would be a logic error. The program would compile and run normally but would give the wrong answer. If the compiler approves of your program and there are no run-time errors but the program does not perform properly, then undoubtedly your program contains a logic error. Logic errors are the hardest kind to diagnose, because the computer gives you no error messages to help find the error. It cannot reasonably be expected to give any error messages. For all the computer knows, you may have meant what you wrote.

### **PITFALL** Assuming Your Program Is Correct

In order to test a new program for logic errors, you should run the program on several representative data sets and check its performance on those inputs. If the program passes those tests, you can have more confidence in it, but this is still not an absolute guarantee that the program is correct. It still may not do what you want it to do when it is run on some other data. The only way to justify confidence in a program is to program carefully and so avoid most errors. ■

### **SELF-TEST EXERCISES**

22. What are the three main kinds of program errors?
23. What kinds of errors are discovered by the compiler?
24. If you omit a punctuation symbol (such as a semicolon) from a program, an error is produced. What kind of error?
25. Omitting the final brace } from a program produces an error. What kind of error?

26. Suppose your program has a situation about which the compiler reports a warning. What should you do about it? Give the text's answer and your local answer if it is different from the text's. Identify your answers as the text's or as based on your local rules.
27. Suppose you write a program that is supposed to compute the interest on a bank account at a bank that computes interest on a daily basis, and suppose you incorrectly write your program so that it computes interest on an annual basis. What kind of program error is this?

## CHAPTER SUMMARY

The collection of programs used by a computer is referred to as the **software** for that computer. The actual physical machines that make up a computer installation are referred to as **hardware**.

- The five main components of a computer are the input device(s), the output device(s), the processor (CPU), the main memory, and the secondary memory.
- A computer has two kinds of memory: main memory and secondary memory. Main memory is used only while the program is running. Secondary memory is used to hold data that will stay in the computer before and/or after the program is run.
- A computer's main memory is divided into a series of numbered locations called **bytes**. The number associated with one of these bytes is called the **address** of the byte. Often, several of these bytes are grouped together to form a larger memory location. In that case, the address of the first byte is used as the address of this larger memory location.
- A **byte** consists of eight binary digits, each either zero or one. A digit that can only be zero or one is called a **bit**.
- A **compiler** is a program that translates a program written in a high-level language like C++ into a program written in the machine language that the computer can directly understand and execute.
- A sequence of precise instructions that leads to a solution is called an **algorithm**. Algorithms can be written in English or in a programming language, like C++. However, the word *algorithm* is usually used to mean a sequence of instructions written in English (or some other human language, such as Spanish or Arabic).
- Before writing a C++ program, you should design the algorithm (method of solution) that the program will use.
- Programming errors can be classified into three groups: syntax errors, run-time errors, and logic errors. The computer will usually tell you about errors in the first two categories. You must discover logic errors yourself.

- The individual instructions in a C++ program are called **statements**.
- A variable in a C++ program can be used to name a number. (Variables are explained more fully in the next chapter.)
- A statement in a C++ program that begins with `cout <<` is an output statement, which tells the computer to output to the screen whatever follows the `<<`.
- A statement in a C++ program that begins with `cin >>` is an input statement.

### Answers to Self-Test Exercises

1. The five main components of a computer are the input device(s), the output device(s), the processor (CPU), the main memory, and the secondary memory.
2. The two numbers to be added.
3. The grades for each student on each test and each assignment.
4. A machine-language program is a low-level language consisting of 0s and 1s that the computer can directly execute. A high-level language is written in a more English-like format and is translated by a compiler into a machine-language program that the computer can directly understand and execute.
5. A compiler translates a high-level language program into a machine-language program.
6. The high-level language program that is input to a compiler is called the source program. The translated machine-language program that is output by the compiler is called the object program.
7. An operating system is a program, or several cooperating programs, but is best thought of as the user's chief servant.
8. An operating system's purpose is to allocate the computer's resources to different tasks the computer must accomplish.
9. Among the possibilities are the Macintosh operating system Mac OS, Windows, VMS, Solaris, SunOS, UNIX (or perhaps one of the UNIX-like operating systems such as Linux). There are many others.
10. The object code for your C++ program must be combined with the object code for routines (such as input and output routines) that your program uses. This process of combining object code is called linking. For simple programs, this linking may be done for you automatically.

11. The answer varies, depending on the compiler you use. Most UNIX and UNIX-like compilers link automatically, as do the compilers in most integrated development environments for Windows and Macintosh operating systems.
12. The following instructions are too vague for use in an algorithm:  

Add vanilla extract to taste.  
Beat until smooth.  
Pour into a pretty glass.  
Sprinkle with nutmeg.

The notions of “to taste,” “smooth,” and “pretty” are not precise. The instruction “sprinkle” is too vague, since it does not specify how much nutmeg to sprinkle. The other instructions are reasonable to use in an algorithm.
13. The first step you should take when creating a program is to be certain that the task to be accomplished by the program is completely and precisely specified.
14. The problem-solving phase and the implementation phase.
15. Experience has shown that the two-phase process produces a correctly working program faster.
16. C++ is easy to understand.
17. The symbols `\n` tell the computer to start a new line in the output so that the next item output will be on the next line.
18. This statement tells the computer to read the next number that is typed in at the keyboard and to send that number to the variable named `peasPerPod`.
19. This statement says to multiply the two numbers in the variables `numberOfPods` and `peasPerPod`, and to place the result in the variable named `totalPeas`.
20. The `#include <iostream>` directive tells the compiler to fetch the file `iostream`. This file contains declarations of `cin`, `cout`, the insertion (`<<`) and extraction (`>>`) operators for I/O (input and output). This enables correct linking of the object code from the `iostream` library with the I/O statements in the program.
21.
  - a. The extra space after the `iostream` file name causes a *file-not-found* error message.
  - b. The extra space before the `iostream` file name causes a *file-not-found* error message.
  - c. This one is correct.

22. The three main kinds of program errors are syntax errors, run-time errors, and logic errors.
23. The compiler detects syntax errors. There are other errors that are not technically syntax errors that we are lumping with syntax errors. You will learn about these later.
24. A syntax error.
25. A syntax error.
26. The text states that you should take warnings as if they had been reported as errors. You should ask your instructor for the local rules on how to handle warnings.
27. A logic error.

## PRACTICE PROGRAMS

*Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.*

1. Using your text editor, enter (that is, type in) the C++ program shown in Display 1.8. Be certain to type the first line exactly as shown in Display 1.8. In particular, be sure that the first line begins at the left-hand end of the line with no space before or after the # symbol. Compile and run the program. If the compiler gives you an error message, correct the program and recompile the program. Do this until the compiler gives no error messages. Then run your program.
2. Modify the C++ program you entered in Practice Program 1. Change the program so that it asks a user for their favourite number, writes it to the screen and then goes on to do the same things that the program in Display 1.8 does. Create an `int` variable to store the number and read a value from the user using `cin`. Print out the text "Your favourite number is:" and the number that the user has entered. Be certain to add the symbols `\n` to your output statement. If the user entered 42, you should print out "Your favourite number is 42". Recompile the changed program and run it.
3. Further modify the C++ program that you already modified in Practice Program 2. Ask the user for two numbers. Store the two numbers the user entered in two `int` variables. Print out the sum of these variables. For example, if the user entered 3 and 5, print out, "The sum of 3 and 5 is 8".

4. Modify the C++ program you wrote in Practice Problem 3. Change the addition sign `+` in your C++ program to the subtraction sign `-`. What happens if the user enters a negative number like `-2` as the second input? What happens if the user enters an extremely big number, such as `9,876,543,210`, and subtracts another number from it?
5. Modify the C++ program you wrote in Practice Problem 3. Change the addition sign `+` in your C++ program to the division sign `/`. What happens if the second number is larger than the first number? What happens when you enter a `0` as the second number?
6. The purpose of this exercise is to produce a catalog of typical syntax errors and error messages that will be encountered by a beginner and to continue acquainting you with the programming environment. This exercise should leave you with a knowledge of what error to look for when given any of a number of common error messages.



VideoNote  
Solution to Practice  
Program 1.6

Your instructor may have a program for you to use for this exercise. If not, you should use a program from one of the previous Practice Programs.

Deliberately introduce errors to the program, compile, record the error and the error message, fix the error, compile again (to be sure you have the program corrected), then introduce another error. Keep the catalog of errors and add program errors and messages to it as you continue through this course.

The sequence of suggested errors to introduce is:

- a. Put an extra space between the `<` and the `iostream` file name.
- b. Omit one of the `<` or `>` symbols in the include directive.
- c. Omit the `int` from `int main()`.
- d. Omit or misspell the word `main`.
- e. Omit one of the `()`; then omit both the `()`.
- f. Continue in this fashion, deliberately misspelling identifiers (`cout`, `cin`, and so on). Omit one or both of the `<<` in the `cout` statement; leave off the ending curly brace `}`.

## PROGRAMMING PROJECTS

*Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit [www.myprogramminglab.com](http://www.myprogramminglab.com) to complete many of these Programming Projects online and get instant feedback.*

1. Write a C++ program that reads in two integers and then outputs both their sum and their product. One way to proceed is to start with the

program in Display 1.8 and to then modify that program to produce the program for this project. Be certain to type the first line of your program exactly the same as the first line in Display 1.8. In particular, be sure that the first line begins at the left-hand end of the line with no space before or after the # symbol. Also, be certain to add the symbols `\n` to the last output statement in your program. For example, the last output statement might be the following:

```
cout << "This is the end of the program.\n";
```

(Some systems require that final `\n`, and your system may be one of these.)

- Write a program that prints out "C S !" in large block letters inside a border of \*s followed by two blank lines then the message Computer Science is Cool Stuff. The output should look as follows:

```
*****
      C C C      S S S S      !!
    C      C      S      S      !!
  C          S          !!
C          S          !!
C          S S S S      !!
C          S          !!
  C          S      S      !!
    C      C      S      S      !!
      C C C      S S S S      00
*****

Computer Science is Cool Stuff!!!
```

- Write a program that allows the user to enter a number of quarters, dimes, and nickels and then outputs the monetary value of the coins in cents. For example, if the user enters 2 for the number of quarters, 3 for the number of dimes, and 1 for the number of nickels, then the program should output that the coins are worth 85 cents.
- Write a program that allows the user to enter a time in seconds and then outputs how far an object would drop if it is in freefall for that length of time. Assume that the object starts at rest, there is no friction or resistance from air, and there is a constant acceleration of 32 feet per second due to gravity. Use the equation:

$$\text{distance} = \frac{\text{acceleration} \times \text{time}^2}{2}$$

You should first compute the product and then divide the result by 2. (The reason for this will be discussed later in the book.)



VideoNote  
Solution to Programming  
Project 1.3



5. Write a program that inputs a character from the keyboard and then outputs a large block letter "C" composed of that character. For example, if the user inputs the character "X," then the output should look as follows:

```
      X X X
    X   X
  X
  X
  X
  X
  X
  X
    X   X
    X X X
```



# C++ Basics 2

## 2.1 VARIABLES AND ASSIGNMENTS 72

Variables 72  
Names: Identifiers 74  
Variable Declarations 76  
Assignment Statements 78  
*Pitfall:* Uninitialized Variables 80  
*Programming Tip:* Use Meaningful Names 81

## 2.2 INPUT AND OUTPUT 82

Output Using `cout` 82  
Include Directives and Namespaces 84  
Escape Sequences 85  
*Programming Tip:* End Each Program with  
a `\n` or `endl` 87  
Formatting for Numbers with a Decimal Point 87  
Input Using `cin` 88  
Designing Input and Output 90  
*Programming Tip:* Line Breaks in I/O 90

## 2.3 DATA TYPES AND EXPRESSIONS 92

The Types `int` and `double` 92  
Other Number Types 94  
C++11 Types 95


The Type `char` 96  
The Type `bool` 98  
Introduction to the Class `string` 98  
Type Compatibilities 100  
Arithmetic Operators and Expressions 101  
*Pitfall:* Whole Numbers in Division 104  
More Assignment Statements 106

## 2.4 SIMPLE FLOW OF CONTROL 106

A Simple Branching Mechanism 107  
*Pitfall:* Strings of Inequalities 112  
*Pitfall:* Using `=` in place of `==` 113  
Compound Statements 114  
Simple Loop Mechanisms 116  
Increment and Decrement Operators 119  
*Programming Example:* Charge Card Balance 121  
*Pitfall:* Infinite Loops 122

## 2.5 PROGRAM STYLE 125

Indenting 125  
Comments 125  
Naming Constants 127



*Don't imagine you know what a computer terminal is. A computer terminal is not some clunky old television with a typewriter in front of it. It is an interface where the mind and the body can connect with the universe and move bits of it about.*

DOUGLAS ADAMS, *Mostly Harmless* (the fifth volume in *The Hitchhiker's Trilogy*)

---

## INTRODUCTION

In this chapter we explain some additional sample C++ programs and present enough details of the C++ language to allow you to write simple C++ programs.

## PREREQUISITES

In Chapter 1 we gave a brief description of one sample C++ program. (If you have not read the description of that program, you may find it helpful to do so before reading this chapter.)

## 2.1 VARIABLES AND ASSIGNMENTS

*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*

Programs manipulate data such as numbers and letters. C++ and most other common programming languages use programming constructs known as *variables* to name and store data. Variables are at the very heart of a programming language like C++, so that is where we start our description of C++. We will use the program in Display 2.1 for our discussion and will explain all the items in that program. While the general idea of that program should be clear, some of the details are new and will require some explanation.

### Variables

A C++ variable can hold a number or data of other types. For the moment, we will confine our attention to variables that hold only numbers. These variables are like small blackboards on which the numbers can be written. Just as the numbers written on a blackboard can be changed, so too can the number held by a C++ variable be changed. Unlike a blackboard that might possibly contain no number at all, a C++ variable is guaranteed to have some value in it, if only a garbage number left in the computer's memory by some previously run

program. The number or other type of data held in a variable is called its **value**; that is, the value of a variable is the item written on the figurative blackboard. In the program in Display 2.1, `numberOfBars`, `oneWeight`, and `totalWeight` are variables. For example, when this program is run with the input shown in the sample dialogue, `numberOfBars` has its value set equal to the number 11 with the statement

```
cin >> numberOfBars;
```

Later, the value of the variable `numberOfBars` is changed to 12 when a second copy of the same statement is executed. We will discuss exactly how this happens a little later in this chapter.

Of course, variables are not blackboards. In programming languages, variables are implemented as memory locations. The compiler assigns a memory location (of the kind discussed in Chapter 1) to each variable name in the program. The value of the variable, in a coded form consisting of 0s and 1s, is kept in the memory location assigned to that variable. For example, the three variables in the program shown in Display 2.1 might be assigned the memory locations with addresses 1001, 1003, and 1007. The exact numbers will depend on your computer, your compiler, and a number of other factors. We do not know, or even care, what addresses the compiler will choose for the variables in our program. We can think as though the memory locations were actually labeled with the variable names.

---

#### DISPLAY 2.1 A C++ Program (*part 1 of 2*)

```

1  #include <iostream>
2  using namespace std;
3  int main( )
4  {
5      int numberOfBars;
6      double oneWeight, totalWeight;
7
8      cout << "Enter the number of candy bars in a package\n";
9      cout << "and the weight in ounces of one candy bar.\n";
10     cout << "Then press return.\n";
11     cin >> numberOfBars;
12     cin >> oneWeight;
13
14     totalWeight = oneWeight * numberOfBars;
15
16     cout << numberOfBars << " candy bars\n";
17     cout << oneWeight << " ounces each\n";
18     cout << "Total weight is " << totalWeight << " ounces.\n";
19
20     cout << "Try another brand.\n";
21     cout << "Enter the number of candy bars in a package\n";
22     cout << "and the weight in ounces of one candy bar.\n";

```

(continued)

**DISPLAY 2.1 A C++ Program (part 2 of 2)**

---

```
23     cout << "Then press return.\n";
24     cin >> numberOfBars;
25     cin >> oneWeight;
26
27     totalWeight = oneWeight * numberOfBars;
28
29     cout << numberOfBars << " candy bars\n";
30     cout << oneWeight << " ounces each\n";
31     cout << "Total weight is " << totalWeight << " ounces.\n";
32
33     cout << "Perhaps an apple would be healthier.\n";
34
35     return 0;
36 }
```

---

**Sample Dialogue**

Enter the number of candy bars in a package and the weight in ounces of one candy bar.

Then press return.

11 2.1

11 candy bars

2.1 ounces each

Total weight is 23.1 ounces.

Try another brand.

Enter the number of candy bars in a package and the weight in ounces of one candy bar.

Then press return.

12 1.8

12 candy bars

1.8 ounces each

Total weight is 21.6 ounces.

Perhaps an apple would be healthier.

---

**Names: Identifiers**

The first thing you might notice about the names of the variables in our sample programs is that they are longer than the names normally used for variables in mathematics classes. To make your program easy to understand, you should always use meaningful names for variables. The name of a variable (or other item you might define in a program) is called an **identifier**.

### Cannot Get Programs to Run?

If you cannot get your C++ programs to compile and run, read the Programming Tip in Chapter 1 entitled “Getting Your Program to Run.” That section has tips for dealing with variations in C++ compilers and C++ environments.

An 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 Big_Bonus
```

All of the previously mentioned names are legal and would be accepted by the compiler, but the first five are poor choices for identifiers, since 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.

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 often spelled with all lowercase letters. The predefined identifiers, such as `main`, `cin`, `cout`, and so forth, must be spelled in all lowercase letters. We will see uses for identifiers spelled with uppercase letters later in this chapter.

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

### Identifiers

**Identifiers** are used as names for variables and other items in a C++ program. An identifier must start with either a letter or the underscore symbol, and the remaining characters must all be letters, digits, or the underscore symbol.

There is a special class of identifiers, called **keywords** or **reserved words**, that have a predefined meaning in C++ and that you cannot use as names for variables or anything else. In this book, keywords are written in a different type font like so: *int*, *double*. (And now you know why those words were written in a funny way.) A complete list of keywords is given in Appendix 1.

You may wonder why the other words that we defined as part of the C++ language are not on the list of keywords. What about words like *cin* and *cout*? The answer is that you are allowed to redefine these words, although it would be confusing to do so. These predefined words are not keywords; however, they are defined in libraries required by the C++ language standard. We will discuss libraries later in this book. For now, you need not worry about libraries. 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.

## Variable Declarations

Every variable in a C++ program must be *declared*. 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 two declarations from the program in Display 2.1 declare the three variables used in that program:

```
int numberOfBars;  
double oneWeight, totalWeight;
```

When there is more than one variable in a declaration, the variables are separated by commas. Also, note that each declaration ends with a semicolon.

The word *int* in the first of these two declarations is an abbreviation of the word *integer*. (But in a C++ program you must use the abbreviated form *int*. Do not write out the entire word *integer*.) This line declares the identifier *numberOfBars* to be a variable of *type int*. This means that the value of *numberOfBars* must be a whole number, such as 1, 2, -1, 0, 37, or -288.

The word *double* in the second of these two lines declares the two identifiers *oneWeight* and *totalWeight* to be variables of *type double*. A variable of *type double* can hold numbers with a fractional part, such as 1.75 or -0.55. The kind of data that is held in a variable is called its **type** and the name for the type, such as *int* or *double*, is called a **type name**.

Every variable in a C++ program must be declared before the variable can be used. There are two natural places to declare a variable: either just before it is used or at the start of the main part of your program right after the lines

```
int main()  
{
```

Do whatever makes your program clearer.

### Variable Declarations

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

#### SYNTAX

```
Type_Name variableName1, variableName2, ...;
```

#### EXAMPLES

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

### Naming Conventions

A naming convention is the set of rules that are used to name identifiers such as variables. In this book, we use the convention that variables start with a lowercase letter. Compound words or phrases are squished together with the first letter of the next word in **uppercase**. This convention is called **camelback** or **camelcase**. Other naming conventions, such as the C-style convention, use an underscore between words, while other naming conventions identify the variable type in the variable name.

Variable declarations provide information the compiler needs in order to implement the variables. Recall that the compiler implements variables as memory locations and that the value of a variable is stored in the memory location assigned to that variable. The value is coded as a string of 0s and 1s. Different types of variables require different sizes of memory locations and different methods for coding their values as a string of 0s and 1s. The computer uses one code to encode integers as a string of 0s and 1s. It uses a different code to encode numbers that have a fractional part. It uses yet another code to encode letters as strings of 0s and 1s. The variable declaration tells the compiler—and, ultimately, the computer—what size memory location to use for the variable and which code to use when representing the variable's value as a string of 0s and 1s.



### Syntax and Semantics

The **syntax** for a programming language (or any other kind of language) is the set of grammar rules for that language. For example, when we talk about the syntax for a variable declaration (as in the box labeled “Variable Declarations”), we are talking about the rules for writing down a well-formed variable declaration. If you follow all the syntax rules for C++, then the compiler will accept your program. Of course, this only guarantees that what you write is legal. It guarantees that your program will do something, but it does not guarantee that your program will do what you want it to do.

The **semantics** for a programming language is the meaning of what the program will do when it is run. As a programmer, you have to learn both the correct syntax and the semantics of a programming language.

### Assignment Statements

The most direct way to change the value of a variable is to use an *assignment statement*. An **assignment statement** is an order to the computer saying, “set the value of this variable to what I have written down.” The following line from the program in Display 2.1 is an example of an assignment statement:

```
totalWeight = oneWeight * numberOfBars;
```

This assignment statement tells the computer to set the value of `totalWeight` equal to the number in the variable `oneWeight` multiplied by the number in `numberOfBars`. (As we noted in Chapter 1, `*` is the sign used for multiplication in C++.)

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, and arithmetic operators such as `*` and `+`. 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. A few more examples may help to clarify the way these assignment statements work.

You may use any arithmetic operator in place of the multiplication sign. The following, for example, is also a valid assignment statement:

```
totalWeight = oneWeight + numberOfBars;
```

This statement is just like the assignment statements in our sample program, except that it performs addition rather than multiplication. This statement changes the value of `totalWeight` to the sum of the values of `oneWeight` and `numberOfBars`. Of course, if you made this change in the program in Display 2.1, the program would give incorrect output, but it would still run.

In an assignment statement, the expression on the right-hand side of the equal sign can simply be another variable. The statement

```
totalWeight = oneWeight;
```

changes the value of the variable `totalWeight` so that it is the same as that of the variable `oneWeight`. If you were to use this in the program in Display 2.1, it would give out incorrectly low values for the total weight of a package (assuming there is more than one candy bar in a package), but it might make sense in some other program.

As another example, the following assignment statement changes the value of `numberOfBars` to 37:

```
numberOfBars = 37;
```

A number, like the 37 in this example, is called a **constant**, because unlike a variable, its value cannot change.

Since variables can change value over time and since the assignment operator is one vehicle for changing their values, there is an element of time involved in the meaning of an assignment statement. First, the expression on the right-hand side of the equal sign is evaluated. After that, the value of the variable on the left side of the equal sign is changed to the value that was obtained from that expression. This means that a variable can meaningfully occur on both sides of an assignment operator. For example, consider the assignment statement

```
numberOfBars = numberOfBars + 3;
```

This assignment statement may look strange at first. If you read it as an English sentence, it seems to say “the `numberOfBars` is equal to the `numberOfBars` plus three.” It may seem to say that, but what it really says is “Make the *new* value of `numberOfBars` equal to the *old* value of `numberOfBars` plus three.” The equal sign in C++ is not used the same way that it is used in English or in simple mathematics.

### 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;
```

## 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, 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 0s and 1s was left in its memory location by the last program that used that portion of memory. Thus if the program is run twice, an uninitialized variable may receive a different value each time the program is run. Whenever a program gives different output on *exactly* the same input data and without *any* changes in the program itself, you should suspect an 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. However, a simple constant is what is most often used. You can initialize some, all, or none of the variables in a declaration that lists more than one variable. For example, the following declares three variables and initializes two of them:

```
double rate = 0.07, time, balance = 0.0;
```

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(0.0);
```

Whether you initialize a variable when it is declared or at some later point in the program depends on the circumstances. Do whatever makes your program the easiest to understand.

### 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 variableName1 = Expression_for_Value_1,
      variableName2 = Expression_for_Value_2, . . . ;
```

#### EXAMPLES

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

#### ALTERNATIVE SYNTAX FOR INITIALIZING IN DECLARATIONS

```
Type_Name variableName1 (Expression_for_Value_1),
      variableName2 (Expression_for_Value_2), . . . ;
```

#### EXAMPLES

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

## PROGRAMMING 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 the following:

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

with the more suggestive:

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

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

## 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. Use 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.
3. Give a C++ statement that will change the value of the variable *sum* to the sum of the values in the variables *n1* and *n2*. The variables are all of type *int*.

4. Give a C++ statement that will increase the value of the variable `length` by 8.3. The variable `length` is of type *double*.
5. Give a C++ statement that will change the value of the variable `product` to its old value multiplied by the value of the variable `n`. The variables are all of type *int*.
6. Write a program that contains statements that output the value of five or six variables that have been declared, but not initialized. Compile and run the program. What is the output? Explain.
7. Give good variable names for each of the following:
  - a. A variable to hold the speed of an automobile
  - b. A variable to hold the pay rate for an hourly employee
  - c. A variable to hold the highest score in an exam

## 2.2 INPUT AND OUTPUT

*Garbage in means garbage out.*

### PROGRAMMERS' SAYING

There are several different ways that a C++ program can perform input and output. We will describe what are called *streams*. An **input stream** is simply the stream of input that is being fed into the computer for the program to use. The word *stream* suggests that the program processes the input in the same way no matter where the input comes from. The intuition for the word *stream* is that the program sees only the stream of input and not the source of the stream, like a mountain stream whose water flows past you but whose source is unknown to you. In this section we will assume that the input comes from the keyboard. In Chapter 6 we will discuss how a program can read its input from a file; as you will see there, you can use the same kinds of input statements to read input from a file as those that you use for reading input from the keyboard. Similarly, an **output stream** is the stream of output generated by the program. In this section we will assume the output is going to a terminal screen; in Chapter 6 we will discuss output that goes to a file.

### Output Using `cout`

The values of variables as well as strings of text may be output to the screen using `cout`. There may be any combination of variables and strings to be output. For example, consider the following line from the program in Display 2.1:

```
cout << numberOfBars << " candy bars\n";
```

This statement tells the computer to output two items: the value of the variable `numberOfBars` and the quoted string `" candy bars\n"`. Notice that you do not need a separate copy of the word `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 above single cout statement is equivalent to the following two cout statements:

```
cout << numberOfBars;  
cout << " candy bars\n";
```

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);
```

The parentheses around arithmetic expressions, like price + tax, are required by some compilers, so it is best to include them.

The symbol < is the same as the “less than” symbol. 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.

Whenever you have two cout statements in a row, you can combine them into a single long cout statement. For example, consider the following lines from Display 2.1:

```
cout << numberOfBars << " candy bars\n";  
cout << oneWeight << " ounces each\n";
```

These two statements can be rewritten as the single following statement, and the program will perform exactly the same:

```
cout << numberOfBars << " candy bars\n" << oneWeight  
    << " ounces each\n";
```

If you want to keep your program lines from running off the screen, you will have to place such a long cout statement on two or more lines. A better way to write the previous long cout statement is

```
cout << numberOfBars << " candy bars\n"  
    << oneWeight << " ounces each\n";
```

You should not break a quoted string across two lines, but otherwise you can start a new line anywhere you can insert a space. Any reasonable pattern of spaces and line breaks will be acceptable to the computer, but the previous example and the sample programs are good models to follow. A good policy is to use one cout for each group of output that is intuitively considered a unit. Notice that there is just one semicolon for each cout, even if the cout statement spans several lines.

Pay particular attention to the quoted strings that are output in the program in Display 2.1. Notice that the strings must be included in double quotes. The double quote symbol used is a single key on your keyboard; do not type two single quotes. Also, notice that the same double quote symbol is used at each end of the string; there are not separate left and right quote symbols.

Also, notice the spaces inside the quotes. The computer does not insert any extra space before or after the items output by a cout statement. That is why the quoted strings in the samples often start and/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;
```

As we noted in Chapter 1, `\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 of the quotes. In C++, going to the next line is considered to be a special character (special symbol) 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 **new-line character**.

## Include Directives and Namespaces

We have started all of our programs with the following two lines:

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

These two lines make the library `iostream` available. This is the library that includes, among other things, the definitions of `cin` and `cout`. So if your program uses either `cin` or `cout`, you should have these two lines at the start of the file that contains your program.

The following line is known as an **include directive**. It “includes” the library `iostream` in your program so that you have `cin` and `cout` available:

```
#include <iostream>
```

The operators `cin` and `cout` are defined in a file named `iostream` and the above `include` directive is equivalent to copying that named file into your program. The second line is a bit more complicated to explain.

C++ divides names into **namespaces**. A namespace is a collection of names, such as the names `cin` and `cout`. A statement that specifies a namespace in the way illustrated by the following is called a **using directive**:

```
using namespace std;
```

This particular *using* directive says that your program is using the `std` (“standard”) namespace. This means that the names you use will have the meaning defined for them in the `std` namespace. In this case, the important thing is that when names such as `cin` and `cout` were defined in `iostream`, their definitions said they were in the `std` namespace. So to use names like `cin` and `cout`, you need to tell the compiler you are using `namespace std`;

That is all you need to know (for now) about namespaces, but a brief clarifying remark will remove some of the mystery that might surround the use of *namespace*. The reason that C++ has namespaces at all is because there

are so many things to name. As a result, sometimes two or more items receive the same name; that is, a single name can get two different definitions. To eliminate these ambiguities, C++ divides items into collections so that no two items in the same collection (the same namespace) have the same name.

Note that a namespace is not simply a collection of names. It is a body of C++ code that specifies the meaning of some names, such as some definitions and/or declarations. The function of namespaces is to divide all the C++ name specifications into collections (called *namespaces*) such that each name in a namespace has only one specification (one “definition”) in that namespace. A namespace divides up the names, but it takes a lot of C++ code along with the names.

What if you want to use two items in two different namespaces such that both items have the same name? It can be done and is not too complicated, but that is a topic for later in the book. For now, we do not need to do this.

Some versions of C++ use the following, older form of the `include` directive (without any `using namespace`):

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

If your programs do not compile or do not run with

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

then try using the following line instead of the previous two lines:

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

If your program requires `iostream.h` instead of `iostream`, then you have an old C++ compiler and should obtain a more recent compiler.

## Escape Sequences

The backslash, `\`, preceding a character tells the compiler that the character following the `\` 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 `\` or a `"` 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 backslash, and `\"` means a real quote, not a string constant end.

A stray `\`, say `\z`, in a string constant will on one compiler simply give back a `z`; on another it will produce an error. The ANSI Standard provides that the 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. We list a few here. The most common escape sequence is `\n` for new line.



```
new line      \n
horizontal tab \t
alert        \a
backslash    \\
double quote \"
```

Alternately, C++11 supports a format called **raw string literals**, which is convenient if you have many escape characters. In this format use an `R` followed by the string in parentheses. For example, the following line outputs the literal string `"c:\files\"`:

```
cout << R"(c:\files\);
```

If you wish to insert a blank line in the output, you can output the new-line 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 of quotes and `endl` should not be placed in quotes.

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 above `cout` statement is as follows:

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

## ■ PROGRAMMING TIP End Each Program with a `\n` or `endl`

It is a good idea to output a new-line 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 action in your program. This serves two purposes. Some compilers will not output the last line of your program unless you include a new-line instruction at the end. On other systems, your program may work fine without this final new-line 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 new-line instruction at the end will make your programs more portable. ■

## Formatting for Numbers with a Decimal Point

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 we will explain in Section 2.3):

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

But it is extremely unlikely that the output will be the following, 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.

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);
```

If you insert the preceding three statements in your program, then any `cout` statement that follows these three statements will output values of type *double* 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 6. 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 5 for all subsequent values of type *double* that are output:

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

## Input Using `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 2.1, the variables `numberOfBars` and `oneWeight` were filled by the following `cin` statements (shown along with the `cout` statements that tell the user what to do):

### 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 that allows for digits after the decimal point) will be output in ordinary, everyday 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.

```
cout << "Enter the number of candy bars in a package\n";  
cout << "and the weight in ounces of one candy bar.\n";  
cout << "Then press return.\n";  
cin >> numberOfBars;  
cin >> oneWeight;
```

You can list more than one variable in a single `cin` statement. So the preceding lines could be rewritten to the following:

```
cout << "Enter the number of candy bars in a package\n";  
cout << "and the weight in ounces of one candy bar.\n";  
cout << "Then press return.\n";  
cin >> numberOfBars >> oneWeight;
```

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

```
cin >> numberOfBars  
    >> oneWeight;
```

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

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.

Numbers in the input must be separated by one or more spaces or by a line break. If, for instance, you want to enter the two numbers 12 and 5 and instead you enter the numbers without any space between them, then the computer will think you have entered the single number 125. 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.

### **cin Statements**

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

#### **SYNTAX**

```
cin >> variable1 >> variable2 >> ... ;
```

#### **EXAMPLE**

```
cin >> number >> size;  
cin >> timeToGo  
    >> pointsNeeded;
```

## Designing Input and Output

Input and output, or, as it is often called, **I/O**, is the part of the program that the user sees, so the user will not be happy with a program unless the program has well-designed I/O.

When the computer executes a `cin` statement, it expects some data to be typed in at the keyboard. If none is typed in, the computer simply waits for it. The program must tell the user when to type in a number (or other data item). The computer will not automatically ask the user to enter data. That is why the sample programs contain output statements like the following:

```
cout << "Enter the number of candy bars in a package\n";  
cout << "and the weight in ounces of one candy bar.\n";  
cout << "Then press return.\n";
```

These output statements **prompt** the user to enter the input. Your programs should always prompt for input.

When entering input from a terminal, the input appears on the screen as it is typed in. Nonetheless, the program should always write out the input values some time before it ends. This is called **echoing the input**, and it serves as a check to see that the input was read in correctly. Just because the input looks good on the screen when it is typed in does not mean that it was read correctly by the computer. There could be an unnoticed typing mistake or other problem. Echoing input serves as a test of the integrity of the input data.

### ■ PROGRAMMING 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
```



## SELF-TEST EXERCISES

8. 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.

9. Give an input statement that will fill the variable `the_number` (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.
10. 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?
11. Write a complete C++ program that writes the phrase `Hello world` to the screen. The program does nothing else.
12. Write a complete C++ program that reads in two whole numbers and outputs their sum. Be sure to prompt for input, echo input, and label all output.
13. Give an output statement that produces the new-line character and a tab character.
14. Write a short program that declares and initializes `double` variables `one`, `two`, `three`, `four`, and `five` to the values 1.000, 1.414, 1.732, 2.000, and 2.236, respectively. Then write output statements to generate the following legend and table. Use the tab escape sequence `\t` to line up the columns. If you are unfamiliar with the tab character, you should experiment with it while doing this exercise. A tab works like a mechanical stop on a typewriter. A tab causes output to begin in a next column, usually a multiple of eight spaces away. Many editors and most word processors will have adjustable tab stops. Our output does not.

The output should be:

N	Square Root
1	1.000
2	1.414
3	1.732
4	2.000
5	2.236

## 2.3 DATA TYPES AND EXPRESSIONS

*They'll never be happy together. He's not her type.*

OVERHEARD AT A COCKTAIL PARTY

### The Types *int* and *double*

Conceptually, the numbers 2 and 2.0 are the same number. But C++ considers them to be of different types. The whole number 2 is of type *int*; the number 2.0 is of type *double*, because it contains a fraction part (even though the fraction is 0). Once again, the mathematics of computer programming is a bit different from what you may have learned in mathematics classes. Something about the practicalities of computers makes a computer's numbers differ from the abstract definitions of these numbers. The whole numbers in C++ behave as you would expect them to. The type *int* holds no surprises. But values of type *double* are more troublesome. Because it can store only a limited number of significant digits, the computer stores numbers of type *double* as approximate values. Numbers of type *int* are stored as exact values. The precision with which *double* values are stored varies from one computer to another, but you can expect them to be stored with 14 or more digits of accuracy. For most applications this is likely to be sufficient, though subtle problems can occur even in simple cases. Thus, if you know that the values in some variable will always be whole numbers in the range allowed by your computer, it is best to declare the variable to be of type *int*.

Number constants of type *double* are written differently from those of type *int*. Constants of type *int* 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. There is,

#### What Is Doubled?

Why is the type for numbers with a fraction 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 was therefore much more precise; it also allowed numbers that were larger (although programmers tend to care more about precision than about size). The kind of numbers that used twice as much storage were called *double-precision* numbers;

(continued)

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*. These types are described in the subsection entitled “Other Number Types.” However, we will rarely use the types *float* and *long double* in this book.

however, one thing that constants of type *double* and constants of type *int* have in common: No number in C++ may contain a comma.

The more complicated notation for constants of type *double* is frequently called **scientific notation** or **floating-point notation** and is particularly handy for writing very large numbers and very small fractions. For instance,

$$3.67 \times 10^{17}$$

which is the same as

367000000000000000.0

is best expressed in C++ by the constant 3.67e17. The number

$$5.89 \times 10^{-6}$$

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.”

This **e notation** is used because keyboards normally have no way to write exponents as superscripts. 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.

Since computers have size limitations on their memory, numbers are typically stored in a limited number of bytes (that is, a limited amount of storage). Hence, there is a limit to how large the magnitude of a number can be, and this limit is different for different number types. The largest allowable



number of type *double* is always much larger than the largest allowable number of type *int*. Most current implementations of C++ will allow values of type *int* as large as 2,147,483,647 and values of type *double* up to about  $10^{308}$ .

## Other Number Types

C++ has other numeric types besides *int* and *double*. Some are described in Display 2.2. The various number types allow for different size numbers and for more or less precision (that is, more or fewer digits after the decimal point). In Display 2.2, the values given for memory used, size range, and precision are only one sample set of values, intended to give you a general feel for how the types differ. The values vary from one system to another and may be different on your system.

Although some of these other numeric types are spelled as two words, you declare variables of these other types just as you declare variables of types *int* and *double*. For example, the following declares one variable of type *long double*:

```
long double bigNumber;
```

The type names *long* and *long int* are two names for the same type. Thus, the following two declarations are equivalent:

```
long bigTotal;
```

and the equivalent

```
long int bigTotal;
```

Of course, in any one program, you should use only one of the above two declarations for the variable *bigTotal*, but it does not matter which one you use. Also, remember that the type name *long* by itself means the same thing as *long int*, not the same thing as *long double*.

The types for whole numbers, such as an *int* and similar types, are called **integer types**. The type for numbers with a decimal point—such as the type *double* and similar types—are called **floating-point types**. They are called *floating-point* because when the computer stores a number written in the usual way, like 392.123, it first converts the number to something like *e* notation, in this case something like 3.92123e2. When the computer performs this conversion, the decimal point *floats* (that is, moves) to a new position.

You should be aware that there are other numeric types in C++. However, in this book we will use only the types *int*, *double*, and occasionally *long*. For most simple applications, you should not need any types except *int* and *double*. However, if you are writing a program that uses very large whole numbers, then you might need to use the type *long*.

**DISPLAY 2.2** Some Number Types

Type Name	Memory Used	Size Range	Precision
<i>short</i> (also called <i>short int</i> )	2 bytes	-32,768 to 32,767	(not applicable)
<i>int</i>	4 bytes	-2,147,483,648 to 2,147,483,647	(not applicable)
<i>long</i> (also called <i>long int</i> )	4 bytes	-2,147,483,648 to 2,147,483,647	(not applicable)
<i>float</i>	4 bytes	approximately $10^{-38}$ to $10^{38}$	7 digits
<i>double</i>	8 bytes	approximately $10^{-308}$ to $10^{308}$	15 digits
<i>long double</i>	10 bytes	approximately $10^{-4932}$ to $10^{4932}$	19 digits

*These 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.*

**C++11 Types**

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 were 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 `<stdint.h>`. Display 2.3 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.

C++11 also includes 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, the following line of code defines a variable named `x` whose data type matches whatever is computed from “expression”:

```
auto x = expression;
```

This feature doesn’t buy us much at this point but will save us some long, messy code when we start to work with longer data types that we define ourselves.

DISPLAY 2.3 Some 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	

In the other direction, C++11 also introduces 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`.



The Type *char*

We do not want to give you the impression that computers and C++ are used only for numeric calculations, so we will introduce some nonnumeric types now, though eventually we will see other more complicated nonnumeric types. Values of the type *char*, which is short for *character*, are single symbols such as a letter, digit, or punctuation mark. Values of this type are frequently called *characters* in books and in conversation, but in a C++ program this type must always be spelled in the abbreviated fashion *char*. For example, the variables `symbol` and `letter` of type *char* are declared as follows:

```
char symbol, letter;
```

A variable of type *char* can hold any single character on the keyboard. So, for example, the variable `symbol` could hold an 'A' or a '+' or an 'a'. Note that uppercase and lowercase versions of a letter are considered different characters.

The text in double quotes that are output using `cout` are called *string* values. For example, the following, which occurs in the program in Display 2.1, is a string:

"Enter the number of candy bars in a package\n"

Be sure to notice that string constants are placed inside of double quotes, while constants of type *char* are placed inside of 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.

The use of the type *char* is illustrated in the program shown in Display 2.4. Notice that the user types a space between the first and second initials. Yet the program skips over the blank and reads the letter B as the second input character. When you use *cin* to read input into a variable of type *char*, the computer skips over all blanks and line breaks until it gets to the first nonblank character and reads that nonblank character into the variable. It makes no difference whether there are blanks in the input or not. The program in Display 2.4 will

#### DISPLAY 2.4 The Type *char*

---

```

1  #include <iostream>
2  using namespace std;
3  int main( )
4  {
5      char symbol1, symbol2, symbol3;

6      cout << "Enter two initials, without any periods:\n";
7      cin >> symbol1 >> symbol2;
8      cout << "The two initials are:\n";
9      cout << symbol1 << symbol2 << endl;
10     cout << "Once more with a space:\n";
11     symbol3 = ' ';
12     cout << symbol1 << symbol3 << symbol2 << endl;
13     cout << "That's all.";
14     return 0;
15 }
```

---

#### Sample Dialogue

```

Enter two initials, without any periods:
J B
The two initials are:
JB
Once more with a space:
J B
That's all.
```

---

give the same output whether the user types in a blank between initials, as shown in the sample dialogue, or the user types in the two initials without a blank, like so:

JB

## The Type *bool*

The next type we discuss here is the type *bool*. This type was added to the C++ language by the ISO/ANSI (International Standards Organization/American National Standards Organization) committee in 1998. Expressions of type *bool* are called *Boolean* after the English mathematician George Boole (1815–1864), who formulated rules for mathematical logic.

Boolean expressions evaluate to one of the two values, *true* or *false*. Boolean expressions are used in branching and looping statements that we study in Section 2.4. We will say more about Boolean expressions and the type *bool* in that section.

## Introduction to the Class *string*

Although C++ lacks a native data type to directly manipulate strings, 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 native data type is discussed in Chapter 10. Further details about the *string* class are discussed in Chapter 8.

To use the *string* class we must first include the *string* library:

```
#include <string>
```

Your program must also contain the following line of code, normally placed at the start of the file:

```
using namespace std;
```

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 "Monday" in it:

```
string day;  
day = "Monday";
```

You may use *cin* and *cout* to read data into strings, as shown in Display 2.5. If you place the '+' symbol between two strings, then this operator concatenates the two strings together to create one longer string. For example, the code:

```
string day, day1, day2;  
day1 = "Monday";
```

```
day2 = "Tuesday";
day = day1 + day2;
```

Results in the concatenated string of:

```
"MondayTuesday"
```

Note that a space is not automatically added between the strings. If you wanted a space between the two days, then a space must be added explicitly:

```
day1 + " " + day2
```

---

### DISPLAY 2.5 The string Class

---

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main()
5  {
6      string middleName, petName;
7      string alterEgoName;
8
9      cout << "Enter your middle name and the name of your pet.\n";
10     cin >> middleName;
11     cin >> petName;
12
13     alterEgoName = petName + " " + middleName;
14
15     cout << "The name of your alter ego is ";
16     cout << alterEgoName << "." << endl;
17
18     return 0;
19 }
```

---

#### *Sample Dialogue 1*

```
Enter your middle name and the name of your pet.
Parker Pippin
The name of your alter ego is Pippin Parker.
```

---

#### *Sample Dialogue 2*

```
Enter your middle name and the name of your pet.
Parker
Mr. Bojangles
The name of your alter ego is Mr. Parker.
```

---

When you use `cin` to read input into a `string` variable, the computer only reads until it encounters a *whitespace* character. **Whitespace** characters are all the characters that are displayed as blank spaces on the screen, including the blank or space character, the tab character, and the new-line character `'\n'`. This means that you cannot input a string that contains spaces. This may sometimes cause errors, as indicated in Display 2.5, 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 8 describes a technique to input a string that may include spaces.

## Type Compatibilities

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 above assignment statement. Some will issue an error message, some will give only a warning message, and some compilers will not object at all. But even if the compiler does allow you to use this assignment, it will probably give `intVariable` the *int* value 2, not the value 3. Since you cannot count on your compiler accepting this assignment, you should not assign a *double* value to a variable of type *int*.

The same problem arises if you use a variable of type *double* instead of the constant 2.99. Most compilers will also object to the following:

```
int intVariable;  
double doubleVariable;  
doubleVariable = 2.00;  
intVariable = doubleVariable;
```

The fact that the value 2.00 "comes out even" makes no difference. The value 2.00 is of type *double*, not of type *int*. As you will see shortly, you can replace 2.00 with 2 in the preceding assignment to the variable `doubleVariable`, but even that is not enough to make the assignment acceptable. The variables `intVariable` and `doubleVariable` are of different types, and that is the cause of the problem.

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. For example, if your compiler lets you assign 2.99 to a variable of type *int*, the variable will receive the value 2, rather than 2.99, which can be confusing since the program seems to say the value will be 2.99.

There are some special cases where it is permitted to assign a value of one type to a variable of another type. It is acceptable to assign a value of type *int* to a variable of type *double*. For example, the following is both legal and acceptable style:

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

The above 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 the 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* and so 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.

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 very strictly, it is a good rule to follow. Placing data of one type in a variable of another type can cause problems, since the value must be changed to a value of the appropriate type and that value may not be what you would expect.

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 and you should not use these features. For completeness and to help you read other people's code, we do give 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.

## Arithmetic Operators and Expressions

In a C++ program, you can combine variables and/or numbers using the arithmetic operators + for addition, - for subtraction, \* for multiplication, and / for division. For example, the following assignment statement, which appears in the program in Display 2.1, uses the \* operator to multiply the numbers in two variables. (The result is then placed in the variable on the left-hand side of the equal sign.)

```
totalWeight = oneWeight * numberOfBars;
```

All of 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 depends 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 is 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 is 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 `/`.

The type of the result can be more significant than you might suspect. For example, `7.0/2` has one operand of type *double*, namely `7.0`. Hence, the result is the type *double* number `3.5`. However, `7/2` has two operands of type *int* and so it yields the type *int*, which is the result `3`. Even if the result “comes out even,” there is a difference. For example, `6.0/2` has one operand of type *double*, namely `6.0`. Hence, the result is the type *double* number `3.0`, which is only an approximate quantity. However, `6/2` has two operands of type *int*, so it yields the result `3`, which is of type *int* and so is an exact quantity. The division operator is the operator that is affected most severely by the type of its arguments.

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 `%` 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) << endl;
cout << "with a remainder of " << (17%5) << endl;
```

yield the following output:

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

Display 2.6 illustrates how `/` and `%` work with values of type `int`.

### DISPLAY 2.6 Integer Division

$\begin{array}{r} 4 \leftarrow 12/3 \\ 3 \overline{)12} \\ 12 \\ \hline 0 \leftarrow 12\%3 \end{array}$	$\begin{array}{r} 4 \leftarrow 14/3 \\ 3 \overline{)14} \\ 12 \\ \hline 2 \leftarrow 14\%3 \end{array}$
---	---

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 non-negative.

Any reasonable spacing will do in arithmetic expressions. You can insert spaces before and after operations and parentheses, or you can omit them. Do whatever produces a result that is easy to read.

You can specify the order of operations by inserting parentheses, as illustrated in the following two expressions:

$$\begin{aligned} &(x + y) * z \\ &x + (y * z) \end{aligned}$$

To evaluate the first expression, the computer first adds `x` and `y` and then multiplies the result by `z`. To evaluate the second expression, it multiplies `y` and `z` and then adds the result to `x`. Although you may be used to using mathematical formulas that contain square brackets and various other forms of parentheses, that is not allowed in C++. C++ allows only one kind of parentheses in arithmetic expressions. The other varieties are reserved for other purposes.

If you omit parentheses, the computer will follow rules called **precedence rules** that determine the order in which the operators, such as `+` and `*`, 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 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.

Display 2.7 shows some examples of common kinds of arithmetic expressions and how they are expressed in C++.



VideoNote  
Precedence and Arithmetic  
Operators

DISPLAY 2.7 Arithmetic Expressions

Mathematical Formula	C++ Expression
$b^2 - ; 4ac$	<code>b*b - ; 4*a*c</code>
$x(y+z)$	<code>x*(y+z)</code>
$\frac{1}{x^2+x+3}$	<code>1/(x*x+x+3)</code>
$\frac{a+b}{c-d}$	<code>(a+b)/(c - ; d)</code>

PITFALL Whole Numbers in Division

When you use the division operator `/` on two whole numbers, the result is a whole number. 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 your 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`, which you think you are getting). So the value assigned to `totalCost` is `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

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

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

16. What is the output of the following program lines when 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';
```

17. What is the output of the following program lines when 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;
```

18. 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 *%*.

19. 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;
```

- What value is assigned to *f*?
  - Explain what is actually happening, and what the programmer likely wanted.
  - Rewrite the code as the programmer intended.
20. What is the output of the following program lines when embedded in a correct program that declares *month*, *day*, *year*, and *date* to be of type *string*?

```
month = "03";
day = "04";
year = "06";
date = month + day + year;
cout << date << endl;
```

### More Assignment Statements

There is a shorthand notation 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

$$\text{Variable Op} = \text{Expression}$$

which is equivalent to

$$\text{Variable} = \text{Variable Op} (\text{Expression})$$

Op is an operator such as +, \*, or /. The *Expression* can be another variable, a constant, or a more complicated arithmetic expression. Following are 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);

## 2.4 SIMPLE FLOW OF CONTROL

*"If you think we're wax-works," he said, "you ought to pay, you know. Wax-works weren't made to be looked at for nothing. Nohow!"*

*"Contrariwise," added the one marked "DEE," "if you think we're alive, you ought to speak."*

LEWIS CARROLL, *Through the Looking-Glass*

The programs you have seen thus far each consist of a simple list of statements to be executed in the order given. However, to write more sophisticated programs, you will also need some way to vary the order in which statements are executed. The order in which statements are executed is often referred to as **flow of control**. In this section we will present two simple ways to add some flow of control to your programs. We will discuss a branching mechanism that lets your program choose between two alternative actions, choosing one or the other depending on the values of variables. We will also present a looping mechanism that lets your program repeat an action a number of times.

## A Simple Branching Mechanism

Sometimes it is necessary to have a program choose one of two alternatives, depending on the input. 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. As long as the employee works 40 or more hours, the pay is then equal to

$$\text{rate} * 40 + 1.5 * \text{rate} * (\text{hours} - 40)$$

However, if there is a possibility that the employee will work less than 40 hours, this formula will unfairly pay a negative amount of overtime. (To see this, just substitute 10 for hours, 1 for rate, and do the arithmetic. The poor employee will get a negative paycheck.) The correct pay formula for an employee who works less than 40 hours is simply

$$\text{rate} * \text{hours}$$

If both more than 40 hours and less than 40 hours of work are possible, then the program will need to choose between the two formulas. In order to compute the employee's pay, the program action should be

Decide whether or not (hours > 40) is true.

If it is, do the following assignment statement:

$$\text{grossPay} = \text{rate} * 40 + 1.5 * \text{rate} * (\text{hours} - 40);$$

If it is not, do the following:

$$\text{grossPay} = \text{rate} * \text{hours};$$

There is a C++ statement that does exactly this kind of branching action. The *if-else* **statement** chooses between two alternative actions. For example, the wage calculation we have been discussing can be accomplished with the following C++ statement:

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

A complete program that uses this statement is given in Display 2.8.

Two forms of an *if-else* statement are described in Display 2.9. The first is the simple form of an *if-else* statement; the second form will be discussed in the subsection entitled "Compound Statements." In the first form shown, the two statements may be any executable statements. The *Boolean\_Expression* is a test that can be checked to see if it is true or false, that is, to see if it is satisfied or not. For example, the *Boolean\_Expression* in the earlier *if-else* statement is

$$\text{hours} > 40$$

When the program reaches the *if-else* statement, exactly one of the two embedded statements is executed. If the *Boolean\_Expression* is true (that is, if it is satisfied), then the *Yes\_Statement* is executed; if the *Boolean\_Expression* is false (that is, if it is not satisfied), then the *No\_Statement* is executed. Notice that the *Boolean\_Expression* must be enclosed in parentheses. (This is required by the syntax rules for *if-else* statements in C++.) Also notice that an *if-else* statement has two smaller statements embedded in it.

---

### DISPLAY 2.8 An if-else Statement (part 1 of 2)

---

```

1    #include <iostream>
2    using namespace std;
3    int main( )
4    {
5        int hours;
6        double grossPay, rate;
7        cout << "Enter the hourly rate of pay: $";
8        cin >> rate;
9        cout << "Enter the number of hours worked,\n"
10           << "rounded to a whole number of hours: ";
11        cin >> hours;
12        if (hours > 40)
13            grossPay = rate * 40 + 1.5 * rate * (hours - 40);
14        else
15            grossPay = rate * hours;
16
17        cout.setf(ios::fixed);
18        cout.setf(ios::showpoint);
19        cout.precision(2);
20        cout << "Hours = " << hours << endl;
21        cout << "Hourly pay rate = $" << rate << endl;
22        cout << "Gross pay = $" << grossPay << endl;
23        return 0;
24    }

```

---

#### Sample Dialogue 1

```

Enter the hourly rate of pay: $20.00
Enter the number of hours worked,
rounded to a whole number of hours: 30
Hours = 30
Hourly pay rate = $20.00
Gross pay = $600.00

```

(continued)

---

**DISPLAY 2.8** An *if-else* Statement (*part 2 of 2*)

---

*Sample Dialogue 2*

```
Enter the hourly rate of pay: $10.00
Enter the number of hours worked,
rounded to a whole number of hours: 41
Hours = 41
Hourly pay rate = $10.00
Gross pay = $415.00
```

---

---

**DISPLAY 2.9** Syntax for an *if-else* Statement

---

*A Single Statement for Each Alternative:*

```
1  if (Boolean_Expression)
2      Yes_Statement
3  else
4      No_Statement
```

*A Sequence of Statements for Each Alternative:*

```
5  if (Boolean_Expression)
6  {
7      Yes_Statement_1
8      Yes_Statement_2
9      ...
10     Yes_Statement_Last
11 }
12 else
13 {
14     No_Statement_1
15     No_Statement_2
16     ...
17     No_Statement_Last
18 }
```

---

A **Boolean expression** is any expression that is either true or false. An *if-else* statement always contains a *Boolean\_Expression*. The simplest form for a *Boolean\_Expression* consists of two expressions, such as numbers or variables, that are compared with one of the comparison operators shown in Display 2.10. Notice that some of the operators are spelled with two symbols: for example, `==`, `!=`, `<=`, `>=`. Be sure to notice that you use a double equal `==` for the equal sign, and