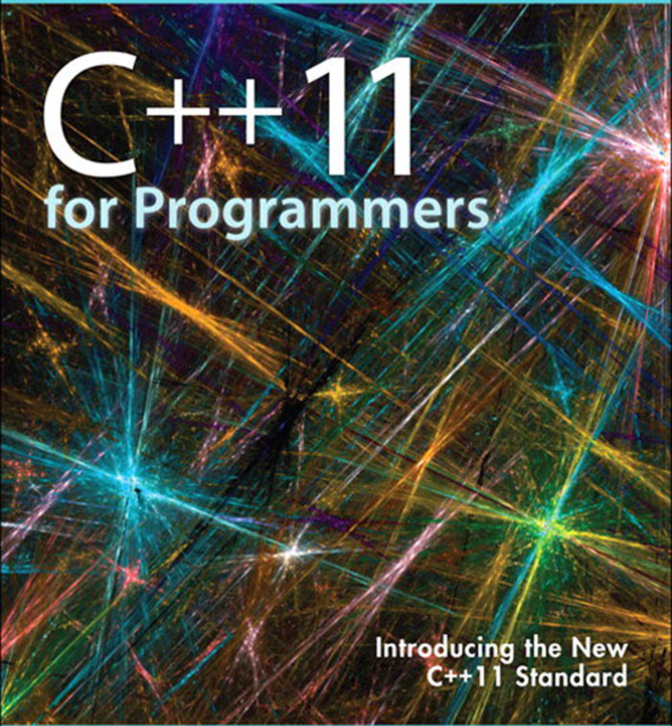
DEITEL® DEVELOPER SERIES

C++11 for Programmers



Introducing the New C++11 Standard

PAUL DEITEL • HARVEY DEITEL

C++11 for Programmers
Second Edition
Deitel® Developer Series

Library of Congress Cataloging-in-Publication Data

On file

ISBN-13: 978-0-13-343985-4
ISBN-10: 0-13-343985-2

Text printed in the United States at RR Donnelley in Crawfordsville, Indiana..
First printing, February 2013

C++11 for Programmers
Second Edition
Deitel® Developer Series

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

DEITEL®

PRENTICE HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Trademarks

To our review team:

Dean Michael Berris
Danny Kalev
Linda M. Krause
James P. McNellis
Robert C. Seacord
José Antonio González Seco

We are grateful for your guidance and expertise.

Paul and Harvey Deitel

This page intentionally left blank

Contents

Chapter 24 and Appendices F–K are PDF documents posted online at
`www.informit.com/title/9780133439854`

Online Chapters and Appendices

Chapter 24 and Appendices F–K are PDF documents posted online at
`www.informit.com/title/9780133439854`

[*Note:* The test drives for Windows and Linux are in Chapter 1.]

Preface

"The chief merit of language is clearness …"
—Galen

Welcome to *C++11 for Programmers*! This book presents leading-edge computing technologies for software developers.

We focus on software engineering best practices. At the heart of the book is the Deitel signature "live-code approach"—concepts are presented in the context of complete working programs, rather than in code snippets. Each complete code example is accompanied by live sample executions. All the source code is available at

```
www.deitel.com/books/cpp11fp
```

As you read the book, if you have questions, we're easy to reach at

```
deitel@deitel.com
```

We'll respond promptly. For book updates, visit `www.deitel.com/books/cpp11fp`. Join our social media communities on Facebook (`www.deitel.com/DeitelFan`), Twitter (`@deitel`), Google+ (`gplus.to/deitel`) and LinkedIn (`bit.ly/DeitelLinkedIn`), and subscribe to the *Deitel® Buzz Online* newsletter (`www.deitel.com/newsletter/subscribe.html`).

Features

Here are the key features of *C++11 for Programmers*.

C++11 Standard

The new C++11 standard, published in 2011, motivated us to write *C++11 for Programmers*. Throughout the book, each new C++11 feature we discuss is marked with the "11" icon you see here in the margin. These are some of the key C++11 features of this new edition:

- ***Conforms to the new C++11 standard.*** Extensive coverage of many of the key new C++11 features (Fig. 1).
- ***Code thoroughly tested on three popular industrial-strength C++11 compilers.*** We tested the code examples on GNU™ C++ 4.7, Microsoft® Visual C++® 2012 and Apple® LLVM in Xcode® 4.5.
- ***Smart pointers.*** Smart pointers help you avoid dynamic memory management errors by providing additional functionality beyond that of built-in pointers. We discuss `unique_ptr` in Chapter 17, and `shared_ptr` and `weak_ptr` in Chapter 24.

C++11 features in *C++11 for Programmers*

`all_of` algorithm
`any_of` algorithm
`array` container
`auto` for type inference
`begin`/`end` functions
`cbegin`/`cend` container member functions
Compiler fix for >> in template types
`copy_if` algorithm
`copy_n` algorithm
`crbegin`/`crend` container member functions
`decltype`
Default type arguments in function templates
`default`ed member functions
Delegating constructors
`delete`d member functions
`explicit` conversion operators
`final` classes
`final` member functions
`find_if_not` algorithm
`forward_list` container
Immutable keys in associative containers
In-class initializers
Inheriting base-class constructors
`insert` container member functions return iterators
`is_heap` algorithm
`is_heap_until` algorithm
Keywords new in C++11
Lambda expressions
List initialization of key–value `pairs`
List initialization of `pair` objects
List initialization of return values
List initializing a dynamically allocated array
List initializing a `vector`
List initializers in constructor calls
`long long int` type
`min` and `max` algorithms with `initializer_list` parameters
`minmax` algorithm
`minmax_element` algorithm
`move` algorithm
Move assignment operators
`move_backward` algorithm
Move constructors
`noexcept`
Non-deterministic random number generation
`none_of` algorithm
Numeric conversion functions
`nullptr`
`override` keyword
Range-based `for` statement
Regular expressions
Rvalue references
Scoped `enums`
`shared_ptr` smart pointer
`shrink_to_fit` `vector`/`deque` member function
Specifying the type of an `enum`'s constants
`static_assert` objects for file names
`string` objects for file names
`swap` non-member function
Trailing return types `for` functions
`tuple` variadic template
`unique_ptr` smart pointer
Unsigned `long long int`
`weak_ptr` smart pointer

Fig. 1 | *A sampling of C++11 features in C++11 for Programmers.*

- ***Earlier coverage of template-based Standard Library containers, iterators and algorithms, enhanced with C++11 capabilities.*** We moved the treatment of Standard Library containers, iterators and algorithms from Chapter 20 in the previous edition to Chapters 15 and 16 and enhanced it with new C++11 features. The vast majority of your data structure needs can be fulfilled by *reusing* these Standard Library capabilities.
- ***Online Chapter 24, C++11: Additional Topics.*** In this chapter, we present additional C++11 topics. The new C++11 standard has been available since 2011, but not all C++ compilers have fully implemented the features. If all three of our key compilers already implemented a particular C++11 feature at the time we wrote this book, we generally integrated a discussion of that feature into the text with a live-code example. If any of these compilers had *not* implemented that feature, we included a bold italic heading followed by a brief discussion of the feature. Many of those discussions will be expanded in online Chapter 24 as the features are implemented. Placing the chapter online allows us to evolve it dynamically. This

chapter includes discussions of regular expressions, the `shared_ptr` and `weak_ptr` smart pointers, move semantics and more. You can access this chapter at:

```
www.informit.com/title/9780133439854
```

- ***Random Number generation, simulation and game playing.*** To help make programs more secure (see Secure C++ Programming on the next page), we now discuss C++11's new non-deterministic random-number generation capabilities.

Object-Oriented Programming

- ***Early-objects approach.*** The book introduces the basic concepts and terminology of object technology in Chapter 1. You'll develop your first customized C++ classes and objects in Chapter 3.
- ***C++ Standard Library* `string`*.*** C++ offers *two* types of strings—`string` class objects (which we begin using in Chapter 3) and C strings (from the C programming language). We've replaced most occurrences of C strings with instances of C++ class `string` to make programs more robust and eliminate many of the security problems of C strings. We discuss C strings later in the book to prepare you for working with the legacy code in industry. In new development, you should favor `string` objects.
- ***C++ Standard Library* `array`*.*** Our primary treatment of arrays now uses the Standard Library's `array` class template instead of built-in, C-style, pointer-based arrays. We also cover built-in arrays because they still have some uses in C++ and so that you'll be able to read legacy code. C++ offers *three* types of arrays—class templates `array` and `vector` (which we start using in Chapter 7) and C-style, pointer-based arrays which we discuss in Chapter 8. As appropriate, we use class template `array` and occasionally, class template `vector`, instead of C arrays throughout the book. In new development, you should favor class templates `array` and `vector`.
- ***Crafting valuable classes.*** A key goal of this book is to prepare you to build valuable reusable C++ classes. In the Chapter 10 case study, you'll build your own custom `Array` class. Chapter 10 begins with a test-drive of class template `string` so you can see an elegant use of operator overloading before you implement your own customized class with overloaded operators.
- ***Case studies in object-oriented programming.*** We provide case studies that span multiple sections and chapters and cover the software development lifecycle. These include the `GradeBook` class in Chapters 3–7, the `Time` class in Chapter 9 and the `Employee` class in Chapters 11–12. Chapter 12 contains a detailed diagram and explanation of how C++ can implement polymorphism, `virtual` functions and dynamic binding "under the hood."
- ***Optional case study: Using the UML to develop an object-oriented design and C++ implementation of an ATM.*** The UML™ (Unified Modeling Language™) is the industry-standard graphical language for modeling object-oriented systems. We introduce the UML in the early chapters. Chapters 22 and 23 include an *optional* case study on object-oriented design using the UML. We design and implement the software for a simple automated teller machine (ATM). We analyze a typical requirements document that specifies the system to be built. We determine the classes

needed to implement that system, the attributes the classes need to have, the behaviors the classes need to exhibit and we specify how the classes must interact with one another to meet the system requirements. From the design we produce a complete C++ implementation. Readers often report that the case study "ties it all together" and helps them achieve a deeper understanding of object orientation.

- ***Exception handling.*** We integrate basic exception handling *early* in the book. You can easily pull more detailed material forward from Chapter 17, Exception Handling: A Deeper Look.
- ***Key programming paradigms.*** We discuss *object-oriented programming* and *generic programming*.

Pedagogic Features

- ***Examples.*** We include a broad range of example programs selected from computer science, business, simulation, game playing and other topics.
- ***Illustrations and figures.*** Abundant tables, line drawings, UML diagrams, programs and program outputs are included.

Other Features

- ***Pointers.*** We provide thorough coverage of the built-in pointer capabilities and the intimate relationship among built-in pointers, C strings and built-in arrays.
- ***Debugger appendices.*** We provide three debugger appendices—Appendix H, Using the Visual Studio Debugger, Appendix I, Using the GNU C++ Debugger and Appendix J, Using the Xcode Debugger.

Secure C++ Programming

It's difficult to build industrial-strength systems that stand up to attacks from viruses, worms, and other forms of "malware." Today, via the Internet, such attacks can be instantaneous and global in scope. Building security into software from the beginning of the development cycle can greatly reduce vulnerabilities.

The CERT® Coordination Center (`www.cert.org`) was created to analyze and respond promptly to attacks. CERT—the Computer Emergency Response Team—is a government-funded organization within the Carnegie Mellon University Software Engineering Institute™. CERT publishes and promotes secure coding standards for various popular programming languages to help software developers implement industrial-strength systems that avoid the programming practices that leave systems open to attacks.

We'd like to thank Robert C. Seacord, Secure Coding Manager at CERT and an adjunct professor in the Carnegie Mellon University School of Computer Science. Mr. Seacord was a technical reviewer for our book, *C How to Program, 7/e*, where he scrutinized our C programs from a security standpoint, recommending that we adhere to the *CERT C Secure Coding Standard*.

We've done the same for *C++11 for Programmers*, adhering to key *CERT C++ Secure Coding Standard* guidelines (as appropriate for a book at this level), which you can find at:

```
www.securecoding.cert.org
```

We were pleased to discover that we've already been recommending many of these coding practices in our books since the early 1990s. If you'll be building industrial-strength C++ systems, *Secure Coding in C and C++, Second Edition* (Robert Seacord, Addison-Wesley Professional) is a must read.

Training Approach

C++11 for Programmers stresses program clarity and concentrates on building well-engineered software.

Live-Code Approach. The book includes hundreds of "live-code" examples—each new concept is presented in the context of a complete working C++ program that is immediately followed by one or more actual executions showing the program's inputs and outputs.

Syntax Shading. For readability, we syntax shade the code, similar to the way most integrated-development environments and code editors syntax color the code. Our syntax-shading conventions are:

```
comments appear like this
keywords appear like this
constants and literal values appear like this
all other code appears in black
```

Code Highlighting. We place light-gray rectangles around each program's key code segments.

Using Fonts for Emphasis. We place the key terms and the index's page reference for each defining occurrence in ***bold italic*** text for easier reference. We emphasize on-screen components in the **bold Helvetica** font (e.g., the **File** menu) and emphasize C++ program text in the `Lucida` font (e.g., `int x = 5`).

Web Access. All of the source-code examples can be downloaded from:

```
www.deitel.com/books/cpp11fp
```

Objectives. The chapter opening quotations are followed by a list of chapter objectives.

Programming Tips. We include hundreds of programming tips to help you focus on important aspects of program development. These tips and practices represent the best we've gleaned from a combined eight decades of programming and teaching experience.

Good Programming Practice

The Good Programming Practices *call attention to techniques that will help you produce programs that are clearer, more understandable and more maintainable.*

Common Programming Error

Pointing out these Common Programming Errors *reduces the likelihood that you'll make them.*

Error-Prevention Tip

These tips contain suggestions for exposing and removing bugs from your programs; many of the tips describe aspects of C++ that prevent bugs from getting into your programs.

Performance Tip

These tips highlight opportunities for making your programs run faster or minimizing the amount of memory that they occupy.

Portability Tip

The Portability Tips *help you write code that will run on a variety of platforms.*

Software Engineering Observation

The Software Engineering Observations *highlight architectural and design issues that affect the construction of software systems, especially large-scale systems.*

Online Chapter and Appendices

The following chapter and appendices are available online:

- Chapter 24, C++11: Additional Features
- Appendix F, C Legacy Code Topics
- Appendix G, UML 2: Additional Diagram Types
- Appendix H, Using the Visual Studio Debugger
- Appendix I, Using the GNU C++ Debugger
- Appendix J, Using the Xcode Debugger
- Appendix K, Test Driving a C++ Program on Mac OS X

To access the online chapter and appendices, go to:

`www.informit.com/register`

You must register for an an InformIT account and then login. After you've logged into your account, you'll see the **Register a Product** box. Enter the book's ISBN (9780133439854) to access the page with the online chapter and appendices.

Obtaining the Software Used in *C++11 for Programmers*

We wrote the code examples in *C++11 for Programmers* using the following C++ development tools:

- Microsoft's free Visual Studio Express 2012 for Windows Desktop, which includes Visual C++ and other Microsoft development tools. This runs on Windows 7 and 8 and is available for download at

  `www.microsoft.com/express`

- GNU's free GNU C++ (`gcc.gnu.org/install/binaries.html`), which is already installed on most Linux systems and can also be installed on Mac OS X and Windows systems.
- Apple's free Xcode, which OS X users can download from the Mac App Store.

C++11 Fundamentals: Parts I, II, III and IV LiveLessons Video Training Product

Our *C++11 Fundamentals: Parts I, II, III and IV* LiveLessons video training product shows you what you need to know to start building robust, powerful software with C++. It includes 20+ hours of expert training synchronized with *C++11 for Programmers*. For additional information about Deitel LiveLessons video products, visit

```
www.deitel.com/livelessons
```

or contact us at `deitel@deitel.com`. You can also access our LiveLessons videos if you have a subscription to Safari Books Online (`www.safaribooksonline.com`). These LiveLessons will be available in the Summer of 2013.

Acknowledgments

We'd like to thank Abbey Deitel and Barbara Deitel of Deitel & Associates, Inc. for long hours devoted to this project. Abbey co-authored Chapter 1 and this Preface, and she and Barbara painstakingly researched the new capabilities of C++11.

We're fortunate to have worked on this project with the dedicated publishing professionals at Prentice Hall/Pearson. We appreciate the extraordinary efforts and mentorship of our friend and professional colleague Mark L. Taub, Editor-in-Chief of Pearson Technology Group. Carole Snyder did a great job recruiting distinguished members of the C++ community to review the manuscript. Chuti Prasertsith designed the cover with creativity and precision—we gave him our vision for the cover and he made it happen. John Fuller does a superb job managing the production of all of our Deitel Developer Series books and LiveLessons video products.

Reviewers

We wish to acknowledge the efforts of the reviewers whose constructive criticisms helped us shape the recent editions of this content. They scrutinized the text and the programs and provided countless suggestions for improving the presentation: Dean Michael Berris (Google, Member ISO C++ Committee), Danny Kalev (C++ expert, certified system analyst and former member of the C++ Standards Committee), Linda M. Krause (Elmhurst College), James P. McNellis (Microsoft Corporation), Robert C. Seacord (Secure Coding Manager at SEI/CERT, author of *Secure Coding in C and C++*); José Antonio González Seco (Parliament of Andalusia), Virginia Bailey (Jackson State University), Thomas J. Borrelli (Rochester Institute of Technology), Ed Brey (Kohler Co.), Chris Cox (Adobe Systems), Gregory Dai (eBay), Peter J. DePasquale (The College of New Jersey), John Dibling (SpryWare), Susan Gauch (University of Arkansas), Doug Gregor (Apple, Inc.), Jack Hagemeister (Washington State University), Williams M. Higdon (University of Indiana), Anne B. Horton (Lockheed Martin), Terrell Hull (Logicalis Integration Solutions), Ed James-Beckham (Borland), Wing-Ning Li (University of Arkansas), Dean Mathias (Utah State University), Robert A. McLain (Tidewater Community College), Robert Myers (Florida State University), Gavin Osborne (Saskatchewan Institute of Applied Science and Technology), Amar Raheja (California State Polytechnic University, Pomona), April Reagan (Microsoft), Raymond Stephenson (Microsoft), Dave Topham (Ohlone College), Anthony Williams (author and C++ Standards Committee member) and Chad Willwerth (University Washington, Tacoma).

As you read the book, we'd sincerely appreciate your comments, criticisms and suggestions for improving the text. Please address all correspondence to:

```
deitel@deitel.com
```

We'll respond promptly. We enjoyed writing *C++11 for Programmers*. We hope you enjoy reading it!

Paul Deitel
Harvey Deitel

About the Authors

Paul Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT, where he studied Information Technology. Through Deitel & Associates, Inc., he has delivered hundreds of programming courses to industry, government and military clients, including Cisco, IBM, Siemens, Sun Microsystems, Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey M. Deitel, are the world's best-selling programming-language textbook/professional book/video authors.

Dr. Harvey Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has more than 50 years of experience in computing. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University. In the 1960s, through Advanced Computer Techniques and Computer Usage Corporation, he worked on the teams building various IBM operating systems. In the 1970s, he built commercial software systems. He has extensive college teaching experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., in 1991 with his son, Paul Deitel. The Deitels' publications have earned international recognition, with translations published in Chinese, Korean, Japanese, German, Russian, Spanish, French, Polish, Italian, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to corporate, academic, government and military clients.

Deitel® Dive-Into® Series Corporate Training

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring and corporate training organization, specializing in computer programming languages, object technology, mobile app development and Internet and web software technology. The company's clients include many of the world's largest corporations, government agencies, branches of the military, and academic institutions. The company offers instructor-led training courses delivered at client sites worldwide on major programming languages and platforms, including C++, Visual C++®, C, Java™, Visual C#®, Visual Basic®, XML®, Python®, object technology, Internet and web programming, Android app development, Objective-C and iOS app development and a growing list of additional programming and software development courses.

Through its 37-year publishing partnership with Prentice Hall/Pearson, Deitel & Associates, Inc., publishes leading-edge programming professional books, college textbooks and *LiveLessons* video courses. Deitel & Associates, Inc. and the authors can be reached at:

```
deitel@deitel.com
```

To learn more about Deitel *Dive-Into® Series* Corporate Training curriculum, visit:

```
www.deitel.com/training
```

To request a proposal for worldwide on-site, instructor-led training at your organization, e-mail `deitel@deitel.com`.

Individuals wishing to purchase Deitel books and *LiveLessons* video training can do so through `www.deitel.com`. Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit

```
www.informit.com/store/sales.aspx
```

2

Introduction to C++ Programming, Input/Output and Operators

Objectives

In this chapter you'll:

- Write simple C++ programs.
- Write input and output statements.
- Use fundamental types.
- Use arithmetic operators.
- Learn the precedence of arithmetic operators.
- Write decision-making statements.

Outline

2.1 Introduction

We now introduce C++ programming. We show how to display messages on the screen and obtain data from the user at the keyboard for processing. We explain how to perform *arithmetic calculations* and save their results for later use. We demonstrate *decision-making* by showing you how to *compare* two numbers, then display messages based on the comparison results.

Compiling and Running Programs

At `www.deitel.com/books/cpp11fp`, we've posted videos that demonstrate compiling and running programs in Microsoft Visual C++, GNU C++ and Xcode.

2.2 First Program in C++: Printing a Line of Text

Consider a simple program that prints a line of text (Fig. 2.1). This program illustrates several important features of the C++ language. The line numbers are *not* part of the source code.

```
1   // Fig. 2.1: fig02_01.cpp
2   // Text-printing program.
3   #include <iostream> // allows program to output data to the screen
4
5   // function main begins program execution
6   int main()
7   {
8      std::cout << "Welcome to C++!\n"; // display message
9
10     return 0; // indicate that program ended successfully
11  } // end function main
```

```
Welcome to C++!
```

Fig. 2.1 | Text-printing program.

Comments

Lines 1 and 2

```
// Fig. 2.1: fig02_01.cpp
// Text-printing program.
```

each begin with `//`, indicating that the remainder of each line is a **comment**. The comment `Text-printing program` describes the purpose of the program. A comment beginning with

// is called a **single-line comment** because it terminates at the end of the current line. [*Note:* You also may use comments containing one or more lines enclosed in /* and */.]

***#include* Preprocessing Directive**

Line 3

```
#include <iostream> // allows program to output data to the screen
```

is a **preprocessing directive**, which is a message to the C++ preprocessor (introduced in Section 1.4). Lines that begin with # are processed by the preprocessor *before* the program is compiled. This line notifies the preprocessor to include in the program the contents of the **input/output stream header <iostream>**. This header is a file containing information used by the compiler when compiling any program that outputs data to the screen or inputs data from the keyboard using C++'s stream input/output. The program in Fig. 2.1 outputs data to the screen, as we'll soon see. We discuss headers in more detail in Chapter 6 and explain the contents of <iostream> in Chapter 13.

Common Programming Error 2.1

Forgetting to include the <iostream> header in a program that inputs data from the keyboard or outputs data to the screen causes the compiler to issue an error message.

Blank Lines and White Space

Line 4 is simply a *blank line*. Together, blank lines, *space characters* and *tab characters* are known as **whitespace**. Whitespace characters are normally *ignored* by the compiler.

*The **main** Function*

Line 5

```
// function main begins program execution
```

is another single-line comment indicating that program execution begins at the next line.

Line 6

```
int main()
```

is a part of every C++ program. The parentheses after main indicate that **main** is a program building block called a **function**. C++ programs typically consist of one or more functions and classes (as you'll learn in Chapter 3). Exactly *one* function in every program *must* be named main. Figure 2.1 contains only one function. C++ programs begin executing at function main, even if main is *not* the first function defined in the program. The keyword int to the left of main indicates that main returns an integer value. The complete list of C++ keywords can be found in Fig. 4.2. We'll say more about return a value when we demonstrate how to create your own functions in Section 3.3. For now, simply include the keyword int to the left of main in each of your programs.

The **left brace**, **{**, (line 7) must *begin* the **body** of every function. A corresponding **right brace**, **}**, (line 11) must *end* each function's body.

An Output Statement

Line 8

```
std::cout << "Welcome to C++!\n"; // display message
```

instructs the computer to perform an action—namely, to print the characters contained between the double quotation marks. Together, the quotation marks and the characters between them are called a **string**, a **character string** or a **string literal**. In this book, we refer to characters between double quotation marks simply as strings. Whitespace characters in strings are not ignored by the compiler.

The entire line 8, including std::cout, the **<< operator**, the string "Welcome to C++!\n" and the **semicolon** (**;**), is called a **statement**. Most C++ statements end with a semicolon, also known as the **statement terminator** (we'll see some exceptions to this soon). Preprocessing directives (like #include) do not end with a semicolon. Typically, output and input in C++ are accomplished with **streams** of characters. Thus, when the preceding statement is executed, it sends the stream of characters Welcome to C++!\n to the **standard output stream object**—**std::cout**—which is normally "connected" to the screen.

Good Programming Practice 2.1

Indent the body of each function one level within the braces that delimit the function's body. This makes a program's functional structure stand out and makes the program easier to read.

Good Programming Practice 2.2

Set a convention for the size of indent you prefer, then apply it uniformly. The tab key may be used to create indents, but tab stops may vary. We prefer three spaces per level of indent.

The std *Namespace*

The std:: before cout is required when we use names that we've brought into the program by the preprocessing directive #include <iostream>. The notation std::cout specifies that we are using a name, in this case cout, that belongs to namespace std. The names cin (the standard input stream) and cerr (the standard error stream)—introduced in Chapter 1—also belong to namespace std. Namespaces are an advanced C++ feature that we discuss in depth in Chapter 21, Other Topics. For now, you should simply remember to include std:: before each mention of cout, cin and cerr in a program. This can be cumbersome—the next example introduces using declarations and the using directive, which will enable you to omit std:: before each use of a name in the std namespace.

The Stream Insertion Operator and Escape Sequences

In the context of an output statement, the << operator is referred to as the **stream insertion operator**. When this program executes, the value to the operator's right, the right **operand**, is inserted in the output stream. Notice that the operator points in the direction of where the data goes. A string literal's characters *normally* print exactly as they appear between the double quotes. However, the characters \n are *not* printed on the screen (Fig. 2.1). The backslash (\) is called an **escape character**. It indicates that a "special" character is to be output. When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an **escape sequence**. The escape sequence \n means **newline**. It causes the screen cursor to move to the beginning of the next line on the screen. Some common escape sequences are listed in Fig. 2.2.

Escape sequence	Description
`\n`	Newline. Position the screen cursor to the beginning of the next line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\r`	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
`\a`	Alert. Sound the system bell.
`\\`	Backslash. Used to print a backslash character.
`\'`	Single quote. Used to print a single quote character.
`\"`	Double quote. Used to print a double quote character.

Fig. 2.2 | Escape sequences.

The `return` *Statement*

Line 10

```
return 0; // indicate that program ended successfully
```

is one of several means we'll use to **exit a function**. When the **`return` statement** is used at the end of `main`, as shown here, the value `0` indicates that the program has *terminated successfully*. The right brace, `}`, (line 11) indicates the end of function `main`. According to the C++ standard, if program execution reaches the end of `main` without encountering a `return` statement, it's assumed that the program terminated successfully—exactly as when the last statement in main is a `return` statement with the value `0`. For that reason, we *omit* the `return` statement at the end of `main` in subsequent programs.

2.3 Modifying Our First C++ Program

We now present two examples that modify the program of Fig. 2.1 to print text on one line by using multiple statements and to print text on several lines by using a single statement.

Printing a Single Line of Text with Multiple Statements

`Welcome to C++!` can be printed several ways. For example, Fig. 2.3 performs stream insertion in multiple statements (lines 8–9), yet produces the same output as the program of Fig. 2.1. [*Note:* From this point forward, we use a *light gray background* to highlight the key features each program introduces.] Each stream insertion resumes printing where the previous one stopped. The first stream insertion (line 8) prints `Welcome` followed by a space, and because this string did not end with `\n`, the second stream insertion (line 9) begins printing on the *same* line immediately following the space.

```
1  // Fig. 2.3: fig02_03.cpp
2  // Printing a line of text with multiple statements.
3  #include <iostream> // allows program to output data to the screen
4
```

Fig. 2.3 | Printing a line of text with multiple statements. (Part 1 of 2.)

```
// function main begins program execution
int main()
{
   std::cout << "Welcome ";
   std::cout << "to C++!\n";
} // end function main
```

```
Welcome to C++!
```

Fig. 2.3 | Printing a line of text with multiple statements. (Part 2 of 2.)

Printing Multiple Lines of Text with a Single Statement

A single statement can print multiple lines by using newline characters, as in line 8 of Fig. 2.4. Each time the `\n` (newline) escape sequence is encountered in the output stream, the screen cursor is positioned to the beginning of the next line. To get a blank line in your output, place two newline characters back to back, as in line 8.

```
1 // Fig. 2.4: fig02_04.cpp
2 // Printing multiple lines of text with a single statement.
3 #include <iostream> // allows program to output data to the screen
4
5 // function main begins program execution
6 int main()
7 {
8    std::cout << "Welcome\nto\n\nC++!\n";
9 } // end function main
```

```
Welcome
to

C++!
```

Fig. 2.4 | Printing multiple lines of text with a single statement.

2.4 Another C++ Program: Adding Integers

Our next program obtains two integers typed by a user at the keyboard, computes the sum of these values and outputs the result using `std::cout`. Figure 2.5 shows the program and sample inputs and outputs. In the sample execution, we highlight the user's input in bold. The program begins execution with function `main` (line 6). The left brace (line 7) begins `main`'s body and the corresponding right brace (line 22) ends it.

```
1 // Fig. 2.5: fig02_05.cpp
2 // Addition program that displays the sum of two integers.
3 #include <iostream> // allows program to perform input and output
```

Fig. 2.5 | Addition program that displays the sum of two integers. (Part 1 of 2.)

```
4
5   // function main begins program execution
6   int main()
7   {
8      // variable declarations
9      int number1 = 0; // first integer to add (initialized to 0)
10     int number2 = 0; // second integer to add (initialized to 0)
11     int sum = 0; // sum of number1 and number2 (initialized to 0)
12
13     std::cout << "Enter first integer: "; // prompt user for data
14     std::cin >> number1; // read first integer from user into number1
15
16     std::cout << "Enter second integer: "; // prompt user for data
17     std::cin >> number2; // read second integer from user into number2
18
19     sum = number1 + number2; // add the numbers; store result in sum
20
21     std::cout << "Sum is " << sum << std::endl; // display sum; end line
22  } // end function main
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 2.5 | Addition program that displays the sum of two integers. (Part 2 of 2.)

Variable Declarations

Lines 9–11

```
int number1 = 0; // first integer to add (initialized to 0)
int number2 = 0; // second integer to add (initialized to 0)
int sum = 0; // sum of number1 and number2 (initialized to 0)
```

are **declarations**. The identifiers `number1`, `number2` and `sum` are the names of **variables**. These declarations specify that the variables `number1`, `number2` and `sum` are data of type **`int`**, meaning that these variables will hold integer values. The declarations also initialize each of these variables to `0`.

> **Error-Prevention Tip 2.1**
>
> *Although it's not always necessary to initialize every variable explicitly, doing so will help you avoid many kinds of problems.*

All variables *must* be declared with a *name* and a *data type before* they can be used in a program. Several variables of the same type may be declared in one declaration or in multiple declarations. We could have declared all three variables in one declaration by using a **comma-separated list** as follows:

```
int number1 = 0, number2 = 0, sum = 0;
```

This makes the program less readable and prevents us from providing comments that describe each variable's purpose.

Good Programming Practice 2.3

Declare only one variable in each declaration and provide a comment that explains the variable's purpose in the program.

Fundamental Types

We'll soon discuss the type `double` for specifying *real numbers*, and the type `char` for specifying *character data*. Real numbers are numbers with decimal points, such as 3.4, 0.0 and –11.19. A `char` variable may hold only a single lowercase letter, a single uppercase letter, a single digit or a single special character (e.g., `$` or `*`). Types such as `int`, `double` and `char` are called **fundamental types**. Fundamental-type names consist of one or more *keywords* and therefore *must* appear in all lowercase letters. Appendix C contains the complete list of fundamental types.

Identifiers

A variable name (such as `number1`) is any valid **identifier** that is *not* a keyword. An identifier is a series of characters consisting of letters, digits and underscores (_) that does *not* begin with a digit. C++ is **case sensitive**—uppercase and lowercase letters are *different*, so `a1` and `A1` are *different* identifiers.

Portability Tip 2.1

C++ allows identifiers of any length, but your C++ implementation may restrict identifier lengths. Use identifiers of 31 characters or fewer to ensure portability.

Good Programming Practice 2.4

Choosing meaningful identifiers makes a program ***self-documenting****—a person can understand the program simply by reading it rather than having to refer to program comments or documentation.*

Good Programming Practice 2.5

Avoid using abbreviations in identifiers. This improves program readability.

Good Programming Practice 2.6

Do not use identifiers that begin with underscores and double underscores, because C++ compilers may use names like that for their own purposes internally. This will prevent the names you choose from being confused with names the compilers choose.

Placement of Variable Declarations

Declarations of variables can be placed almost anywhere in a program, but they *must* appear *before* their corresponding variables are used in the program. For example, in the program of Fig. 2.5, the declaration in line 9

```
int number1 = 0; // first integer to add (initialized to 0)
```

could have been placed immediately before line 14

```
std::cin >> number1; // read first integer from user into number1
```

Obtaining the First Value from the User

Line 13

```
    std::cout << "Enter first integer: "; // prompt user for data
```

displays `Enter first integer:` followed by a space. This message is called a **prompt** because it directs the user to take a specific action. We like to pronounce the preceding statement as "`std::cout` *gets* the string `"Enter first integer: "`." Line 14

```
    std::cin >> number1; // read first integer from user into number1
```

uses the **standard input stream object `cin`** (of namespace `std`) and the **stream extraction operator**, `>>`, to obtain a value from the keyboard. Using the stream extraction operator with `std::cin` takes character input from the standard input stream, which is usually the keyboard. We like to pronounce the preceding statement as, "`std::cin` *gives* a value to `number1`" or simply "`std::cin` *gives* `number1`."

When the computer executes the preceding statement, it waits for the user to enter a value for variable `number1`. The user responds by typing an integer (as characters), then pressing the *Enter* key (sometimes called the *Return* key) to send the characters to the computer. The computer converts the character representation of the number to an integer and assigns (i.e., copies) this number (or **value**) to the variable `number1`. Any subsequent references to `number1` in this program will use this same value.

The `std::cout` and `std::cin` stream objects facilitate interaction between the user and the computer.

Users can, of course, enter *invalid* data from the keyboard. For example, when your program is expecting the user to enter an integer, the user could enter alphabetic characters, special symbols (like # or @) or a number with a decimal point (like 73.5), among others. In these early programs, we assume that the user enters *valid* data. As you progress through the book, you'll learn various techniques for dealing with the broad range of possible data-entry problems.

Obtaining the Second Value from the User

Line 16

```
    std::cout << "Enter second integer: "; // prompt user for data
```

prints `Enter second integer:` on the screen, prompting the user to take action. Line 17

```
    std::cin >> number2; // read second integer from user into number2
```

obtains a value for variable `number2` from the user.

Calculating the Sum of the Values Input by the User

The assignment statement in line 19

```
    sum = number1 + number2; // add the numbers; store result in sum
```

adds the values of variables `number1` and `number2` and assigns the result to variable `sum` using the **assignment operator `=`**. We like to read this statement as, "`sum` *gets* the value of `number1 + number2`." Most calculations are performed in assignment statements. The `=` operator and the `+` operator are **binary operators**—each has *two* operands. In the case of the `+` operator, the two operands are `number1` and `number2`. In the case of the preceding `=` operator, the two operands are `sum` and the value of the expression `number1 + number2`.

Good Programming Practice 2.7

Place spaces on either side of a binary operator. This makes the operator stand out and makes the program more readable.

Displaying the Result

Line 21

```
std::cout << "Sum is " << sum << std::endl; // display sum; end
line
```

displays the character string `Sum is` followed by the numerical value of variable `sum` followed by `std::endl`—a **stream manipulator**. The name `endl` is an abbreviation for "end line" and belongs to namespace `std`. The `std::endl` stream manipulator outputs a newline, then "flushes the output buffer." This simply means that, on some systems where outputs accumulate in the machine until there are enough to "make it worthwhile" to display them on the screen, `std::endl` forces any accumulated outputs to be displayed at that moment. This can be important when the outputs are prompting the user for an action, such as entering data.

The preceding statement outputs multiple values of different types. The stream insertion operator "knows" how to output each type of data. Using multiple stream insertion operators (<<) in a single statement is referred to as **concatenating**, **chaining** or **cascading stream insertion operations**.

Calculations can also be performed in output statements. We could have combined the statements in lines 19 and 21 into the statement

```
std::cout << "Sum is " << number1 + number2 << std::endl;
```

thus eliminating the need for the variable `sum`.

A powerful feature of C++ is that you can create your own data types called classes (we introduce this capability in Chapter 3 and explore it in depth in Chapter 9). You can then "teach" C++ how to input and output values of these new data types using the `>>` and `<<` operators (this is called **operator overloading**—a topic we explore in Chapter 10).

2.5 Arithmetic

Most programs perform arithmetic calculations. Figure 2.6 summarizes the C++ **arithmetic operators**. The **asterisk** (*) indicates *multiplication* and the **percent sign** (**%**) is the *modulus operator* that will be discussed shortly. The arithmetic operators in Fig. 2.6 are all *binary* operators, i.e., operators that take two operands. For example, the expression `number1 + number2` contains the binary operator `+` and the two operands `number1` and `number2`.

Integer division (i.e., where both the numerator and the denominator are integers) yields an integer quotient; for example, the expression `7 / 4` evaluates to `1` and the expression `17 / 5` evaluates to `3`. *Any fractional part in integer division is* ***truncated*** *—no rounding occurs.*

C++ provides the **modulus operator**, `%`, that yields the *remainder after integer division*. The modulus operator can be used *only* with integer operands. The expression `x % y` yields the *remainder* after `x` is divided by `y`. Thus, `7 % 4` yields `3` and `17 % 5` yields `2`. In later chapters, we discuss many interesting applications of the modulus operator, such as determining whether one number is a *multiple* of another (a special case of this is determining whether a number is *odd* or *even*).

C++ operation	C++ arithmetic operator	Algebraic expression	C++ expression
Addition	`+`	$f + 7$	`f + 7`
Subtraction	`-`	$p - c$	`p - c`
Multiplication	`*`	bm or $b \cdot m$	`b * m`
Division	`/`	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Modulus	`%`	$r \bmod s$	`r % s`

Fig. 2.6 | Arithmetic operators.

Arithmetic Expressions in Straight-Line Form

Arithmetic expressions in C++ must be entered into the computer in **straight-line form**. Thus, expressions such as "`a` divided by `b`" must be written as `a / b`, so that all constants, variables and operators appear in a straight line. The algebraic notation

$$\frac{a}{b}$$

is generally *not* acceptable to compilers, although some special-purpose software packages do support more natural notation for complex mathematical expressions.

Parentheses for Grouping Subexpressions

Parentheses are used in C++ expressions in the same manner as in algebraic expressions. For example, to multiply `a` times the quantity `b + c` we write `a * ( b + c )`.

Rules of Operator Precedence

C++ applies the operators in arithmetic expressions in a precise order determined by the following **rules of operator precedence**, which are generally the same as those in algebra:

1. Operators in expressions contained within pairs of *parentheses* are evaluated first. Parentheses are at the highest level of precedence. In cases of **nested**, or **embedded**, **parentheses**, such as

```
( a * ( b + c ) )
```

   the operators in the *innermost* pair of parentheses are applied first.

2. Multiplication, division and modulus operations are applied next. If an expression contains several multiplication, division and modulus operations, operators are applied from *left to right*. Multiplication, division and modulus are on the *same* level of precedence.

3. Addition and subtraction operations are applied last. If an expression contains several addition and subtraction operations, operators are applied from *left to right*. Addition and subtraction also have the *same* level of precedence.

The rules of operator precedence define the order in which C++ applies operators. When we say that certain operators are applied from left to right, we are referring to the **associativity** of the operators. For example, the addition operators (+) in the expression

```
a + b + c
```

associate from left to right, so `a + b` is calculated first, then `c` is added to that sum to determine the whole expression's value. We'll see that some operators associate from *right to left*. Figure 2.7 summarizes these rules of operator precedence. We expand this table as we introduce additional C++ operators. Appendix A contains the complete precedence chart.

Operator(s)	Operation(s)	Order of evaluation (precedence)
`( )`	Parentheses	Evaluated first. If the parentheses are *nested*, such as in the expression `a * ( b + c / d + e ) )`, the expression in the *innermost* pair is evaluated first. [*Caution:* If you have an expression such as `(a + b) * (c - d)` in which two sets of parentheses are not nested, but appear "on the same level," the C++ Standard does *not* specify the order in which these parenthesized subexpressions will be evaluated.]

`*`
`/`
`%`
	Multiplication
	Division
	Modulus
		Evaluated second. If there are several, they're evaluated left to right.

`+` `-`	Addition Subtraction	Evaluated last. If there are several, they're evaluated left to right.

Fig. 2.7 | Precedence of arithmetic operators.

Sample Algebraic and C++ Expressions

Now consider several expressions in light of the rules of operator precedence. Each example lists an algebraic expression and its C++ equivalent. The following is an example of an arithmetic mean (average) of five terms:

Algebra: $m = \dfrac{a + b + c + d + e}{5}$

C++: `m = ( a + b + c + d + e ) / 5;`

The parentheses are required because division has *higher* precedence than addition. The *entire* quantity `( a + b + c + d + e )` is to be divided by `5`.

The following is an example of the equation of a straight line:

Algebra: $y = mx + b$

C++: `y = m * x + b;`

No parentheses are required. The multiplication is applied first because multiplication has a *higher* precedence than addition.

The following example contains modulus (%), multiplication, division, addition, subtraction and assignment operations:

Algebra: $z = pr\%q + w/x - y$

C++: `z = p * r % q + w / x - y;`

`=`	`*`	`%`	`+`	`/`	`-`
6	1	2	4	3	5

The circled numbers indicate the order in which C++ applies the operators. The multiplication, modulus and division are evaluated *first* in left-to-right order (i.e., they associate from left to right) because they have *higher precedence* than addition and subtraction. The addition and subtraction are applied next. These are also applied left to right. The assignment operator is applied *last* because its precedence is *lower* than that of any of the arithmetic operators.

Evaluation of a Second-Degree Polynomial

To develop a better understanding of the rules of operator precedence, consider the evaluation of a second-degree polynomial $y = ax^2 + bx + c$:

```
y = a * x * x + b * x + c;
  6   1   2   4   3   5
```

The circled numbers indicate the order in which C++ applies the operators. *There is no arithmetic operator for exponentiation in C++*, so we've represented x^2 as `x * x`. In Chapter 5, we'll discuss the standard library function `pow` ("power") that performs exponentiation.

Suppose variables `a`, `b`, `c` and `x` in the preceding second-degree polynomial are initialized as follows: `a = 2`, `b = 3`, `c = 7` and `x = 5`. Figure 2.8 illustrates the order in which the operators are applied and the final value of the expression.

Step 1.	`y = 2 * 5 * 5 + 3 * 5 + 7;` `2 * 5 is 10`	*(Leftmost multiplication)*
Step 2.	`y = 10 * 5 + 3 * 5 + 7;` `10 * 5 is 50`	*(Leftmost multiplication)*
Step 3.	`y = 50 + 3 * 5 + 7;` `3 * 5 is 15`	*(Multiplication before addition)*
Step 4.	`y = 50 + 15 + 7;` `50 + 15 is 65`	*(Leftmost addition)*
Step 5.	`y = 65 + 7;` `65 + 7 is 72`	*(Last addition)*
Step 6.	`y = 72`	*(Last operation—place `72` in `y`)*

Fig. 2.8 | Order in which a second-degree polynomial is evaluated.

Redundant Parentheses

As in algebra, it's acceptable to place *unnecessary* parentheses in an expression to make the expression clearer. These are called **redundant parentheses**. For example, the preceding assignment statement could be parenthesized as follows:

```
y = ( a * x * x ) + ( b * x ) + c;
```

2.6 Decision Making: Equality and Relational Operators

We now introduce a simple version of C++'s **`if` statement** that allows a program to take alternative action based on whether a **condition** is true or false. If the condition is *true*, the statement in the body of the `if` statement *is* executed. If the condition is *false*, the body statement *is not* executed. We'll see an example shortly.

Conditions in `if` statements can be formed by using the **relational operators** and **equality operators** summarized in Fig. 2.9. The relational operators all have the same level of precedence and associate left to right. The equality operators both have the same level of precedence, which is *lower* than that of the relational operators, and associate left to right.

Algebraic relational or equality operator	C++ relational or equality operator	Sample C++ condition	Meaning of C++ condition
Relational operators			
>	`>`	`x > y`	`x` is greater than `y`
<	`<`	`x < y`	`x` is less than `y`
≥	`>=`	`x >= y`	`x` is greater than or equal to `y`
≤	`<=`	`x <= y`	`x` is less than or equal to `y`
Equality operators			
=	`==`	`x == y`	`x` is equal to `y`
≠	`!=`	`x != y`	`x` is not equal to `y`

Fig. 2.9 | Relational and equality operators.

Common Programming Error 2.2

*Reversing the order of the pair of symbols in the operators `!=`, `>=` and `<=` (by writing them as `=!`, `=>` and `=<`, respectively) is normally a syntax error. In some cases, writing `!=` as `=!` will not be a syntax error, but almost certainly will be a **logic error** that has an effect at execution time. You'll understand why when you learn about logical operators in Chapter 5. A **fatal logic error** causes a program to fail and terminate prematurely. A **nonfatal logic error** allows a program to continue executing, but usually produces incorrect results.*

Common Programming Error 2.3

Confusing the equality operator `==` with the assignment operator `=` results in logic errors. We like to read the equality operator as "is equal to" or "double equals," and the assignment operator as "gets" or "gets the value of" or "is assigned the value of." As you'll see in Section 5.9, confusing these operators may not necessarily cause an easy-to-recognize syntax error, but may cause subtle logic errors.

Using the `if` Statement

The following example (Fig. 2.10) uses six `if` statements to compare two numbers input by the user. If the condition in any of these `if` statements is satisfied, the output statement associated with that `if` statement is executed.

```
// Fig. 2.13: fig02_13.cpp
// Comparing integers using if statements, relational operators
// and equality operators.
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl

// function main begins program execution
int main()
{
   int number1 = 0; // first integer to compare (initialized to 0)
   int number2 = 0; // second integer to compare (initialized to 0)

   cout << "Enter two integers to compare: "; // prompt user for data
   cin >> number1 >> number2; // read two integers from user

   if ( number1 == number2 )
      cout << number1 << " == " << number2 << endl;

   if ( number1 != number2 )
      cout << number1 << " != " << number2 << endl;

   if ( number1 < number2 )
      cout << number1 << " < " << number2 << endl;

   if ( number1 > number2 )
      cout << number1 << " > " << number2 << endl;

   if ( number1 <= number2 )
      cout << number1 << " <= " << number2 << endl;

   if ( number1 >= number2 )
      cout << number1 << " >= " << number2 << endl;
} // end function main
```

```
Enter two integers to compare: 3 7
3 != 7
3 < 7
3 <= 7
```

```
Enter two integers to compare: 22 12
22 != 12
22 > 12
22 >= 12
```

```
Enter two integers to compare: 7 7
7 == 7
7 <= 7
7 >= 7
```

Fig. 2.10 | Comparing integers using `if` statements, relational operators and equality operators.

`using` Declarations
Lines 6–8

```
using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl
```

are **`using` declarations** that eliminate the need to repeat the `std::` prefix as we did in earlier programs. We can now write `cout` instead of `std::cout`, `cin` instead of `std::cin` and `endl` instead of `std::endl`, respectively, in the remainder of the program.

In place of lines 6–8, many programmers prefer to provide the **`using` directive**

```
using namespace std;
```

which enables a program to use *all* the names in any standard C++ header (such as `<iostream>`) that a program might include. From this point forward in the book, we'll use the preceding directive in our programs. In Chapter 21, Other Topics, we'll discuss some issues with `using` directives in large-scale systems.

Variable Declarations and Reading the Inputs from the User
Lines 13–14

```
int number1 = 0; // first integer to compare (initialized to 0)
int number2 = 0; // second integer to compare (initialized to 0)
```

declare the variables used in the program and initializes them to `0`.

The program uses cascaded stream extraction operations (line 17) to input two integers. Remember that we're allowed to write `cin` (instead of `std::cin`) because of line 7. First a value is read into variable `number1`, then a value is read into variable `number2`.

Comparing Numbers
The `if` statement in lines 19–20

```
if ( number1 == number2 )
   cout << number1 << " == " << number2 << endl;
```

compares the values of variables `number1` and `number2` to test for equality. If the values are equal, the statement in line 20 displays a line of text indicating that the numbers are equal. If the conditions are `true` in one or more of the `if` statements starting in lines 22, 25, 28, 31 and 34, the corresponding body statement displays an appropriate line of text.

Each `if` statement in Fig. 2.10 has a single statement in its body and each body statement is indented. In Chapter 4 we show how to specify `if` statements with multiple-statement bodies (by enclosing the body statements in a pair of braces, `{ }`, creating what's called a **compound statement** or a **block**).

Common Programming Error 2.4

Placing a semicolon immediately after the right parenthesis after the condition in an `if` statement is often a logic error (although not a syntax error). The semicolon causes the body of the `if` statement to be empty, so the `if` statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the `if` statement now becomes a statement in sequence *with the `if` statement and* always *executes, often causing the program to produce incorrect results.*

White Space

Recall that whitespace characters, such as tabs, newlines and spaces, are normally ignored by the compiler. So, statements may be split over several lines and may be spaced according to your preferences. It's a syntax error to split identifiers, strings (such as `"hello"`) and constants (such as the number `1000`) over several lines.

Good Programming Practice 2.8

A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose meaningful breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines and left-align the group of indented lines.

Operator Precedence

Figure 2.11 shows the precedence and associativity of the operators introduced in this chapter. The operators are shown top to bottom in decreasing order of precedence. All these operators, with the exception of the assignment operator `=`, associate from left to right. Addition is left-associative, so an expression like `x + y + z` is evaluated as if it had been written `(x + y) + z`. The assignment operator `=` associates from *right to left*, so an expression such as `x = y = 0` is evaluated as if it had been written `x = (y = 0)`, which, as we'll soon see, first assigns `0` to `y`, then assigns the *result* of that assignment—`0`—to `x`.

Operators				Associativity	Type
`()`				*[See caution in Fig. 2.7]*	grouping parentheses
`*`	`/`	`%`		left to right	multiplicative
`+`	`-`			left to right	additive
`<<`	`>>`			left to right	stream insertion/extraction
`<`	`<=`	`>`	`>=`	left to right	relational
`==`	`!=`			left to right	equality
`=`				right to left	assignment

Fig. 2.11 | Precedence and associativity of the operators discussed so far.

Good Programming Practice 2.9

Refer to the operator precedence and associativity chart (Appendix A) when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you're uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you'd do in an algebraic expression. Be sure to observe that some operators such as assignment (=) associate right to left rather than left to right.

2.7 Wrap-Up

You learned many important basic features of C++ in this chapter, including displaying data on the screen, inputting data from the keyboard and declaring variables of fundamental types. In particular, you learned to use the output stream object `cout` and the input stream object `cin` to build simple interactive programs. We explained how variables are stored in and retrieved from memory. You also learned how to use arithmetic operators to perform calculations. We discussed the order in which C++ applies operators (i.e., the rules of operator precedence), as well as the associativity of the operators. You also learned how C++'s `if` statement allows a program to make decisions. Finally, we introduced the equality and relational operators, which you use to form conditions in `if` statements.

The non-object-oriented applications presented here introduced you to basic programming concepts. As you'll see in Chapter 3, C++ applications typically contain just a few lines of code in function `main`—these statements normally create the objects that perform the work of the application, then the objects "take over from there." In Chapter 3, you'll learn how to implement your own classes and use objects of those classes in applications.

This page intentionally left blank

Index

Symbols

Numerics

A

E

G

H

O

P

Q

R