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®

Pearson

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.]

This page intentionally left blank

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
```

This page intentionally left blank

1

Introduction

Objectives

In this chapter you'll:

- Review object-technology concepts.
- Learn the elements of a typical C++ program-development environment.
- Test-drive a C++ application.

Outline

1.1 Introduction

Welcome to C++—a powerful computer programming language that's appropriate for technically oriented people with little or no programming experience, and for experienced programmers to use in building substantial information systems.

You'll learn *object-oriented programming in C++*. You'll create many *C++ software objects* that model *things* in the real-world.

C++ is one of today's most popular software development languages. This text provides an introduction to programming in C++11—the latest version standardized through the **International Organization for Standardization** (**ISO**) and the **International Electrotechnical Commission** (IEC).

1.2 C++

C++ evolved from C, which was developed by Dennis Ritchie at Bell Laboratories. C is available for most computers and is hardware independent. With careful design, it's possible to write C programs that are **portable** to most computers.

The widespread use of C with various kinds of computers (sometimes called **hardware platforms**) unfortunately led to many variations. A standard version of C was needed. The American National Standards Institute (ANSI) cooperated with the International Organization for Standardization (ISO) to standardize C worldwide; the joint standard document was published in 1990 and is referred to as *ANSI/ISO 9899: 1990*.

C11 is the latest ANSI standard for the C programming language. It was developed to evolve the C language to keep pace with increasingly powerful hardware and ever more demanding user requirements. C11 also makes C more consistent with C++. For more information on C and C11, see our book *C How to Program, 7/e* and our C Resource Center (located at `www.deitel.com/C`).

C++, an extension of C, was developed by Bjarne Stroustrup in 1979 at Bell Laboratories. Originally called "C with Classes", it was renamed to C++ in the early 1980s. C++ provides a number of features that "spruce up" the C language, but more importantly, it provides capabilities for object-oriented programming.

You'll begin developing customized, reusable classes and objects in Chapter 3, Introduction to Classes, Objects and Strings. The book is object oriented, where appropriate, from the start and throughout the text.

We also provide an *optional* automated teller machine (ATM) case study in Chapters 22–23, which contains a complete C++ implementation. The case study presents a carefully paced introduction to object-oriented design using the UML—an industry-standard graphical modeling language for developing object-oriented systems. We guide you through a friendly design experience intended for the novice.

C++ Standard Library

C++ programs consist of pieces called **classes** and **functions.** You can program each piece yourself, but most C++ programmers take advantage of the rich collections of classes and functions in the **C++ Standard Library**. Thus, there are really two parts to learning the C++ "world." The first is learning the C++ language itself; the second is learning how to use the classes and functions in the C++ Standard Library. We discuss many of these classes and functions. Most compiler vendors provide online C++ Standard Library reference documentation. You can also learn about the C++ Standard library at:

```
www.cppreference.com
```

In addition to the C++ Standard Library, many special-purpose class libraries are supplied by independent software vendors and by the open-source community.

Software Engineering Observation 1.1

Use a "building-block" approach to create programs. Avoid reinventing the wheel. Use existing pieces wherever possible. Called ***software reuse****, this practice is central to object-oriented programming.*

Software Engineering Observation 1.2

When programming in C++, you typically will use the following building blocks: classes and functions from the C++ Standard Library, classes and functions you and your colleagues create and classes and functions from various popular third-party libraries.

The advantage of creating your own functions and classes is that you'll know exactly how they work. You'll be able to examine the C++ code. The disadvantage is the time-consuming and complex effort that goes into designing, developing and maintaining new functions and classes that are correct and that operate efficiently.

Performance Tip 1.1

Using C++ Standard Library functions and classes instead of writing your own versions can improve program performance, because they're written carefully to perform efficiently. This technique also shortens program development time.

Portability Tip 1.1

Using C++ Standard Library functions and classes instead of writing your own improves program portability, because they're included in every C++ implementation.

1.3 Object Technology

Building software quickly, correctly and economically remains an elusive goal at a time when demands for new and more powerful software are soaring. *Objects*, or more precisely—as we'll see in Chapter 3—the *classes* objects come from, are essentially *reusable* software components. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any *noun* can be reasonably represented as a software object in terms of *attributes* (e.g., name, color and size) and *behaviors* (e.g., calculating, moving and communicating). Software developers have discovered that using a modular, object-oriented design-and-implementation approach can make software-development groups much more productive than was possible with earlier techniques—object-oriented programs are often easier to understand, correct and modify.

The Automobile as an Object

Let's begin with a simple analogy. Suppose you want to *drive a car and make it go faster by pressing its accelerator pedal*. What must happen before you can do this? Well, before you can drive a car, someone has to *design* it. A car typically begins as engineering drawings, similar to the *blueprints* that describe the design of a house. These drawings include the design for an accelerator pedal. The pedal *hides* from the driver the complex mechanisms that actually make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel *hides* the mechanisms that turn the car. This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Before you can drive a car, it must be *built* from the engineering drawings that describe it. A completed car has an *actual* accelerator pedal to make the car go faster, but even that's not enough—the car won't accelerate on its own (hopefully!), so the driver must *press* the pedal to accelerate the car.

Member Functions and Classes

Let's use our car example to introduce some key object-oriented programming concepts. Performing a task in a program requires a **member function**. The member function houses the program statements that actually perform its task. It hides these statements from its user, just as the accelerator pedal of a car hides from the driver the mechanisms of making the car go faster. In C++, we create a program unit called a **class** to house the set of member functions that perform the class's tasks. For example, a class that represents a bank account might contain one member function to *deposit* money to an account, another to *withdraw* money from an account and a third to *inquire* what the account's current balance is. A class is similar in concept to a car's engineering drawings, which house the design of an accelerator pedal, steering wheel, and so on.

Instantiation

Just as someone has to *build a car* from its engineering drawings before you can actually drive a car, you must *build an object* from a class before a program can perform the tasks that the class's methods define. The process of doing this is called *instantiation*. An object is then referred to as an **instance** of its class.

Reuse

Just as a car's engineering drawings can be *reused* many times to build many cars, you can *reuse* a class many times to build many objects. Reuse of existing classes when building new classes and programs saves time and effort. Reuse also helps you build more reliable and effective systems, because existing classes and components often have gone through extensive *testing*, *debugging* and *performance tuning*. Just as the notion of *interchangeable parts* was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that has been spurred by object technology.

Messages and Member Function Calls

When you drive a car, pressing its gas pedal sends a *message* to the car to perform a task—that is, to go faster. Similarly, you *send messages to an object*. Each message is implemented as a **member function call** that tells a member function of the object to perform its task. For example, a program might call a particular bank account object's *deposit* member function to increase the account's balance.

Attributes and Data Members

A car, besides having capabilities to accomplish tasks, also has *attributes*, such as its color, its number of doors, the amount of gas in its tank, its current speed and its record of total miles driven (i.e., its odometer reading). Like its capabilities, the car's attributes are represented as part of its design in its engineering diagrams (which, for example, include an odometer and a fuel gauge). As you drive an actual car, these attributes are carried along with the car. Every car maintains its *own* attributes. For example, each car knows how much gas is in its own gas tank, but *not* how much is in the tanks of *other* cars.

An object, similarly, has attributes that it carries along as it's used in a program. These attributes are specified as part of the object's class. For example, a bank account object has a *balance attribute* that represents the amount of money in the account. Each bank account object knows the balance in the account it represents, but *not* the balances of the *other* accounts in the bank. Attributes are specified by the class's **data members**.

Encapsulation

Classes **encapsulate** (i.e., wrap) attributes and member functions into objects—an object's attributes and member functions are intimately related. Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented—implementation details are *hidden* within the objects themselves. This **information hiding**, as we'll see, is crucial to good software engineering.

Inheritance

A new class of objects can be created quickly and conveniently by **inheritance**—the new class absorbs the characteristics of an existing class, possibly customizing them and adding unique characteristics of its own. In our car analogy, an object of class "convertible" certainly *is an* object of the more *general* class "automobile," but more *specifically*, the roof can be raised or lowered.

Object-Oriented Analysis and Design (OOAD)

Soon you'll be writing programs in C++. How will you create the **code** (i.e., the program instructions) for your programs? Perhaps, like many programmers, you'll simply turn on your computer and start typing. This approach may work for small programs (like the ones we present in the early chapters of the book), but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of thousands of software developers building the next U.S. air traffic control system? For projects so large and complex, you should not simply sit down and start writing programs.

To create the best solutions, you should follow a detailed **analysis** process for determining your project's **requirements** (i.e., defining *what* the system is supposed to do) and developing a **design** that satisfies them (i.e., deciding *how* the system should do it). Ideally, you'd go through this process and carefully review the design (and have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it's called an **object-oriented analysis and design (OOAD) process**. Languages like C++ are object oriented. Programming in such a language, called **object-oriented programming (OOP)**, allows you to implement an object-oriented design as a working system.

The UML (Unified Modeling Language)

Although many different OOAD processes exist, a single graphical language for communicating the results of *any* OOAD process has come into wide use. This language, known as the Unified Modeling Language (UML), is now the most widely used graphical scheme for modeling object-oriented systems. We present our first UML diagrams in Chapters 3 and 4, then use them in our deeper treatment of object-oriented programming through Chapter 12. In our *optional* ATM Software Engineering Case Study in Chapters 22–23 we present a simple subset of the UML's features as we guide you through an object-oriented design experience.

1.4 Typical C++ Development Environment

C++ systems generally consist of three parts: a program development environment, the language and the C++ Standard Library. C++ programs typically go through six phases: edit, preprocess, compile, link, load and execute. The following discussion explains a typical C++ program development environment.

Phase 1: Editing a Program

Phase 1 consists of editing a file with an *editor program*, normally known simply as an *editor* (Fig. 1.1). You type a C++ program (typically referred to as **source code**) using the editor, make any necessary corrections and save the program on a secondary storage device, such as your hard drive. C++ source code filenames often end with the `.cpp`, `.cxx`, `.cc` or `.C` extensions (note that `C` is in uppercase) which indicate that a file contains C++ source code. See the documentation for your C++ compiler for more information on file-name extensions.

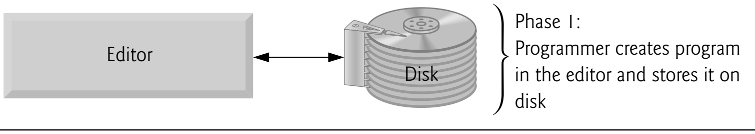

Fig. 1.1 | Typical C++ development environment—editing phase.

Two editors widely used on Linux systems are `vi` and `emacs`. C++ software packages for Microsoft Windows such as Microsoft Visual C++ (`microsoft.com/express`) have editors integrated into the programming environment. You can also use a simple text editor, such as Notepad in Windows, to write your C++ code.

For organizations that develop substantial information systems, **integrated development environments (IDEs)** are available from many major software suppliers. IDEs provide tools that support the software-development process, including editors for writing and editing programs and debuggers for locating **logic errors**—errors that cause programs to execute incorrectly. Popular IDEs include Microsoft® Visual Studio 2012 Express Edition, Dev C++, NetBeans, Eclipse, Apple's Xcode and CodeLite.

Phase 2: Preprocessing a C++ Program

In Phase 2, you give the command to **compile** the program (Fig. 1.2). In a C++ system, a **preprocessor** program executes automatically before the compiler's translation phase begins (so we call preprocessing Phase 2 and compiling Phase 3). The C++ preprocessor obeys commands called **preprocessing directives,** which indicate that certain manipulations are to be performed on the program before compilation. These manipulations usually include other text files to be compiled, and perform various text replacements. The most common preprocessing directives are discussed in the early chapters; a detailed discussion of preprocessor features appears in Appendix E, Preprocessor.

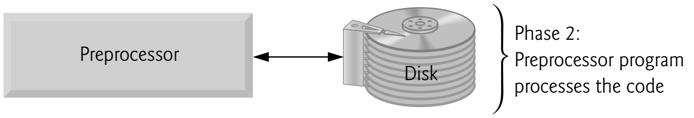

Fig. 1.2 | Typical C++ development environment—preprocessor phase.

Phase 3: Compiling a C++ Program

In Phase 3, the compiler translates the C++ program into machine-language code—also referred to as object code (Fig. 1.3).

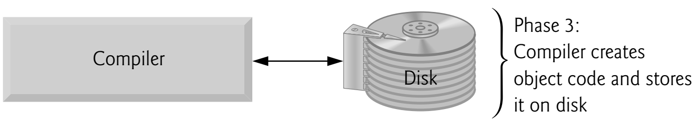

Fig. 1.3 | Typical C++ development environment—compilation phase.

Phase 4: Linking

Phase 4 is called **linking.** C++ programs typically contain references to functions and data defined elsewhere, such as in the standard libraries or in the private libraries of groups of programmers working on a particular project (Fig. 1.4). The object code produced by the C++ compiler typically contains "holes" due to these missing parts. A **linker** links the object code with the code for the missing functions to produce an **executable program** (with no missing pieces). If the program compiles and links correctly, an executable image is produced.

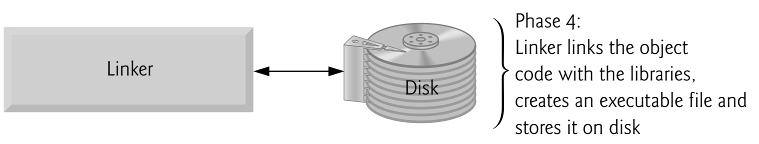

Fig. 1.4 | Typical C++ development environment—linking phase.

Phase 5: Loading
Phase 5 is called **loading.** Before a program can be executed, it must first be placed in memory (Fig. 1.5). This is done by the **loader,** which takes the executable image from disk and transfers it to memory. Additional components from shared libraries that support the program are also loaded.

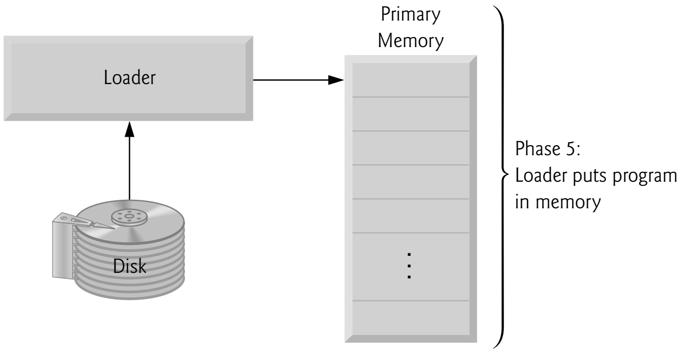

Fig. 1.5 | Typical C++ development environment—loading phase.

Phase 6: Execution
Finally, the computer, under the control of its CPU, **executes** the program one instruction at a time (Fig. 1.6). Some modern computer architectures can execute several instructions in parallel.

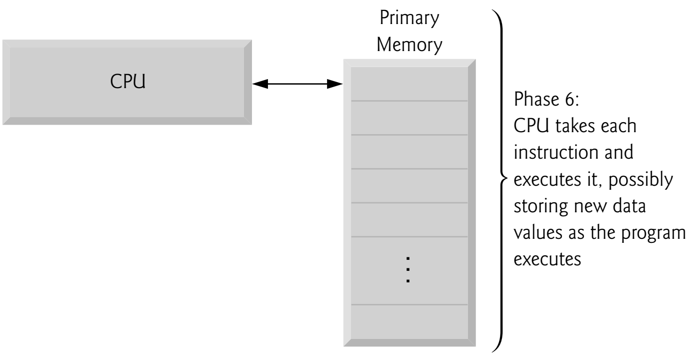

Fig. 1.6 | Typical C++ development environment—execution phase.

Problems That May Occur at Execution Time
Programs might not work on the first try. Each of the preceding phases can fail because of various errors that we'll discuss throughout this book. For example, an executing program

might try to divide by zero (an illegal operation for integer arithmetic in C++). This would cause the C++ program to display an error message. If this occurred, you'd have to return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine that the corrections fixed the problem(s). [*Note:* Most programs in C++ input or output data. Certain C++ functions take their input from `cin` (the **standard input stream**; pronounced "see-in"), which is normally the keyboard, but `cin` can be redirected to another device. Data is often output to `cout` (the **standard output stream**; pronounced "see-out"), which is normally the computer screen, but `cout` can be redirected to another device. When we say that a program prints a result, we normally mean that the result is displayed on a screen. Data may be output to other devices, such as disks and hardcopy printers. There is also a **standard error stream** referred to as **cerr.** The `cerr` stream (normally connected to the screen) is used for displaying error messages.

Common Programming Error 1.1

Errors such as division by zero occur as a program runs, so they're called ***runtime errors*** *or* ***execution-time errors. Fatal runtime errors*** *cause programs to terminate immediately without having successfully performed their jobs.* ***Nonfatal runtime errors*** *allow programs to run to completion, often producing incorrect results.*

1.5 Test-Driving a C++ Application

In this section, you'll run and interact with your first C++ application. You'll begin by running an entertaining guess-the-number game, which picks a number from 1 to 1000 and prompts you to guess it. If your guess is correct, the game ends. If your guess is not correct, the application indicates whether your guess is higher or lower than the correct number. There is no limit on the number of guesses you can make. [*Note:* Normally this application randomly selects the correct answer as you execute the program. This test-drive version of the application uses the same correct answer every time the program executes (though this may vary by compiler), so you can use the *same* guesses we use in this section and see the *same* results as we walk you through interacting with your first C++ application.]

We'll demonstrate running a C++ application using the Windows **Command Prompt** and a shell on Linux. The application runs similarly on both platforms. Many development environments are available in which you can compile, build and run C++ applications, such as GNU™ C++, Microsoft® Visual C++®, Apple® Xcode®, NetBeans®, Eclipse™, etc.

We use fonts to distinguish between features you see on the screen (e.g., the **Command Prompt**) and elements that are not directly related to the screen. We emphasize screen features like titles and menus (e.g., the **File** menu) in a semibold **sans-serif Helvetica** font and emphasize filenames, text displayed by an application and values you should enter into an application (e.g., `GuessNumber` or `500`) in a `sans-serif Lucida` font. As you've noticed, the **defining occurrence** of each term is set in bold type. For the figures in this section, we point out significant parts of the application. To make these features more visible, we've modified the background color of the **Command Prompt** window (for the Windows test drive only). To modify the **Command Prompt** colors on your system, open a **Command Prompt** by selecting **Start > All Programs > Accessories > Command Prompt**, then right click the title bar and select **Properties**. In the **"Command Prompt" Properties** dialog box that appears, click the **Colors** tab, and select your preferred text and background colors.

Running a C++ Application from the Windows Command Prompt

1. ***Checking your setup.*** It's important to read the Before You Begin section at `www.deitel.com/books/cpp11fp/` to make sure that you've copied the book's examples to your hard drive correctly.
2. ***Locating the completed application.*** Open a **Command Prompt** window. To change to the directory for the completed **GuessNumber** application, type **`cd C:\examples\ch01\GuessNumber\Windows`**, then press *Enter* (Fig. 1.7). The command `cd` is used to change directories.

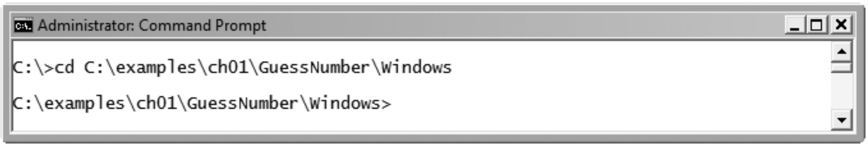

Fig. 1.7 | Opening a **Command Prompt** window and changing the directory.

3. ***Running the* GuessNumber *application.*** Now that you are in the directory that contains the **GuessNumber** application, type the command **`GuessNumber`** (Fig. 1.8) and press *Enter*. [*Note:* `GuessNumber.exe` is the actual name of the application; however, Windows assumes the `.exe` extension by default.]

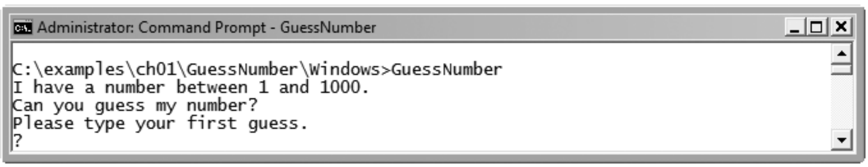

Fig. 1.8 | Running the **GuessNumber** application.

4. ***Entering your first guess.*** The application displays `"Please type your first guess."`, then displays a question mark (`?`) as a prompt on the next line (Fig. 1.8). At the prompt, enter **`500`** (Fig. 1.9).

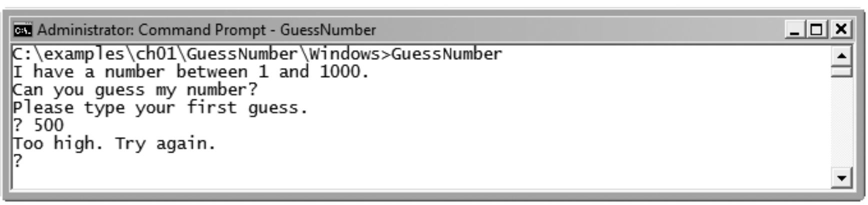

Fig. 1.9 | Entering your first guess.

5. ***Entering another guess.*** The application displays `"Too high. Try again."`, meaning that the value you entered is greater than the number the application chose as

the correct guess. So, you should enter a lower number for your next guess. At the prompt, enter **`250`** (Fig. 1.10). The application again displays `"Too high. Try again."`, because the value you entered is still greater than the number that the application chose as the correct guess.

```
Administrator: Command Prompt - GuessNumber
C:\examples\ch01\GuessNumber\Windows>GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
?
```

Fig. 1.10 | Entering a second guess and receiving feedback.

6. ***Entering additional guesses.*** Continue to play the game by entering values until you guess the correct number. The application will display `"Excellent! You guessed the number!"` (Fig. 1.11).

```
Administrator: Command Prompt - GuessNumber
Too high. Try again.
? 125
Too low. Try again.
? 187
Too high. Try again.
? 156
Too high. Try again.
? 140
Too high. Try again.
? 132
Too high. Try again.
? 128
Too low. Try again.
? 130
Too low. Try again.
? 131

Excellent! You guessed the number!
Would you like to play again (y or n)?
```

Fig. 1.11 | Entering additional guesses and guessing the correct number.

7. ***Playing the game again or exiting the application.*** After you guess correctly, the application asks if you'd like to play another game (Fig. 1.11). At the `"Would you like to play again (y or n)?"` prompt, entering the one character **`y`** causes the application to choose a new number and displays the message `"Please type your first guess."` followed by a question mark prompt (Fig. 1.12) so you can make your first guess in the new game. Entering the character **`n`** ends the application and returns you to the application's directory at the **Command Prompt** (Fig. 1.13). Each time you execute this application from the beginning (i.e., *Step 3*), it will choose the same numbers for you to guess.

8. *Close the* **Command Prompt** *window.*

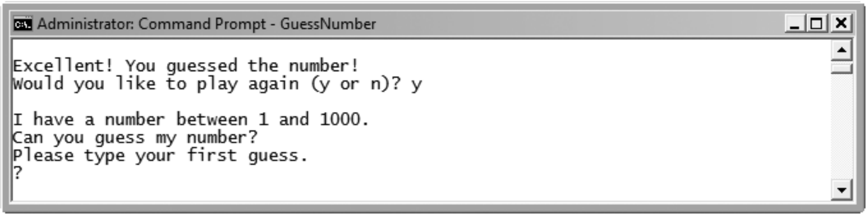

Fig. 1.12 | Playing the game again.

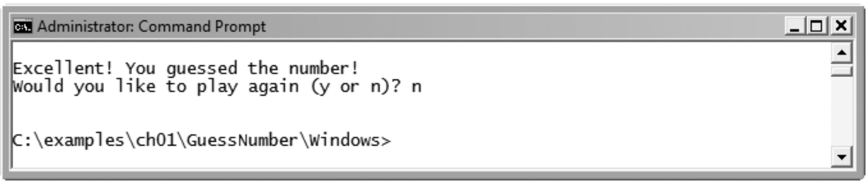

Fig. 1.13 | Exiting the game.

Running a C++ Application Using GNU C++ with Linux

For this test drive, we assume that you know how to copy the examples into your home directory. Also, for the figures in this section, we use a bold highlight to point out the user input required by each step. The prompt in the shell on our system uses the tilde (~) character to represent the home directory, and each prompt ends with the dollar sign ($) character. The prompt will vary among Linux systems.

1. *Locating the completed application.* From a Linux shell, change to the completed **GuessNumber** application directory (Fig. 1.14) by typing

```
cd Examples/ch01/GuessNumber/GNU_Linux
```

then pressing *Enter*. The command `cd` is used to change directories.

```
~$ cd examples/ch01/GuessNumber/GNU_Linux
~/examples/ch01/GuessNumber/GNU_Linux$
```

Fig. 1.14 | Changing to the **GuessNumber** application's directory.

2. *Compiling the* **GuessNumber** *application.* To run an application on the GNU C++ compiler, you must first compile it by typing

```
g++ GuessNumber.cpp -o GuessNumber
```

as in Fig. 1.15. This command compiles the application and produces an executable file called `GuessNumber`.

```
~/examples/ch01/GuessNumber/GNU_Linux$ g++ GuessNumber.cpp -o GuessNumber
~/examples/ch01/GuessNumber/GNU_Linux$
```

Fig. 1.15 | Compiling the **GuessNumber** application using the g++ command.

3. ***Running the* GuessNumber *application.*** To run the executable file `GuessNumber`, type `./GuessNumber` at the next prompt, then press *Enter* (Fig. 1.16).

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Fig. 1.16 | Running the **GuessNumber** application.

4. ***Entering your first guess.*** The application displays `"Please type your first guess."`, then displays a question mark (`?`) as a prompt on the next line (Fig. 1.16). At the prompt, enter **`500`** (Fig. 1.17). [*Note:* This is the same application that we modified and test-drove for Windows, but the outputs could vary based on the compiler being used.]

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

Fig. 1.17 | Entering an initial guess.

5. ***Entering another guess.*** The application displays `"Too high. Try again."`, meaning that the value you entered is greater than the number the application chose as the correct guess (Fig. 1.17). At the next prompt, enter **`250`** (Fig. 1.18). This time the application displays `"Too low. Try again."`, because the value you entered is less than the correct guess.

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too low. Try again.
?
```

Fig. 1.18 | Entering a second guess and receiving feedback.

6. ***Entering additional guesses.*** Continue to play the game (Fig. 1.19) by entering values until you guess the correct number. When you guess correctly, the application displays "`Excellent! You guessed the number.`"

```
Too low. Try again.
? 375
Too low. Try again.
? 437
Too high. Try again.
? 406
Too high. Try again.
? 391
Too high. Try again.
? 383
Too low. Try again.
? 387
Too high. Try again.
? 385
Too high. Try again.
? 384
Excellent! You guessed the number.
Would you like to play again (y or n)?
```

Fig. 1.19 | Entering additional guesses and guessing the correct number.

7. ***Playing the game again or exiting the application.*** After you guess the correct number, the application asks if you'd like to play another game. At the "`Would you like to play again (y or n)?`" prompt, entering the one character **`y`** causes the application to choose a new number and displays the message "`Please type your first guess.`" followed by a question mark prompt (Fig. 1.20) so you can make your first guess in the new game. Entering the character **`n`** ends the application and returns you to the application's directory in the shell (Fig. 1.21). Each time you execute this application from the beginning (i.e., *Step 3*), it will choose the same numbers for you to guess.

```
Excellent! You guessed the number.
Would you like to play again (y or n)? y

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Fig. 1.20 | Playing the game again.

```
Excellent! You guessed the number.
Would you like to play again (y or n)? n

~/examples/ch01/GuessNumber/GNU_Linux$
```

Fig. 1.21 | Exiting the game.

1.6 Operating Systems

Popular desktop operating systems include Linux, Windows and OS X (formerly called Mac OS X)—we used all three in developing this book. Popular mobile operating systems used in smartphones and tablets include Google's Android, Apple's iOS (for iPhone, iPad and iPod Touch devices), BlackBerry OS and Windows Phone. You can develop applications in C++ for all of the following key operating systems, including several of the latest mobile operating systems.

1.6.1 Windows—A Proprietary Operating System

In the mid-1980s, Microsoft developed the **Windows operating system**, consisting of a graphical user interface built on top of DOS—an enormously popular personal-computer operating system that users interacted with by *typing* commands. Windows borrowed from many concepts (such as icons, menus and windows) developed by Xerox PARC and popularized by early Apple Macintosh operating systems. Windows 8 is Microsoft's latest operating system—its features include enhancements to the user interface, faster startup times, further refinement of security features, touch-screen and multitouch support, and more. Windows is a *proprietary* operating system—it's controlled by Microsoft exclusively. Windows is by far the world's most widely used desktop operating system.

1.6.2 Linux—An Open-Source Operating System

The Linux operating system is perhaps the greatest success of the *open-source* movement. **Open-source software** departs from the *proprietary* software development style that dominated software's early years. With open-source development, individuals and companies *contribute* their efforts in developing, maintaining and evolving software in exchange for the right to use that software for their own purposes, typically at *no charge*. Open-source code is often scrutinized by a much larger audience than proprietary software, so errors often get removed faster. Open source also encourages innovation. Enterprise systems companies, such as IBM, Oracle and many others, have made significant investments in Linux open-source development.

Some key organizations in the open-source community are the Eclipse Foundation (the Eclipse Integrated Development Environment helps programmers conveniently develop software), the Mozilla Foundation (creators of the Firefox web browser), the Apache Software Foundation (creators of the Apache web server used to develop web-based applications) and SourceForge (which provides tools for managing open-source projects—it has hundreds of thousands of them under development). Rapid improvements to computing and communications, decreasing costs and open-source software have made it much easier and more economical to create a software-based business now than just a decade ago. A great example is Facebook, which was launched from a college dorm room and built with open-source software.

The **Linux** kernel is the core of the most popular open-source, freely distributed, full-featured operating system. It's developed by a loosely organized team of volunteers and is popular in servers, personal computers and embedded systems. Unlike that of proprietary operating systems like Microsoft's Windows and Apple's OS X, Linux source code (the program code) is available to the public for examination and modification and is free to download and install. As a result, Linux users benefit from a community of developers actively debugging and improving the kernel, and the ability to customize the operating system to meet specific needs.

A variety of issues—such as Microsoft's market power, the small number of user-friendly Linux applications and the diversity of Linux distributions, such as Red Hat Linux, Ubuntu Linux and many others—have prevented widespread Linux use on desktop computers. Linux has become extremely popular on servers and in embedded systems, such as Google's Android-based smartphones.

1.6.3 Apple's OS X; Apple's iOS for iPhone®, iPad® and iPod Touch® Devices

Apple, founded in 1976 by Steve Jobs and Steve Wozniak, quickly became a leader in personal computing. In 1979, Jobs and several Apple employees visited Xerox PARC (Palo Alto Research Center) to learn about Xerox's desktop computer that featured a graphical user interface (GUI). That GUI served as the inspiration for the Apple Macintosh, launched with much fanfare in a memorable Super Bowl ad in 1984.

The Objective-C programming language, created by Brad Cox and Tom Love at Stepstone in the early 1980s, added capabilities for object-oriented programming (OOP) to the C programming language. At the time of this writing, Objective-C was comparable in popularity to C++.[1] Steve Jobs left Apple in 1985 and founded NeXT Inc. In 1988, NeXT licensed Objective-C from StepStone and developed an Objective-C compiler and libraries which were used as the platform for the NeXTSTEP operating system's user interface and Interface Builder—used to construct graphical user interfaces.

Jobs returned to Apple in 1996 when Apple bought NeXT. Apple's OS X operating system is a descendant of NeXTSTEP. Apple's proprietary operating system, **iOS**, is derived from Apple's OS X and is used in the iPhone, iPad and iPod Touch devices.

1.6.4 Google's Android

Android—the fastest growing mobile and smartphone operating system—is based on the Linux kernel and Java. Experienced Java programmers can quickly dive into Android development. One benefit of developing Android apps is the openness of the platform. The operating system is open source and free.

The Android operating system was developed by Android, Inc., which was acquired by Google in 2005. In 2007, the Open Handset Alliance™—a consortium of 34 companies initially and 84 by 2011—was formed to continue developing Android. As of June 2012, more than 900,000 Android devices were being activated each day![2] Android smartphones are now outselling iPhones in the United States.[3] The Android operating system is used in numerous smartphones (such as the Motorola Droid, HTC One S, Samsung Galaxy Nexus and many more), e-reader devices (such as the Kindle Fire and Barnes and Noble Nook™), tablet computers (such as the Dell Streak and the Samsung Galaxy Tab), in-store touch-screen kiosks, cars, robots, multimedia players and more.

---

1. www.tiobe.com/index.php/content/paperinfo/tpci/index.html.
2. mashable.com/2012/06/11/900000-android-devices/.
3. www.pcworld.com/article/196035/android_outsells_the_iphone_no_big_surprise.html.

1.7 C++11 and the Open Source Boost Libraries

C++11 (formerly called C++0x)—the latest C++ programming language standard—was published by ISO/IEC in 2011. Bjarne Stroustrup, the creator of C++, expressed his vision for the future of the language—the main goals were to make C++ easier to learn, improve library building capabilities and increase compatibility with the C programming language. The new standard extends the C++ Standard Library and includes several features and enhancements to improve performance and security. The major C++ compiler vendors have already implemented many of the new C++11 features (Fig. 1.22). Throughout the book, we discuss various key features of C++11. For more information, visit the C++ Standards Committee website at `www.open-std.org/jtc1/sc22/wg21/` and `isocpp.org`. Copies of the C++11 language specification (ISO/IEC 14882:2011) can be purchased at:

```
http://bit.ly/CPlusPlus11Standard
```

C++ Compiler	URL of C++11 feature descriptions
C++11 features implemented in each of the major C++ compilers.	`wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport`
Microsoft® Visual C++	`msdn.microsoft.com/en-us/library/hh567368.aspx`
GNU Compiler Collection (g++)	`gcc.gnu.org/projects/cxx0x.html`
Intel® C++ Compiler	`software.intel.com/en-us/articles/c0x-features-supported-by-intel-c-compiler/`
IBM® XL C/C++	`www.ibm.com/developerworks/mydeveloperworks/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/xlc_compiler_s_c_11_support50?lang=en`
Clang	`clang.llvm.org/cxx_status.html`
EDG ecpp	`www.edg.com/docs/edg_cpp.pdf`

Fig. 1.22 | C++ compilers that have implemented major portions of C++11.

Boost C++ Libraries

The **Boost C++ Libraries** are free, open-source libraries created by members of the C++ community. They are peer reviewed and portable across many compilers and platforms. Boost has grown to over 100 libraries, with more being added regularly. Today there are thousands of programmers in the Boost open source community. Boost provides C++ programmers with useful libraries that work well with the existing C++ Standard Library. The Boost libraries can be used by C++ programmers working on a wide variety of platforms with many different compilers.

Some of the new C++11 Standard Library features were derived from corresponding Boost libraries. We overview the libraries and provide code examples for the "regular expression" and "smart pointer" libraries, among others.

Regular expressions are used to match specific character patterns in text. They can be used to validate data to ensure that it's in a particular format, to replace parts of one string with another, or to split a string.

Many common bugs in C and C++ code are related to pointers, a powerful programming capability that C++ absorbed from C. As you'll see, **smart pointers** help you avoid errors associated with traditional pointers.

1.8 Web Resources

This section provides links to our C++ and related Resource Centers that will be useful to you as you learn C++. These include blogs, articles, whitepapers, compilers, development tools, downloads, FAQs, tutorials, webcasts, wikis and links to C++ game programming resources. For updates on Deitel publications, Resource Centers, training courses, partner offers and more, follow us on Facebook® at `www.facebook.com/deitelfan/`, Twitter® `@deitel`, Google+ at `gplus.to/deitel` and LinkedIn at `bit.ly/DeitelLinkedIn`.

Deitel & Associates Websites

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

The Deitel & Associates *C++11 for Programmers* site. Here you'll find links to the book's examples and other resources.

`www.deitel.com/cplusplus/`
`www.deitel.com/visualcplusplus/`
`www.deitel.com/codesearchengines/`
`www.deitel.com/programmingprojects/`

Check these Resource Centers for compilers, code downloads, tutorials, documentation, books, e-books, articles, blogs, RSS feeds and more that will help you develop C++ applications.

`www.deitel.com`

Check this site for updates, corrections and additional resources for all Deitel publications.

`www.deitel.com/newsletter/subscribe.html`

Subscribe here to the *Deitel® Buzz Online* e-mail newsletter to follow the Deitel & Associates publishing program, including updates and errata to *C++11 for Programmers*.

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

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.

3

Introduction to Classes, Objects and Strings

Objectives

In this chapter you'll:

- Define a class and use it to create an object.
- Implement a class's behaviors as member functions.
- Implement a class's attributes as data members.
- Call a member function of an object to perform a task.
- Learn the differences between data members of a class and local variables of a function.
- Use a constructor to initialize an object's data when the object is created.
- Engineer a class to separate its interface from its implementation and encourage reuse.
- Use objects of class `string`.

Outline

3.1 Introduction

In this chapter, you'll begin writing programs that employ the basic concepts of *object-oriented programming* that we introduced in Section 1.3. One common feature of every program in Chapter 2 was that all the statements that performed tasks were located in function `main`. Typically, the programs you develop in this book will consist of function `main` and one or more *classes*, each containing *data members* and *member functions*. If you become part of a development team in industry, you might work on software systems that contain hundreds, or even thousands, of classes. In this chapter, we develop a simple, well-engineered framework for organizing object-oriented programs in C++.

We present a carefully paced sequence of complete working programs to demonstrate creating and using your own classes. These examples begin our integrated case study on developing a grade-book class that instructors can use to maintain student test scores. We also introduce the C++ standard library class `string`.

3.2 Defining a Class with a Member Function

We begin with an example (Fig. 3.1) that consists of class `GradeBook` (lines 8–16)—which, when it's fully developed in Chapter 7, will represent a grade book that an instructor can use to maintain student test scores—and a `main` function (lines 19–23) that creates a `GradeBook` object. Function `main` uses this object and its `displayMessage` member function (lines 12–15) to display a message on the screen welcoming the instructor to the grade-book program.

```
1  // Fig. 3.1: fig03_01.cpp
2  // Define class GradeBook with a member function displayMessage,
3  // create a GradeBook object, and call its displayMessage function.
4  #include <iostream>
5  using namespace std;
6
7  // GradeBook class definition
8  class GradeBook
9  {
```

Fig. 3.1 | Define class `GradeBook` with a member function `displayMessage`, create a `GradeBook` object and call its `displayMessage` function. (Part 1 of 2.)

```
10 public:
11    // function that displays a welcome message to the GradeBook user
12    void displayMessage() const
13    {
14       cout << "Welcome to the Grade Book!" << endl;
15    } // end function displayMessage
16 }; // end class GradeBook
17
18 // function main begins program execution
19 int main()
20 {
21    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
22    myGradeBook.displayMessage(); // call object's displayMessage function
23 } // end main
```

```
Welcome to the Grade Book!
```

Fig. 3.1 | Define class `GradeBook` with a member function `displayMessage`, create a `GradeBook` object and call its `displayMessage` function. (Part 2 of 2.)

*Class **GradeBook***

Before function `main` (lines 19–23) can create a `GradeBook` object, we must tell the compiler what member functions and data members belong to the class. The `GradeBook` **class definition** (lines 8–16) contains a member function called `displayMessage` (lines 12–15) that displays a message on the screen (line 14). We need to make an object of class `GradeBook` (line 21) and call its `displayMessage` member function (line 22) to get line 14 to execute and display the welcome message. We'll soon explain lines 21–22 in detail.

The class definition begins in line 8 with the keyword `class` followed by the class name `GradeBook`. By convention, the name of a user-defined class begins with a capital letter, and for readability, each subsequent word in the class name begins with a capital letter. This capitalization style is often referred to as **Pascal case**, because the convention was widely used in the Pascal programming language. The occasional uppercase letters resemble a camel's humps. More generally, **camel case** capitalization style allows the first letter to be either lowercase or uppercase (e.g., `myGradeBook` in line 21).

Every class's **body** is enclosed in a pair of left and right braces (`{` and `}`), as in lines 9 and 16. The class definition terminates with a semicolon (line 16).

Common Programming Error 3.1

Forgetting the semicolon at the end of a class definition is a syntax error.

Recall that the function `main` is always called automatically when you execute a program. Most functions do *not* get called automatically. As you'll soon see, you must call member function `displayMessage` *explicitly* to tell it to perform its task.

Line 10 contains the keyword **`public`**, which is an **access specifier**. Lines 12–15 define member function `displayMessage`. This member function appears *after* access specifier `public:` to indicate that the function is "available to the public"—that is, it can be called by other functions in the program (such as `main`), and by member functions of other classes (if there are any). Access specifiers are always followed by a colon (`:`). For the remainder of the text, when we refer to the access specifier `public` in the text, we'll omit the colon as we did in this sentence. Section 3.4 introduces the access specifier `private`. Later in the book we'll study the access specifier `protected`.

When you define a function, you must specify a **return type** to indicate the type of the value returned by the function when it completes its task. In line 12, keyword **`void`** to the left of the function name `displayMessage` is the function's return type. Return type `void` indicates that `displayMessage` will *not* return any data to its **calling function** (in this example, line 22 of `main`, as we'll see in a moment) when it completes its task. In Fig. 3.5, you'll see an example of a function that *does* return a value.

The name of the member function, `displayMessage`, follows the return type (line 12). By convention, our function names use the *camel case* style with a lowercase first letter. The parentheses after the member function name indicate that this is a *function*. An empty set of parentheses, as shown in line 12, indicates that this member function does *not* require additional data to perform its task. You'll see an example of a member function that *does* require additional data in Section 3.3.

We declared member function `displayMessage` **`const`** in line 12 because in the process of displaying `"Welcome to the Grade Book!"` the function *does not*, and *should not*, modify the `GradeBook` object on which it's called. Declaring `displayMessage const` tells the compiler, "this function should *not* modify the object on which it's called—if it does, please issue a compilation error." This can help you locate errors if you accidentally insert code in `displayMessage` that *would* modify the object. Line 12 is commonly referred to as a **function header**.

Every function's *body* is delimited by left and right braces (`{` and `}`), as in lines 13 and 15. The *function body* contains statements that perform the function's task. In this case, member function `displayMessage` contains one statement (line 14) that displays the message `"Welcome to the Grade Book!"`. After this statement executes, the function has completed its task.

Testing Class **GradeBook**

Next, we'd like to use class `GradeBook` in a program. As you saw in Chapter 2, the function `main` (lines 19–23) begins the execution of every program.

In this program, we'd like to call class `GradeBook`'s `displayMessage` member function to display the welcome message. Typically, you cannot call a member function of a class until you *create an object* of that class. (As you'll learn in Section 9.14, `static` member functions are an exception.) Line 21 creates an object of class `GradeBook` called `myGradeBook`. The variable's type is `GradeBook`—the class we defined in lines 8–16. When we declare variables of type `int`, as we did in Chapter 2, the compiler knows what `int` is—it's a *fundamental type* that's "built into" C++. In line 21, however, the compiler does *not* automatically know what type `GradeBook` is—it's a **user-defined type**. We tell the compiler what `GradeBook` is by including the *class definition* (lines 8–16). If we omitted these lines, the compiler would issue an error message. Each class you create becomes a new *type* that can be used to create objects. You can define new class types as needed; this is one reason why C++ is known as an **extensible programming language**.

Line 22 *calls* the member function `displayMessage` using variable `myGradeBook` followed by the **dot operator** (`.`), the function name `displayMessage` and an empty set of parentheses. This call causes the `displayMessage` function to perform its task. At the beginning of line 22, "`myGradeBook.`" indicates that `main` should use the `GradeBook` object that was created in line 21. The *empty parentheses* in line 12 indicate that member function `displayMessage` does *not* require additional data to perform its task, which is why we called this function with empty parentheses in line 22. (In Section 3.3, you'll see how to pass data to a function.) When `displayMessage` completes its task, the program reaches the end of `main` (line 23) and terminates.

UML Class Diagram for Class `GradeBook`

Recall from Section 1.3 that the UML is a standardized graphical language used by software developers to represent their object-oriented systems. In the UML, each class is modeled in a **UML class diagram** as a *rectangle* with three *compartments*. Figure 3.2 presents a class diagram for class `GradeBook` (Fig. 3.1). The *top compartment* contains the class's name centered horizontally and in boldface type. The *middle compartment* contains the class's attributes, which correspond to data members in C++. This compartment is currently empty, because class `GradeBook` does not yet have any attributes. (Section 3.4 presents a version of class `GradeBook` with an attribute.) The *bottom compartment* contains the class's operations, which correspond to member functions in C++. The UML models operations by listing the operation name followed by a set of parentheses. Class `GradeBook` has only one member function, `displayMessage`, so the bottom compartment of Fig. 3.2 lists one operation with this name. Member function `displayMessage` does *not* require additional information to perform its tasks, so the parentheses following `displayMessage` in the class diagram are *empty*, just as they are in the member function's header in line 12 of Fig. 3.1. The *plus sign (+)* in front of the operation name indicates that `displayMessage` is a *public* operation in the UML (i.e., a `public` member function in C++).

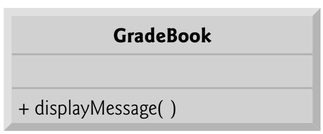

Fig. 3.2 | UML class diagram indicating that class `GradeBook` has a public `displayMessage` operation.

3.3 Defining a Member Function with a Parameter

In our car analogy from Section 1.3, we mentioned that pressing a car's gas pedal sends a *message* to the car to perform a task—make the car go faster. But *how fast* should the car accelerate? As you know, the farther down you press the pedal, the faster the car accelerates. So the message to the car includes *both* the *task to perform* and *additional information that helps the car perform the task*. This additional information is known as a **parameter**—the *value* of the parameter helps the car determine how fast to accelerate. Similarly, a member function can require one or more parameters that represent additional data it needs to perform its task. A function call supplies values—called **arguments**—for each of the function's parameters. For example, to make a deposit into a bank account, suppose a `deposit` member function of an `Account` class specifies a parameter that represents the *deposit amount*. When the `deposit` member function is called, an argument value representing the deposit amount is copied to the member function's parameter. The member function then adds that amount to the account balance.

Defining and Testing Class GradeBook

Our next example (Fig. 3.3) redefines class `GradeBook` (lines 9–18) with a `displayMessage` member function (lines 13–17) that displays the course name as part of the welcome message. The new version of `displayMessage` requires a *parameter* (`courseName` in line 13) that represents the course name to output.

```
1  // Fig. 3.3: fig03_03.cpp
2  // Define class GradeBook with a member function that takes a parameter,
3  // create a GradeBook object and call its displayMessage function.
4  #include <iostream>
5  #include <string> // program uses C++ standard string class
6  using namespace std;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    // function that displays a welcome message to the GradeBook user
13    void displayMessage( string courseName ) const
14    {
15       cout << "Welcome to the grade book for\n" << courseName << "!"
16          << endl;
17    } // end function displayMessage
18 }; // end class GradeBook
19
20 // function main begins program execution
21 int main()
22 {
23    string nameOfCourse; // string of characters to store the course name
24    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
25
26    // prompt for and input course name
27    cout << "Please enter the course name:" << endl;
28    getline( cin, nameOfCourse ); // read a course name with blanks
29    cout << endl; // output a blank line
30
31    // call myGradeBook's displayMessage function
32    // and pass nameOfCourse as an argument
33    myGradeBook.displayMessage( nameOfCourse );
34 } // end main
```

```
Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

Fig. 3.3 | Define class `GradeBook` with a member function that takes a parameter, create a `GradeBook` object and call its `displayMessage` function.

Before discussing the new features of class GradeBook, let's see how the new class is used in main (lines 21–34). Line 23 creates a variable of type **string** called nameOfCourse that will be used to store the course name entered by the user. A variable of type string represents a string of characters such as "CS101 Introduction to C++ Programming". A string is actually an *object* of the C++ Standard Library class string. This class is defined in **header <string>**, and the name string, like cout, belongs to namespace std. To enable lines 13 and 23 to compile, line 5 *includes* the <string> header. The using directive in line 6 allows us to simply write string in line 23 rather than std::string. For now, you can think of string variables like variables of other types such as int. You'll learn additional string capabilities in Section 3.8 and in Chapter 19.

Line 24 creates an object of class GradeBook named myGradeBook. Line 27 prompts the user to enter a course name. Line 28 reads the name from the user and assigns it to the nameOfCourse variable, using the library function **getline** to perform the input. Before we explain this line of code, let's explain why we cannot simply write

```
cin >> nameOfCourse;
```

to obtain the course name.

In our sample program execution, we use the course name "CS101 Introduction to C++ Programming," which contains multiple words *separated by blanks*. (Recall that we highlight user-entered data in bold.) When reading a string with the stream extraction operator, cin reads characters *until the first white-space character is reached*. Thus, only "CS101" would be read by the preceding statement. The rest of the course name would have to be read by subsequent input operations.

In this example, we'd like the user to type the complete course name and press *Enter* to submit it to the program, and we'd like to store the *entire* course name in the string variable nameOfCourse. The function call getline(cin, nameOfCourse) in line 28 reads characters (*including* the space characters that separate the words in the input) from the standard input stream object cin (i.e., the keyboard) until the *newline* character is encountered, places the characters in the string variable nameOfCourse and *discards* the newline character. When you press *Enter* while entering data, a newline is inserted in the input stream. The <string> header must be included in the program to use function getline, which belongs to namespace std.

Line 33 calls myGradeBook's displayMessage member function. The nameOfCourse variable in parentheses is the *argument* that's passed to member function displayMessage so that it can perform its task. The value of variable nameOfCourse in main is *copied* to member function displayMessage's parameter courseName in line 13. When you execute this program, member function displayMessage outputs as part of the welcome message the course name you type (in our sample execution, CS101 Introduction to C++ Programming).

More on Arguments and Parameters

To specify in a function definition that the function requires data to perform its task, you place additional information in the function's **parameter list**, which is located in the parentheses following the function name. The parameter list may contain *any* number of parameters, including *none at all* (represented by empty parentheses as in Fig. 3.1, line 12) to indicate that a function does *not* require any parameters. The displayMessage member function's parameter list (Fig. 3.3, line 13) declares that the function requires one parameter. Each parameter specifies a *type* and an *identifier*. The type string and the identifier `courseName` indicate that member function `displayMessage` requires a `string` to perform its task. The member function body uses the parameter `courseName` to access the value that's passed to the function in the function call (line 33 in `main`). Lines 15–16 display parameter `courseName`'s value as part of the welcome message. The parameter variable's name (`courseName` in line 13) can be the *same* as or *different* from the argument variable's name (`nameOfCourse` in line 33)—you'll learn why in Chapter 6.

A function can specify multiple parameters by separating each from the next with a comma. The number and order of arguments in a function call *must match* the number and order of parameters in the parameter list of the called member function's header. Also, the argument types in the function call must be consistent with the types of the corresponding parameters in the function header. (As you'll learn in subsequent chapters, an argument's type and its corresponding parameter's type need not always be *identical*, but they must be "consistent.") In our example, the one `string` argument in the function call (i.e., `nameOfCourse`) *exactly matches* the one `string` parameter in the member-function definition (i.e., `courseName`).

Updated UML Class Diagram for Class **GradeBook**

The UML class diagram of Fig. 3.4 models class `GradeBook` of Fig. 3.3. Like the class `GradeBook` defined in Fig. 3.1, this `GradeBook` class contains `public` member function `displayMessage`. However, this version of `displayMessage` has a *parameter*. The UML models a parameter by listing the parameter name, followed by a colon and the parameter type in the parentheses following the operation name. The UML has its *own* data types *similar* to those of C++. The UML is *language independent*—it's used with many different programming languages—so its terminology does not exactly match that of C++. For example, the UML type `String` corresponds to the C++ type `string`. Member function `displayMessage` of class `GradeBook` (Fig. 3.3, lines 13–17) has a `string` parameter named `courseName`, so Fig. 3.4 lists `courseName : String` between the parentheses following the operation name `displayMessage`. This version of the `GradeBook` class still does *not* have any data members.

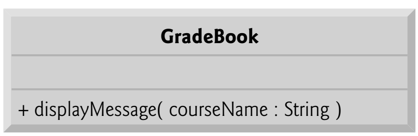

Fig. 3.4 | UML class diagram indicating that class `GradeBook` has a public `displayMessage` operation with a `courseName` parameter of UML type `String`.

3.4 Data Members, *set* Member Functions and *get* Member Functions

In Chapter 2, we declared all of a program's variables in its `main` function. Variables declared in a function definition's body are known as **local variables** and can be used *only* from the line of their declaration in the function to the closing right brace (`}`) of the block in which they're declared. A local variable must be declared *before* it can be used in a function. A local variable cannot be accessed *outside* the function in which it's declared. *When* *a function terminates, the values of its local variables are lost.* (You'll see an exception to this in Chapter 6 when we discuss `static` local variables.)

A class normally consists of one or more member functions that manipulate the attributes that belong to a particular object of the class. Attributes are represented as variables in a class definition. Such variables are called **data members** and are declared *inside* a class definition but *outside* the bodies of the class's member-function definitions. Each object of a class maintains its own attributes in memory. These attributes exist throughout the life of the object. The example in this section demonstrates a `GradeBook` class that contains a `courseName` data member to represent a particular `GradeBook` object's course name. If you create more than one `GradeBook` object, each will have its own `courseName` data member, and these can contain *different* values.

`GradeBook` Class with a Data Member, and set *and* get *Member Functions*

In our next example, class `GradeBook` (Fig. 3.5) maintains the course name as a *data member* so that it can be *used* or *modified* throughout a program's execution. The class contains member functions `setCourseName`, `getCourseName` and `displayMessage`. Member function `setCourseName` *stores* a course name in a `GradeBook` data member. Member function `getCourseName` *obtains* the course name from that data member. Member function `displayMessage`—which now specifies *no parameters*—still displays a welcome message that includes the course name. However, as you'll see, the function now *obtains* the course name by calling another function in the same class—`getCourseName`.

```
// Fig. 3.5: fig03_05.cpp
// Define class GradeBook that contains a courseName data member
// and member functions to set and get its value;
// Create and manipulate a GradeBook object with these functions.
#include <iostream>
#include <string> // program uses C++ standard string class
using namespace std;

// GradeBook class definition
class GradeBook
{
public:
   // function that sets the course name
   void setCourseName( string name )
   {
      courseName = name; // store the course name in the object
   } // end function setCourseName

   // function that gets the course name
   string getCourseName() const
   {
      return courseName; // return the object's courseName
   } // end function getCourseName

```

Fig. 3.5 | Defining and testing class `GradeBook` with a data member and *set* and *get* member functions. (Part 1 of 2.)

```
25    // function that displays a welcome message
26    void displayMessage() const
27    {
28       // this statement calls getCourseName to get the
29       // name of the course this GradeBook represents
30       cout << "Welcome to the grade book for\n" << getCourseName() << "!"
31          << endl;
32    } // end function displayMessage
33 private:
34    string courseName; // course name for this GradeBook
35 }; // end class GradeBook
36
37 // function main begins program execution
38 int main()
39 {
40    string nameOfCourse; // string of characters to store the course name
41    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
42
43    // display initial value of courseName
44    cout << "Initial course name is: " << myGradeBook.getCourseName()
45       << endl;
46
47    // prompt for, input and set course name
48    cout << "\nPlease enter the course name:" << endl;
49    getline( cin, nameOfCourse ); // read a course name with blanks
50    myGradeBook.setCourseName( nameOfCourse ); // set the course name
51
52    cout << endl; // outputs a blank line
53    myGradeBook.displayMessage(); // display message with new course name
54 } // end main
```

```
Initial course name is:

Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

Fig. 3.5 | Defining and testing class `GradeBook` with a data member and *set* and *get* member functions. (Part 2 of 2.)

A typical instructor teaches *several* courses, each with its own course name. Line 34 declares that `courseName` is a variable of type `string`. Because the variable is declared in the class definition (lines 10–35) but outside the bodies of the class's member-function definitions (lines 14–17, 20–23 and 26–32), the variable is a *data member*. Every instance (i.e., object) of class `GradeBook` contains each of the class's data members—if there are two `GradeBook` objects, each has its *own* `courseName` (one per object), as you'll see in the example of Fig. 3.7. A benefit of making `courseName` a data member is that *all* the member functions of the class can manipulate any data members that appear in the class definition (in this case, `courseName`).

Access Specifiers **`public`** *and* **`private`**

Most data-member declarations appear after the **`private`** access specifier. Variables or functions declared after access specifier `private` (and *before* the next access specifier if there is one) are accessible only to member functions of the class for which they're declared (or to "friends" of the class, as you'll see in Chapter 9). Thus, data member `courseName` can be used *only* in member functions `setCourseName`, `getCourseName` and `displayMessage` of class `GradeBook` (or to "friends" of the class, if there are any).

Error-Prevention Tip 3.1

Making the data members of a class `private` *and the member functions of the class* `public` *facilitates debugging because problems with data manipulations are localized to either the class's member functions or the friends of the class.*

Common Programming Error 3.2

An attempt by a function, which is not a member of a particular class (or a friend of that class) to access a `private` *member of that class is a compilation error.*

The *default access* for class members is `private` so all members *after* the class header and *before* the first access specifier (if there are any) are `private`. The access specifiers `public` and `private` may be repeated, but this is unnecessary and can be confusing.

Declaring data members with access specifier `private` is known as **data hiding**. When a program creates a `GradeBook` object, data member `courseName` is *encapsulated* (hidden) in the object and can be accessed only by member functions of the object's class. In class `GradeBook`, member functions `setCourseName` and `getCourseName` manipulate the data member `courseName` directly.

Member Functions **`setCourseName`** *and* **`getCourseName`**

Member function `setCourseName` (lines 14–17) does not *return* any data when it completes its task, so its return type is `void`. The member function *receives* one parameter—`name`—which represents the course name that will be passed to it as an argument (as we'll see in line 50 of `main`). Line 16 assigns `name` to data member `courseName`, thus *modifying* the object—for this reason, we do *not* declare `setCourseName const`. In this example, `setCourseName` does not *validate* the course name—i.e., the function does *not* check that the course name adheres to any particular format or follows any other rules regarding what a "valid" course name looks like. Suppose, for instance, that a university can print student transcripts containing course names of only 25 characters or fewer. In this case, we might want class `GradeBook` to ensure that its data member `courseName` never contains more than 25 characters. We discuss validation in Section 3.8.

Member function `getCourseName` (lines 20–23) *returns* a particular `GradeBook` object's `courseName`, *without* modifying the object—for this reason, we declare `getCourseName const`. The member function has an *empty parameter list*, so it does *not* require additional data to perform its task. The function specifies that it returns a `string`. When a function that specifies a return type other than `void` is called and completes its task, the function uses a **`return` statement** (as in line 22) to *return a result* to its calling function. For example, when you go to an automated teller machine (ATM) and request your account balance, you expect the ATM to give you a value that represents your balance. Similarly, when a statement calls member function `getCourseName` on a `GradeBook` object, the statement expects to receive the GradeBook's course name (in this case, a string, as specified by the function's return type).

If you have a function square that returns the square of its argument, the statement

```
result = square( 2 );
```

returns 4 from function square and assigns to variable result the value 4. If you have a function maximum that returns the largest of three integer arguments, the statement

```
biggest = maximum( 27, 114, 51 );
```

returns 114 from function maximum and assigns this value to variable biggest.

The statements in lines 16 and 22 each use variable courseName (line 34) even though it was *not* declared in any of the member functions. We can do this because courseName is a *data member* of the class and data members are accessible from a class's member functions.

Member Function displayMessage

Member function displayMessage (lines 26–32) does *not* return any data when it completes its task, so its return type is void. The function does *not* receive parameters, so its parameter list is empty. Lines 30–31 output a welcome message that includes the value of data member courseName. Line 30 calls member function getCourseName to obtain the value of courseName. Member function displayMessage could also access data member courseName directly, just as member functions setCourseName and getCourseName do. We explain shortly why it's preferable from a software engineering perspective to call member function getCourseName to obtain the value of courseName.

Testing Class GradeBook

The main function (lines 38–54) creates one object of class GradeBook and uses each of its member functions. Line 41 creates a GradeBook object named myGradeBook. Lines 44–45 display the initial course name by calling the object's getCourseName member function. The first line of the output does not show a course name, because the object's courseName data member (i.e., a string) is initially empty—by default, the initial value of a string is the so-called **empty string**, i.e., a string that does not contain any characters. Nothing appears on the screen when an empty string is displayed.

Line 48 prompts the user to enter a course name. Local string variable nameOfCourse (declared in line 40) is set to the course name entered by the user, which is obtained by the call to the getline function (line 49). Line 50 calls object myGradeBook's setCourseName member function and supplies nameOfCourse as the function's argument. When the function is called, the argument's value is copied to parameter name (line 14) of member function setCourseName. Then the parameter's value is assigned to data member courseName (line 16). Line 52 skips a line; then line 53 calls object myGradeBook's displayMessage member function to display the welcome message containing the course name.

Software Engineering with Set *and* Get *Functions*

A class's private data members can be manipulated *only* by member functions of that class (and by "friends" of the class as you'll see in Chapter 9). So a **client of an object**—that is, any statement that calls the object's member functions from *outside* the object—calls the class's `public` member functions to request the class's services for particular objects of the class. This is why the statements in function `main` call member functions `setCourseName`, `getCourseName` and `displayMessage` on a `GradeBook` object. Classes often provide `public` member functions to allow clients of the class to ***set*** (i.e., assign values to) or ***get*** (i.e., obtain the values of) `private` data members. These member function names need not begin with `set` or `get`, but this naming convention is common. In this example, the member function that *sets* the `courseName` data member is called `setCourseName`, and the member function that *gets* the value of the `courseName` data member is called `getCourseName`. *Set* functions are sometimes called **mutators** (because they mutate, or change, values), and *get* functions are also called **accessors** (because they access values).

Recall that declaring data members with access specifier `private` enforces data hiding. Providing `public` *set* and *get* functions allows clients of a class to access the hidden data, but only *indirectly*. The client knows that it's attempting to modify or obtain an object's data, but the client does *not* know *how* the object performs these operations. In some cases, a class may *internally* represent a piece of data one way, but expose that data to clients in a different way. For example, suppose a `Clock` class represents the time of day as a `private int` data member `time` that stores the number of seconds since midnight. However, when a client calls a `Clock` object's `getTime` member function, the object could return the time with hours, minutes and seconds in a `string` in the format `"HH:MM:SS"`. Similarly, suppose the `Clock` class provides a *set* function named `setTime` that takes a `string` parameter in the `"HH:MM:SS"` format. Using `string` capabilities presented in Chapter 19, the `setTime` function could convert this `string` to a number of seconds, which the function stores in its `private` data member. The *set* function could also check that the value it receives represents a valid time (e.g., `"12:30:45"` is valid but `"42:85:70"` is not). The *set* and *get* functions allow a client to interact with an object, but the object's `private` data remains safely *encapsulated* (i.e., hidden) in the object itself.

The *set* and *get* functions of a class also should be used by other member functions *within* the class to manipulate the class's `private` data, even though these member functions *can* access the `private` data directly. In Fig. 3.5, member functions `setCourseName` and `getCourseName` are `public` member functions, so they're accessible to clients of the class, as well as to the class itself. Member function `displayMessage` calls member function `getCourseName` to obtain the value of data member `courseName` for display purposes, even though `displayMessage` can access `courseName` directly—accessing a data member via its *get* function creates a better, more robust class (i.e., a class that's easier to maintain and less likely to malfunction). If we decide to change the data member `courseName` in some way, the `displayMessage` definition will *not* require modification—only the bodies of the *get* and *set* functions that directly manipulate the data member will need to change. For example, suppose we want to represent the course name as two separate data members—`courseNumber` (e.g., `"CS101"`) and `courseTitle` (e.g., `"Introduction to C++ Programming"`). Member function `displayMessage` can still issue a single call to member function `getCourseName` to obtain the full course name to display as part of the welcome message. In this case, `getCourseName` would need to build and return a `string` containing the `courseNumber` followed by the `courseTitle`. Member function `displayMessage` could continue to display the complete course title "`CS101 Introduction to C++ Programming`." The benefits of calling a *set* function from another member function of the same class will become clearer when we discuss validation in Section 3.8.

Good Programming Practice 3.1

Always try to localize *the effects of changes to a class's data members by accessing and manipulating the data members through their corresponding* get *and* set *functions.*

Software Engineering Observation 3.1

Write programs that are clear and easy to maintain. Change is the rule rather than the exception. You should anticipate that your code will be modified, and possibly often.

GradeBook's UML Class Diagram with a Data Member and* set *and* get *Functions

Figure 3.6 contains an updated UML class diagram for the version of class `GradeBook` in Fig. 3.5. This diagram models `GradeBook`'s data member `courseName` as an attribute in the middle compartment. The UML represents data members as attributes by listing the attribute name, followed by a colon and the attribute type. The UML type of attribute `courseName` is `String`, which corresponds to `string` in C++. Data member `courseName` is `private` in C++, so the class diagram lists a *minus sign (–)* in front of the corresponding attribute's name. Class `GradeBook` contains three `public` member functions, so the class diagram lists three operations in the third compartment. Operation `setCourseName` has a `String` parameter called `name`. The UML indicates the *return type* of an operation by placing a colon and the return type after the parentheses following the operation name. Member function `getCourseName` of class `GradeBook` has a `string` return type in C++, so the class diagram shows a `String` return type in the UML. Operations `setCourseName` and `displayMessage` do not return values (i.e., they return `void` in C++), so the UML class diagram does not specify a return type after the parentheses of these operations.

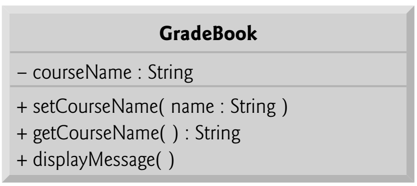

Fig. 3.6 | UML class diagram for class `GradeBook` with a private `courseName` attribute and public operations `setCourseName`, `getCourseName` and `displayMessage`.

3.5 Initializing Objects with Constructors

As mentioned in Section 3.4, when an object of class `GradeBook` (Fig. 3.5) is created, its data member `courseName` is initialized to the empty string by default. What if you want to provide a course name when you *create* a `GradeBook` object? Each class you declare can provide one or more **constructors** that can be used to initialize an object of the class when the object is created. A constructor is a special member function that must be defined with the *same name as the class*, so that the compiler can distinguish it from the class's other member functions. An important difference between constructors and other functions is that *constructors cannot return values*, so they *cannot* specify a return type (not even `void`). Normally, constructors are declared `public`. In the early chapters, our classes will generally have one constructor—in later chapters, you'll see how to create classes with more that one constructor using the technique of *function overloading*, which we introduce in Section 6.17.

C++ automatically calls a constructor for each object that's created, which helps ensure that objects are initialized properly before they're used in a program. The constructor call occurs when the object is created. If a class does not *explicitly* include constructors, the compiler provides a **default constructor** with *no* parameters. For example, when line 41 of Fig. 3.5 creates a `GradeBook` object, the default constructor is called. The default constructor provided by the compiler creates a `GradeBook` object without giving any initial values to the object's fundamental type data members. For data members that are objects of other classes, the default constructor implicitly calls each data member's default constructor to ensure that the data member is initialized properly. This is why the `string` data member `courseName` (in Fig. 3.5) was initialized to the empty string—the default constructor for class `string` sets the `string`'s value to the empty string.

In the example of Fig. 3.7, we specify a course name for a `GradeBook` object when the object is created (e.g., line 47). In this case, the argument `"CS101 Introduction to C++ Programming"` is passed to the `GradeBook` object's constructor (lines 14–18) and used to initialize the `courseName`. Figure 3.7 defines a modified `GradeBook` class containing a constructor with a `string` parameter that receives the initial course name.

```
1  // Fig. 3.7: fig03_07.cpp
2  // Instantiating multiple objects of the GradeBook class and using
3  // the GradeBook constructor to specify the course name
4  // when each GradeBook object is created.
5  #include <iostream>
6  #include <string> // program uses C++ standard string class
7  using namespace std;
8
9  // GradeBook class definition
10 class GradeBook
11 {
12 public:
13    // constructor initializes courseName with string supplied as argument
14    explicit GradeBook( string name )
15       : courseName( name ) // member initializer to initialize courseName
16    {
17       // empty body
18    } // end GradeBook constructor
19
20    // function to set the course name
21    void setCourseName( string name )
22    {
23       courseName = name; // store the course name in the object
24    } // end function setCourseName
25
26    // function to get the course name
27    string getCourseName() const
28    {
```

Fig. 3.7 | Instantiating multiple objects of the `GradeBook` class and using the `GradeBook` constructor to specify the course name when each `GradeBook` object is created. (Part 1 of 2.)

```
29          return courseName; // return object's courseName
30       } // end function getCourseName
31
32       // display a welcome message to the GradeBook user
33       void displayMessage() const
34       {
35          // call getCourseName to get the courseName
36          cout << "Welcome to the grade book for\n" << getCourseName()
37             << "!" << endl;
38       } // end function displayMessage
39    private:
40       string courseName; // course name for this GradeBook
41    }; // end class GradeBook
42
43    // function main begins program execution
44    int main()
45    {
46       // create two GradeBook objects
47       GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
48       GradeBook gradeBook2( "CS102 Data Structures in C++" );
49
50       // display initial value of courseName for each GradeBook
51       cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
52          << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
53          << endl;
54    } // end main
```

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

Fig. 3.7 | Instantiating multiple objects of the `GradeBook` class and using the `GradeBook` constructor to specify the course name when each `GradeBook` object is created. (Part 2 of 2.)

Defining a Constructor

Lines 14–18 of Fig. 3.7 define a constructor for class `GradeBook`. The constructor has the *same* name as its class, `GradeBook`. A constructor specifies in its parameter list the data it requires to perform its task. When you create a new object, you place this data in the parentheses that follow the object name (as we did in lines 47–48). Line 14 indicates that class `GradeBook`'s constructor has a `string` parameter called `name`. We declared this constructor **`explicit`**, because it takes a *single* parameter—this is important for subtle reasons that you'll learn in Section 10.13. For now, just declare *all* single-parameter constructors `explicit`. Line 14 does *not* specify a return type, because constructors *cannot* return values (or even `void`). Also, constructors cannot be declared `const` (because initializing an object modifies it).

The constructor uses a **member-initializer list** (line 15) to initialize the `courseName` data member with the value of the constructor's parameter `name`. *Member initializers* appear between a constructor's parameter list and the left brace that begins the constructor's body. The member initializer list is separated from the parameter list with a *colon (:)*. A member initializer consists of a data member's *variable name* followed by parentheses containing the member's *initial value*. In this example, `courseName` is initialized with the value of the parameter `name`. If a class contains more than one data member, each data member's initializer is separated from the next by a comma. The member initializer list executes *before* the body of the constructor executes. You can perform initialization in the constructor's body, but you'll learn later in the book that it's more efficient to do it with member initializers, and some types of data members must be initialized this way.

Notice that both the constructor (line 14) and the `setCourseName` function (line 21) use a parameter called `name`. You can use the *same* parameter names in *different* functions because the parameters are *local* to each function—they do *not* interfere with one another.

Testing Class GradeBook

Lines 44–54 of Fig. 3.7 define the `main` function that tests class `GradeBook` and demonstrates initializing `GradeBook` objects using a constructor. Line 47 creates and initializes `GradeBook` object `gradeBook1`. When this line executes, the `GradeBook` constructor (lines 14–18) is called with the argument `"CS101 Introduction to C++ Programming"` to initialize `gradeBook1`'s course name. Line 48 repeats this process for `GradeBook` object `gradeBook2`, this time passing the argument `"CS102 Data Structures in C++"` to initialize `gradeBook2`'s course name. Lines 51–52 use each object's `getCourseName` member function to obtain the course names and show that they were indeed initialized when the objects were created. The output confirms that each `GradeBook` object maintains its *own* data member `courseName`.

Ways to Provide a Default Constructor for a Class

Any constructor that takes *no* arguments is called a default constructor. A class can get a default constructor in one of several ways:

1. The compiler *implicitly* creates a default constructor in every class that does *not* have any user-defined constructors. The default constructor does *not* initialize the class's data members, but *does* call the default constructor for each data member that's an object of another class. An uninitialized variable contains an undefined ("garbage") value.
2. You *explicitly* define a constructor that takes no arguments. Such a default constructor will call the default constructor for each data member that's an object of another class and will perform additional initialization specified by you.
3. *If you define any constructors with arguments, C++ will not implicitly create a default constructor for that class.* We'll show later that C++11 allows you to force the compiler to create the default constructor even if you've defined non-default constructors.

For each version of class `GradeBook` in Fig. 3.1, Fig. 3.3 and Fig. 3.5 the compiler *implicitly* defined a default constructor.

Error-Prevention Tip 3.2

Unless no initialization of your class's data members is necessary (almost never), provide constructors to ensure that your class's data members are initialized with meaningful values when each new object of your class is created.

Software Engineering Observation 3.2

Data members can be initialized in a constructor, or their values may be set *later after the object is created. However, it's a good software engineering practice to ensure that an object is fully initialized* before *the client code invokes the object's member functions. You should not rely on the client code to ensure that an object gets initialized properly.*

Adding the Constructor to Class `GradeBook`*'s UML Class Diagram*

The UML class diagram of Fig. 3.8 models the `GradeBook` class of Fig. 3.7, which has a constructor with a `name` parameter of type `string` (represented by type `String` in the UML). Like operations, the UML models constructors in the third compartment of a class in a class diagram. To distinguish a constructor from a class's operations, the UML places the word "constructor" between guillemets (« and ») before the constructor's name. By convention, you list the class's constructor *before* other operations in the third compartment.

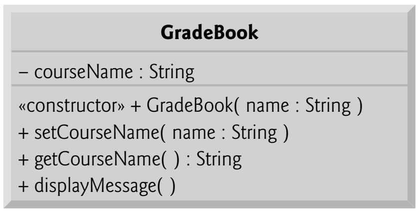

Fig. 3.8 | UML class diagram indicating that class `GradeBook` has a constructor with a `name` parameter of UML type `String`.

3.6 Placing a Class in a Separate File for Reusability

One of the benefits of creating class definitions is that, when packaged properly, your classes can be *reused* by other programmers. For example, you can *reuse* C++ Standard Library type `string` in any C++ program by including the header `<string>` (and, as you'll see, by being able to link to the library's object code).

Programmers who wish to use our `GradeBook` class cannot simply include the file from Fig. 3.7 in another program. As you learned in Chapter 2, function `main` begins the execution of every program, and every program must have *exactly one* `main` function. If other programmers include the code from Fig. 3.7, they get extra "baggage"—our `main` function—and their programs will then have two `main` functions. Attempting to compile a program with two `main` functions produces an error. So, placing `main` in the same file with a class definition *prevents that class from being reused* by other programs. In this section, we demonstrate how to make class `GradeBook` reusable by *separating it into another file* from the `main` function.

Headers

Each of the previous examples in the chapter consists of a single `.cpp` file, also known as a **source-code file**, that contains a `GradeBook` class definition and a `main` function. When building an object-oriented C++ program, it's customary to define *reusable* source code (such as a class) in a file that by convention has a `.h` filename extension—known as a **header**. Programs use `#include` preprocessing directives to include headers and take advantage of reusable software components, such as type `string` provided in the C++ Standard Library and user-defined types like class `GradeBook`.

Our next example separates the code from Fig. 3.7 into two files—`GradeBook.h` (Fig. 3.9) and `fig03_10.cpp` (Fig. 3.10). As you look at the header in Fig. 3.9, notice that it contains only the `GradeBook` class definition (lines 7–38) and the headers on which the class depends. The `main` function that *uses* class `GradeBook` is defined in the source-code file `fig03_10.cpp` (Fig. 3.10) in lines 8–18. To help you prepare for the larger programs you'll encounter later in this book and in industry, we often use a separate source-code file containing function `main` to test our classes (this is called a **driver program**). You'll soon learn how a source-code file with `main` can use the class definition found in a header to create objects of a class.

```
1  // Fig. 3.9: GradeBook.h
2  // GradeBook class definition in a separate file from main.
3  #include <iostream>
4  #include <string> // class GradeBook uses C++ standard string class
5  
6  // GradeBook class definition
7  class GradeBook
8  {
9  public:
10    // constructor initializes courseName with string supplied as argument
11    explicit GradeBook( std::string name )
12       : courseName( name ) // member initializer to initialize courseName
13    {
14       // empty body
15    } // end GradeBook constructor
16 
17    // function to set the course name
18    void setCourseName( std::string name )
19    {
20       courseName = name; // store the course name in the object
21    } // end function setCourseName
22 
23    // function to get the course name
24    std::string getCourseName() const
25    {
26       return courseName; // return object's courseName
27    } // end function getCourseName
28 
29    // display a welcome message to the GradeBook user
30    void displayMessage() const
31    {
32       // call getCourseName to get the courseName
33       std::cout << "Welcome to the grade book for\n" << getCourseName()
34          << "!" << std::endl;
35    } // end function displayMessage
36 private:
37    std::string courseName; // course name for this GradeBook
38 }; // end class GradeBook
```

Fig. 3.9 | `GradeBook` class definition in a separate file from `main`.

```cpp
1  // Fig. 3.10: fig03_10.cpp
2  // Including class GradeBook from file GradeBook.h for use in main.
3  #include <iostream>
4  #include "GradeBook.h" // include definition of class GradeBook
5  using namespace std;
6
7  // function main begins program execution
8  int main()
9  {
10    // create two GradeBook objects
11    GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
12    GradeBook gradeBook2( "CS102 Data Structures in C++" );
13
14    // display initial value of courseName for each GradeBook
15    cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
16       << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
17       << endl;
18 } // end main
```

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

Fig. 3.10 | Including class `GradeBook` from file `GradeBook.h` for use in `main`.

Use `std::` with Standard Library Components in Headers

Throughout the header (Fig. 3.9), we use `std::` when referring to `string` (lines 11, 18, 24 and 37), `cout` (line 33) and `endl` (line 34). For subtle reasons that we'll explain in a later chapter, headers should *never* contain `using` directives or `using` declarations (Section 2.6).

Including a Header That Contains a User-Defined Class

A header such as `GradeBook.h` (Fig. 3.9) cannot be used as a complete program, because it does not contain a `main` function. To test class `GradeBook` (defined in Fig. 3.9), you must write a separate source-code file containing a `main` function (such as Fig. 3.10) that instantiates and uses objects of the class.

The compiler doesn't know what a `GradeBook` is because it's a user-defined type. In fact, the compiler doesn't even know the classes in the C++ Standard Library. To help it understand how to use a class, we must explicitly provide the compiler with the class's definition—that's why, for example, to use type `string`, a program must include the `<string>` header. This enables the compiler to determine the amount of memory that it must reserve for each `string` object and ensure that a program calls a `string`'s member functions correctly.

To create `GradeBook` objects `gradeBook1` and `gradeBook2` in lines 11–12 of Fig. 3.10, the compiler must know the *size* of a `GradeBook` object. While objects conceptually contain data members and member functions, C++ objects actually contain *only* data. The compiler creates only *one* copy of the class's member functions and *shares* that copy among all the class's objects. Each object, of course, needs its own data members, because their contents can vary among objects (such as two different `BankAccount` objects having two different balances). The member-function code, however, is *not modifiable*, so it can be shared among all objects of the class. Therefore, the size of an object depends on the