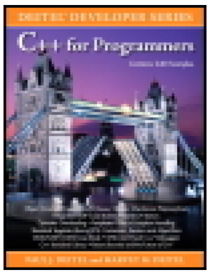


**C++ for Programmers: Deitel® Developer Series**
by Paul J. Deitel - Deitel & Associates, Inc.; Harvey M. Deitel - Deitel & Associates, Inc.

Publisher: **Prentice Hall**
Pub Date: **January 23, 2009**
Print ISBN-10: **0-13-700130-4**
Print ISBN-13: **978-0-13-700130-9**
Web ISBN-10: **0-13-701849-5**
Web ISBN-13: **978-0-13-701849-9**
Pages: **1056**

## Overview

***PRACTICAL, EXAMPLE-RICH COVERAGE OF:***

- Classes, Objects, Encapsulation, Inheritance, Polymorphism
- Integrated OOP Case Studies: Time, GradeBook, Employee
- Industrial-Strength, 95-Page OOD/UML® 2 ATM Case Study
- Standard Template Library (STL): Containers, Iterators and Algorithms
- I/O, Types, Control Statements, Functions
- Arrays, Vectors, Pointers, References
- String Class, C-Style Strings
- Operator Overloading, Templates
- Exception Handling, Files
- Bit and Character Manipulation
- Boost Libraries and the Future of C++
- GNU™ and Visual C++® Debuggers
- And more…

***VISIT WWW.DEITEL.COM***

- For information on Deitel® Dive-Into® Series corporate training courses offered at customer sites worldwide (or write to deitel@deitel.com)
- Download code examples
- Check out the growing list of programming, Web 2.0 and software-related Resource Centers
- To receive updates for this book, subscribe to the free *DEITEL® BUZZ ONLINE* e-mail newsletter at www.deitel.com/newsletter/subscribe.html
- Read archived issues of the *DEITEL® BUZZ ONLINE*

***The professional programmer's DEITEL® guide to C++ and object-oriented application development***

Written for programmers with a background in high-level language programming, this book applies the Deitel signature live-code approach to teaching programming and explores the C++ language and C++ Standard Libraries in depth. The book presents the concepts in the context of fully tested programs, complete with syntax shading, code highlighting, code walkthroughs and program outputs. The book features 240 C++ applications with over 15,000 lines of proven C++ code, and hundreds of tips that will help you build robust applications.

Start with an introduction to C++ using an early classes and objects approach, then rapidly move on to more advanced topics, including templates, exception handling, the Standard Template Library (STL) and selected features from the Boost libraries. You'll enjoy the Deitels' classic treatment of object-oriented programming and the OOD/UML® 2 ATM case study, including a complete C++ implementation. When you're finished, you'll have everything you need to build object-oriented C++ applications.

The ***DEITEL® Developer Series*** is designed for practicing programmers. The series presents focused treatments of emerging technologies, including C++, .NET, Java™, web services, Internet and web development and more.

***PRE-PUBLICATION REVIEWER TESTIMONIALS***

"An excellent 'objects first' coverage of C++. The example-driven presentation is enriched by the optional UML case study that contextualizes the material in an ongoing software engineering project." —Gavin Osborne, Saskatchewan Institute of Applied Science and Technology

"Introducing the UML early on is a great idea." —Raymond Stephenson, Microsoft

"Good use of diagrams, especially of the activation call stack and recursive functions." —Amar Raheja, California State Polytechnic University, Pomona

"Terrific discussion of pointers—probably the best I have seen." —Anne B. Horton, Lockheed Martin

"Great coverage of polymorphism and how the compiler implements polymorphism 'under the hood.'" —Ed James-Beckham, Borland

"The Boost/C++0x chapter will get you up and running quickly with the memory management and regular expression libraries, plus whet your appetite for new C++ features being standardized." —Ed Brey, Kohler Co.

"Excellent introduction to the Standard Template Library (STL). The best book on C++ programming!" —Richard Albright, Goldey-Beacom College

"Just when you think you are focused on learning one topic, suddenly you discover you've learned more than you expected." —Chad Willwerth, University of Washington, Tacoma

"The most thorough C++ treatment I've seen. Replete with real-world case studies covering the full software development lifecycle. Code examples are extraordinary!" —Terrell Hull, Logicalis Integration Solutions/

## Copyright

*Library of Congress Cataloging-in-Publication Data*

*On file*

ISBN-13: 978-0-13-700130-9

Text printed in the United States on recycled paper at R.R . Donnelley in Crawfordsville, Indiana.

First printing, January 2009

### Trademarks

## Dedication

*In memory of Joseph Weizenbaum MIT Professor Emeritus of Computer Science: For making us think.*

*—Paul and Harvey Deitel*

Ruby on Rails
Silverlight
Visual Basic
Visual C++
Visual Studio Team System
Web 3D Technologies
Web Services
Windows Presentation Foundation
XHTML
XML

***Games and Game Programming***
Computer Game Programming
Computer Games
Mobile Gaming
Sudoku

***Internet Business***
Affiliate Programs
Competitive Analysis
Facebook Social Ads
Google AdSense
Google Analytics
Google Services
Internet Advertising
Internet Business Initiative
Internet Public Relations
Link Building
Location-Based Services
Online Lead Generation
Podcasting
Search Engine Optimization
Selling Digital Content
Sitemaps
Web Analytics
Website Monetization
YouTube and AdSense

***Java***
Java
Java Certification and Assessment Testing
Java Design Patterns
Java EE 5
Java SE 6
Java SE 7 (Dolphin) Resource Center
JavaFX

***Microsoft***
ASP.NET
ASP.NET 3.5
ASP.NET Ajax
C#
DotNetNuke (DNN)
Internet Explorer 7 (IE7)
Microsoft LINQ
.NET
.NET 3.0

.NET 3.5
SharePoint
Silverlight
Visual Basic
Visual C++
Visual Studio Team System
Windows Presentation Foundation
Windows Vista
Microsoft Popfly

***Open Source & LAMP Stack***
Apache
DotNetNuke (DNN)
Eclipse
Firefox
Linux
MySQL
Open Source
Perl
PHP
Python
Ruby

***Software***
Apache
DotNetNuke (DNN)
Eclipse
Firefox
Internet Explorer 7 (IE7)
Linux
MySQL
Open Source
Search Engines
SharePoint
Skype
Web Servers
Wikis
Windows Vista

***Web 2.0***
Alert Services
Attention Economy
Blogging
Building Web Communities
Community Generated Content
Facebook Developer Platform
Facebook Social Ads
Google Base
Google Video
Google Web Toolkit (GWT)
Internet Video
Joost
Location-Based Services
Mashups
Microformats
Recommender Systems
RSS

Social Graph
Social Media
Social Networking
Software as a Service (SaaS)
Virtual Worlds
Web 2.0
Web 3.0
Widgets

***Dive Into Web 2.0 eBook***
Web 2 eBook

***Other Topics***
Computer Games
Computing Jobs
Gadgets and Gizmos
Ring Tones
Sudoku

## Preface

*"The chief merit of language is clearness . . ."*

*—Galen*

Welcome to *C++ for Programmers* ! At Deitel & Associates, we write programming language professional books and textbooks for publication by Prentice Hall, deliver programming languages corporate training courses at organizations worldwide and develop Internet businesses. This book is intended for programmers who do not yet know C++, and may or may not know object-oriented programming.

### Features of *C++ for Programmers*

The Tour of the Book section of this Preface will give you a sense of *C++ for Programmers'* coverage of C++ and object-oriented programming. Here's some key features of the book:

- ***Early Classes and Objects Approach.*** We present object-oriented programming, where appropriate, from the start and throughout the text.
- ***Integrated Case Studies.*** We develop the GradeBook class in Chapters 3–7, the Time class in several sections of Chapters 9–10, the Employee class in Chapters 12–13, and the optional OOD/UML ATM case study in Chapters 1–7, 9, 13 and Appendix E.
- ***Unified Modeling Language™ 2 (UML 2).*** The Unified Modeling Language (UML) has become the preferred graphical modeling language for designers of object-oriented systems. We use UML class diagrams to visually represent classes and their inheritance relationships, and we use UML activity diagrams to demonstrate the flow of control in each of C++'s control statements. We emphasize the UML in the optional OOD/UML ATM case study
- ***Optional OOD/UML ATM Case Study.*** We introduce a concise subset of the UML 2, then guide you through a first design experience intended for the novice object-oriented designer/programmer. The case study was reviewed by a distinguished team of OOD/UML industry professionals and academics. The case study is not an exercise; rather, it's a fully developed end-to-end learning experience that concludes with a detailed walkthrough of the complete 877-line C++ code implementation. We take a detailed tour of the nine sections of this case study later in the Preface.
- ***Function Call Stack Explanation.*** In Chapter 6 , we provide a detailed discussion (with illustrations) of the function call stack and activation records to explain how C++ is able to keep track of which function is currently executing, how automatic variables of functions are maintained in memory and how a function knows where to return after it completes execution.
- ***Class string.*** We use class string instead of C-like pointer-based char * strings for most string manipulations throughout the book. We include discussions of char * strings in Chapters 8, 10, 11 and 19 to give you practice with pointer manipulations, to illustrate dynamic memory allocation with new and delete , to build our own String class, and to prepare you for working with char * strings in C and C++ legacy code.
- ***Class Template vector.*** We use class template vector instead of C-like pointer-based array manipulations throughout the book. However, we begin by discussing C-like pointer-based arrays in Chapter 7 to prepare you for working with C and C++ legacy code and to use as a basis for building our own customized Array class in Chapter 11.
- ***Treatment of Inheritance and Polymorphism.*** Chapters 12–13 include an Employee class hierarchy that makes the treatment of inheritance and polymorphism clear and accessible for programmers who are new to OOP.
- ***Discussion and Illustration of How Polymorphism Works "Under the Hood."*** Chapter 13 contains a detailed diagram and explanation of how C++ can implement polymorphism, virtual functions and dynamic binding

internally. This gives you a solid understanding of how these capabilities really work. More importantly, it helps you appreciate the overhead of polymorphism—in terms of additional memory consumption and processor time. This helps you determine when to use polymorphism and when to avoid it.

- ***Standard Template Library (STL).*** This might be one of the most important topics in the book in terms of software reuse. The STL defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures. Chapter 20 introduces the STL and discusses its three key components—containers, iterators and algorithms. Using STL components provides tremendous expressive power, often reducing many lines of non-STL code to a single statement.

- ***ISO/IEC C++ Standard Compliance.*** We have audited our presentation against the most recent ISO/IEC C++ standard document for completeness and accuracy. [*Note:* A PDF copy of the C++ standard (document number INCITS/ISO/IEC 14882-2003) can be purchased at webstore.ansi.org/ansidocstore/default.asp.]

- ***Future of C++.*** In Chapter 21, which considers the future of C++, we introduce the Boost C++ Libraries, Technical Report 1 (TR1) and C++0x. The free Boost open source libraries are created by members of the C++ community. Technical Report 1 describes the proposed changes to the C++ Standard Library, many of which are based on current Boost libraries. The C++ Standards Committee is revising the C++ Standard. The main goals for the new standard are to make C++ easier to learn, improve library building capabilities, and increase compatibility with the C programming language. The last standard was published in 1998. Work on the new standard, currently referred to as C++0x, began in 2003. The new standard is likely to be released in 2009. It will include changes to the core language and, most likely, many of the libraries in TR1. We overview the TR1 libraries and provide code examples for the "regular expression" and "smart pointer" libraries.

- ***Debugger Appendices.*** We include two Using the Debugger appendices—Appendix G, Using the Visual Studio Debugger, and Appendix H, Using the GNU C++ Debugger.

- ***Code Testing on Multiple Platforms.*** We tested the code examples on various popular C++ platforms. For the most part, the book's examples port easily to standard-compliant compilers.

- ***Errors and Warnings Shown for Multiple Platforms.*** For programs that intentionally contain errors to illustrate a key concept, we show the error messages that result on several popular platforms.

All of this was carefully reviewed by distinguished industry developers and academics. We believe that this book will provide you with an informative, interesting, challenging and entertaining C++ educational experience.

As you read this book, if you have questions, send an e-mail to deitel@deitel.com; we'll respond promptly. For updates on this book and the status of all supporting C++ software, and for the latest news on all Deitel publications and services, visit www.deitel.com. Sign up at www.deitel.com/newsletter/subscribe.html for the free *Deitel*® *Buzz Online* e-mail newsletter and check out our growing list of C++ and related Resource Centers at www.deitel.com/ResourceCenters.html. Each week we announce our latest Resource Centers in the newsletter.

**Learning Features**

*C++ for Programmers* contains a rich collection of examples. The book concentrates on the principles of good software engineering and stresses program clarity. We teach by example. We are educators who teach programming languages in industry classrooms worldwide. The Deitels have taught courses at all levels to government, industry, military and academic clients of Deitel & Associates.

Live-Code Approach. *C++ for Programmers* is loaded with "live-code" examples—by this we mean that each new concept is presented in the context of a complete working C++ application that is immediately followed by one or more actual executions showing the program's inputs and outputs.

Syntax Shading. We syntax-shade all the C++ code, similar to the way most C++ integrated development environments (IDEs) and code editors syntax-color code. This greatly improves code readability—an especially important goal, given that this book contains over 15,500 lines of code. Our syntax-shading conventions are as follows:

comments appear in italic

keywords appear in bold italic

errors and ASP.NET script delimiters appear in bold black

constants and literal values appear in bold gray

all other code appears in plain black

Code Highlighting. We place white rectangles around the key code segments in each program.

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 for *C++ for Programmers* are available for download from www.deitel.com/books/cppfp/.

Objectives. Each chapter begins with a statement of objectives. This lets you know what to expect and gives you an opportunity, after reading the chapter, to determine if you've met the objectives.

Quotations. The learning objectives are followed by quotations. Some are humorous; some are philosophical; others offer interesting insights. We hope that you enjoy relating the quotations to the chapter material.

Outline. The chapter outlines help you approach the material in a top-down fashion, so you can anticipate what is to come and set a comfortable and effective learning pace.

Illustrations/Figures. Abundant charts, tables, line drawings, programs and program output are included. We model the flow of control in control statements with UML activity diagrams. UML class diagrams model the fields, constructors and methods of classes. We make extensive use of six major UML diagram types in the optional OOD/UML 2 ATM case study.

Programming Tips. We include 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 seven decades of programming experience—they provide a basis on which to build good software.

Good Programming Practice

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 the same mistakes.*

Error-Prevention Tip

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

Performance Tip

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

Portability Tip

*We include* Portability Tips *to help you write code that will run on a variety of platforms and to explain how C++ achieves its high degree of portability.*

Software Engineering Observation

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

Wrap-Up Section. Each of the chapters ends with a brief "wrap-up" section that recaps the chapter content and transitions to the next chapter.

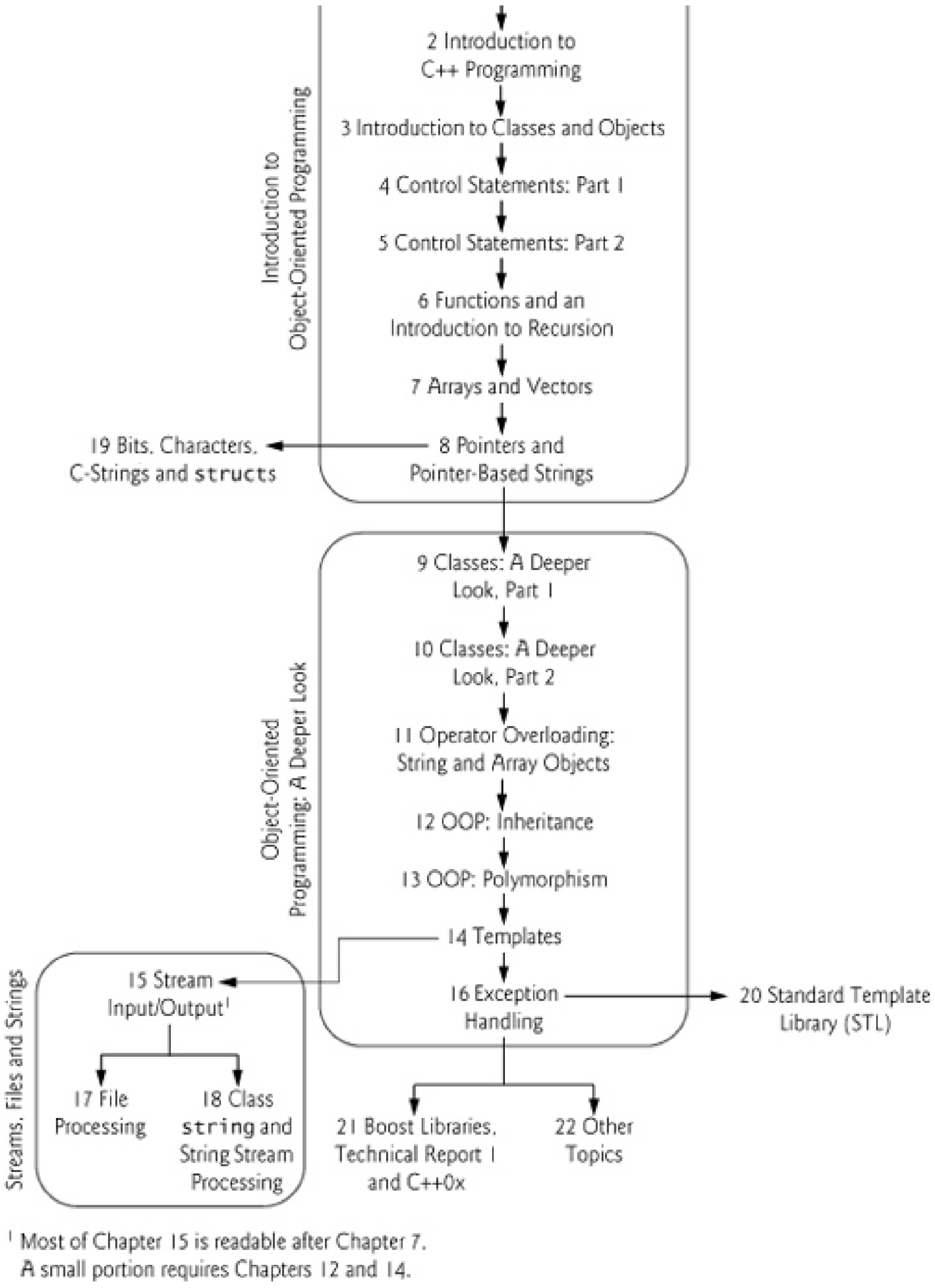
Thousands of Index Entries. We've included an extensive index which is especially useful when you use the book as a reference.

"Double Indexing" of C++ Live-Code Examples. For every source-code program in the book, we index the figure caption both alphabetically and as a subindex item under "Examples." This makes it easier to find examples using particular features.

**Tour of the Book**

You'll now take a tour of the C++ capabilities you'll study in C++ *for Programmers*. Figure 1 illustrates the dependencies among the chapters. We recommend studying the topics in the order indicated by the arrows, though other orders are possible.

**Fig. 1. *C++ for Programmers* chapter dependency chart.**


2 Introduction to C++ Programming
3 Introduction to Classes and Objects
4 Control Statements: Part 1
5 Control Statements: Part 2
6 Functions and an Introduction to Recursion
7 Arrays and Vectors
8 Pointers and Pointer-Based Strings
19 Bits, Characters, C-Strings and structs
Introduction to Object-Oriented Programming
9 Classes: A Deeper Look, Part 1
10 Classes: A Deeper Look, Part 2
11 Operator Overloading: String and Array Objects
12 OOP: Inheritance
13 OOP: Polymorphism
14 Templates
16 Exception Handling
20 Standard Template Library (STL)
Object-Oriented Programming: A Deeper Look
15 Stream Input/Output[1]
17 File Processing
18 Class string and String Stream Processing
Streams, Files and Strings
21 Boost Libraries, Technical Report 1 and C++0x
22 Other Topics
[1] Most of Chapter 15 is readable after Chapter 7. A small portion requires Chapters 12 and 14.

***Chapter 1, Introduction***, discusses the origin of the C++ programming language, and introduces a typical C++ programming environment. We walk through a "test drive" of a typical C++ application on the Windows and Linux platforms. We also introduce basic object technology concepts and terminology, and the Unified Modeling Language.

***Chapter 2*, Introduction to C++ Programming**, provides a lightweight introduction to programming applications in C++. The programs in this chapter illustrate how to display data on the screen, obtain data from the keyboard, make decisions and perform arithmetic operations.

***Chapter 3*, Introduction to Classes and Objects**, provides a friendly early introduction to classes and objects. We introduce classes, objects, member functions, constructors and data members using a series of simple real-world examples. We develop a well-engineered framework for organizing object-oriented programs in C++. We motivate the notion of classes with a simple example. Then we present a carefully paced sequence of seven 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 an instructor can use to maintain student test scores. This case study is enhanced over the next several chapters, culminating with the version presented in Chapter 7 . The GradeBook class case study describes how to define a class and how to use it to create an object. The case study discusses how to declare and define member functions to implement the class's behaviors, how to declare data members to implement the class's attributes and how to call an object's member functions to make them perform their tasks. We introduce C++ Standard Library class string and create string objects to store the name of the course that a GradeBook object represents. We explain the differences between data members of a class and local variables of a function, and how to use a constructor to ensure that an object's data is initialized when the object is created. We show how to promote software reusability by separating a class definition from the client code (e.g., function main ) that uses the class. We also introduce another fundamental principle of good software engineering—separating interface from implementation.

***Chapter 4*, Control Statements: Part 1**, focuses on the program-development process involved in creating useful classes. The chapter introduces some control statements for decision making (if and if...else) and repetition (while). We examine counter-controlled and sentinel-controlled repetition using the GradeBook ***class*** from Chapter 3 , and introduce C++'s increment, decrement and assignment operators. The chapter includes ***two enhanced versions of the GradeBook class***, each based on Chapter 3 's final version. The chapter uses simple UML activity diagrams to show the flow of control through each of the control statements.

***Chapter 5* , Control Statements: Part 2**, continues the discussion of C++ control statements with examples of the for repetition statement, the do...while repetition statement, the switch selection statement, the break statement and the continue statement. We create an ***enhanced version of class GradeBook*** that uses a switch statement to count the number of A, B, C, D and F grades entered by the user. The chapter also a discusses logical operators.

***Chapter 6*, Functions and an Introduction to Recursion**, takes a deeper look inside objects and their member functions. We discuss C++ Standard Library functions and examine more closely how you can build your own functions. The chapter's first example continues the GradeBook ***class case study*** with an example of a function with multiple parameters. You may enjoy the chapter's treatment of random numbers and simulation, and the discussion of the dice game of craps, which makes elegant use of control statements. The chapter discusses the so-called "C++ enhancements to C," including inline functions, reference parameters, default arguments, the unary scope resolution operator, function overloading and function templates. We also present C++'s call-by-value and call-by-reference capabilities. The header files table introduces many of the header files that you'll use throughout the book. We discuss the function call stack and activation records to explain how C++ keeps track of which function is currently executing, how automatic variables of functions are maintained in memory and how a function knows where to return after it completes execution. The chapter then offers a solid introduction to recursion.

***Chapter 7*, Arrays and Vectors**, explains how to process lists and tables of values. We discuss the structuring of data in arrays of data items of the same type and demonstrate how arrays facilitate the tasks performed by objects. The early parts of this chapter use C-style, pointer-based arrays, which, as you'll see in Chapter 8 , can be treated as pointers to the array contents in memory. We then present arrays as full-fledged objects, introducing the C++ Standard Library vector class template—a robust array data structure. The chapter presents numerous examples of both one-dimensional arrays and two-dimensional arrays. Examples in the chapter investigate various common array manipulations, printing bar charts, sorting data and passing arrays to functions. The chapter includes the ***final two GradeBook case study sections***, in which we use arrays to store student grades for the duration of a program's execution. Previous versions of the class process a set of grades entered by the user, but do not maintain the individual grade values in data members of the class. In this chapter, we use arrays to enable an object of the GradeBook class to maintain a set of grades in memory, thus eliminating the need to repeatedly input the same set of grades. The first version of the class stores the grades in a one-dimensional array. The second version uses a two-dimensional array to store the grades of a number of students on multiple exams in a semester.

Another key feature of this chapter is the discussion of elementary sorting and searching techniques.

***Chapter 8*, Pointers and Pointer-Based Strings**, presents one of the most powerful features of the C++ language—pointers. The chapter provides detailed explanations of pointer operators, call by reference, pointer expressions, pointer arithmetic, the relationship between pointers and arrays, arrays of pointers and pointers to functions. We demonstrate how to use const with pointers to enforce the principle of least privilege to build more robust software. We discuss using the sizeof operator to determine the size of a data type or data items in bytes during program compilation. There is an intimate relationship between pointers, arrays and C-style strings in C++, so we introduce basic C-style string-manipulation concepts and discuss some of the most popular C-style string-handling functions, such as getline (input a line of text), strcpy and strncpy (copy a string), strcat and strncat (concatenate two strings), strcmp and strncmp (compare two strings), strtok ("tokenize" a string into its pieces) and strlen (return the length of a string). We frequently use string objects (introduced in Chapter 3) in place of C-style, char * pointer-based strings. However, we include char * strings in Chapter 8 to help you master pointers and prepare for the professional world in which you'll see a great deal of C legacy code that has been implemented over the last three decades. In C and "raw C++" arrays and strings are pointers to array and string contents in memory (even function names are pointers).

***Chapter 9* , Classes: A Deeper Look, Part 1**, continues our discussion of object-oriented programming. This chapter uses a rich Time class case study to illustrate accessing class members, separating interface from implementation, using access functions and utility functions, initializing objects with constructors, destroying objects with destructors, assignment by default memberwise copy and software reusability. We discuss the order in which constructors and destructors are called during the lifetime of an object. A modification of the Time case study demonstrates the problems that can occur when a member function returns a reference to a private data member, which breaks the encapsulation of the class.

***Chapter 10*, Classes: A Deeper Look, Part 2**, continues the study of classes and presents additional object-oriented programming concepts. The chapter discusses declaring and using constant objects, constant member functions, composition—the process of building classes that have objects of other classes as members, friend functions and friend classes that have special access rights to the private and protected members of classes, the this pointer, which enables an object to know its own address, dynamic memory allocation, static class members for containing and manipulating class-wide data, examples of popular abstract data types (arrays, strings and queues), container classes and iterators. In our discussion of const objects, we mention keyword mutable which is used in a subtle manner to enable modification of "non-visible" implementation in const objects. We discuss dynamic memory allocation using new and delete. When new fails, the program terminates by default because new "throws an exception" in standard C++. We motivate the discussion of static class members with a video-game-based scenario. We emphasize how important it is to hide implementation details from clients of a class; then, we discuss proxy classes, which provide a means of hiding implementation (including the private data in class headers) from clients of a class.

***Chapter 11*, Operator Overloading; String and Array Objects**, presents one of the most popular topics in our C++ courses. Professionals really enjoy this material. They find it a perfect complement to the detailed discussion of crafting valuable classes in Chapters 9 and 10 . Operator overloading enables you to tell the compiler how to use existing operators with objects of new types. C++ already knows how to use these operators with built-in types, such as integers, floats and characters. But suppose that we create a new String class—what would the plus sign mean when used between String objects? Many programmers use plus (+) with strings to mean concatenation. In Chapter 11 , you'll see how to "overload" the plus sign, so when it is written between two String objects in an expression, the compiler will generate a function call to an "operator function" that will concatenate the two String s. The chapter discusses the fundamentals of operator overloading, restrictions in operator overloading, overloading with class member functions vs. with nonmember functions, overloading unary and binary operators and converting between types. Chapter 11 features a collection of substantial case studies including an Array class, a String class and a Date class. Using operator overloading wisely helps you add extra "polish" to your classes.

***Chapter 12*, Object-Oriented Programming: Inheritance**, introduces one of the most fundamental capabilities of object-oriented programming languages—inheritance: a form of software reusability in which new classes are developed quickly and easily by absorbing the capabilities of existing classes and adding appropriate new capabilities. In the context of an Employee ***hierarchy*** case study, this chapter presents a five-example sequence demonstrating private data, protected data and good software engineering with inheritance. The chapter discusses the notions of base classes and derived classes, protected members, public inheritance, protected inheritance, private inheritance, direct base classes, indirect base classes, constructors and destructors in base classes and derived classes, and software engineering with inheritance. The chapter also compares inheritance (the *is-a* relationship) with composition (the *has-a* relationship) and introduces the *uses-a* and *knows-a* relationships.

***Chapter 13*, *Object-Oriented Programming: Polymorphism***, deals with another fundamental capability of object-oriented programming: polymorphic behavior. Chapter 13 builds on the inheritance concepts presented in Chapter 12 and focuses on the relationships among classes in a class hierarchy and the powerful processing capabilities that these relationships enable. When many classes are related to a common base class through inheritance, each derived-class object may be treated as a base-class object. This enables programs to be written in a simple and general manner independent of the specific types of the derived-class objects. New kinds of objects can be handled by the same program, thus making systems more extensible. The chapter discusses the mechanics of achieving polymorphic behavior via virtual functions. It distinguishes between abstract classes (from which objects cannot be instantiated) and concrete classes (from which objects can be instantiated). Abstract classes are useful for providing an inheritable interface to classes throughout the hierarchy. We include an illustration and a precise explanation of the *vtables* (virtual function tables) that the C++ compiler builds automatically to support polymorphism. To conclude, we introduce run-time type information (RTTI) and dynamic casting, which enable a program to determine an object's type at execution time, then act on that object accordingly.

***Chapter 14*, *Templates***, discusses one of C++'s more powerful software reuse features, namely templates. Function templates and class templates enable you to specify, with a single code segment, an entire range of related overloaded functions (called function template specializations) or an entire range of related classes (called class-template specializations). This technique is called generic programming. We might write a single class template for a stack class, then have C++ generate separate class-template specializations, such as a "stack-of-int" class, a "stack-of-float" class, a "stack-of-string" class and so on. The chapter discusses using type parameters, nontype parameters and default types for class templates. We also discuss the relationships between templates and other C++ features, such as overloading, inheritance, friends and static members. We greatly enhance the treatment of templates in our discussion of the Standard Template Library (STL) containers, iterators and algorithms in Chapter 20.

***Chapter 15*, *Stream Input/Output***, contains a comprehensive treatment of standard C++ input/output capabilities. This chapter discusses a range of capabilities sufficient for performing most common I/O operations and overviews the remaining capabilities. Many of the I/O features are object oriented. The various I/O capabilities of C++, including output with the stream insertion operator, input with the stream extraction operator, type-safe I/O, formatted I/O, unformatted I/O (for performance). Users can specify how to perform I/O for objects of user-defined types by overloading the stream insertion operator (<<) and the stream extraction operator (>>). C++ provides various stream manipulators that perform formatting tasks. This chapter discusses stream manipulators that provide capabilities such as displaying integers in various bases, controlling floating-point precision, setting field widths, displaying decimal point and trailing zeros, justifying output, setting and unsetting format state, setting the fill character in fields. We also present an example that creates user-defined output stream manipulators.

***Chapter 16*, *Exception Handling***, discusses how exception handling enables you to write programs that are robust, fault tolerant and appropriate for business-critical and mission-critical environments. The chapter discusses when exception handling is appropriate; introduces the basic capabilities of exception handling with try blocks, throw statements and catch handlers; indicates how and when to rethrow an exception; explains how to write an exception specification and process unexpected exceptions; and discusses the important ties between exceptions and constructors, destructors and inheritance. We discuss rethrowing an exception, and illustrate how new can fail when memory is exhausted. Many older C++ compilers return 0 by default when new fails. We show the new style of new failing by throwing a bad_alloc (bad allocation) exception. We illustrate how to use function set_new_handler to specify a custom function to be called to deal with memory-exhaustion situations. We discuss how to use the auto_ptr class template to delete dynamically allocated memory implicitly, thus avoiding memory leaks. To conclude this chapter, we present the Standard Library exception hierarchy.

***Chapter 17*, *File Processing***, discusses techniques for creating and processing both sequential files and random-access files. The chapter begins with an introduction to the data hierarchy from bits, to bytes, to fields, to records and to files. Next, we present the C++ view of files and streams. We discuss sequential files and build programs that show how to open and close files, how to store data sequentially in a file and how to read data sequentially from a file. We then discuss random-access files and build programs that show how to create a file for random access, how to read and write data to a file with random access and how to read data sequentially from a randomly accessed file. The case study combines the techniques of accessing files both sequentially and randomly into a complete transaction-processing program.

***Chapter 18*, *Class* string *and String Stream Processing***, The chapter discusses C++'s capabilities for inputting data from strings in memory and outputting data to strings in memory; these capabilities often are referred to as in-core formatting or string stream processing. Class string is a required component of the Standard Library. We preserved the treatment of C-like, pointer-based strings in Chapter 8 and later for several reasons. First, it strengthens your understanding of pointers. Second, for the next decade or so, C++ programmers will need to be able to read and modify the enormous amounts of C legacy code that has accumulated over the last quarter of a century—this code processes strings as pointers, as does a large portion of the C++ code that has been written in industry over the last many years. In Chapter 18 we discuss string assignment, concatenation and comparison. We show how to determine various string characteristics such as a string 's size, capacity and whether or not it is empty. We discuss how to resize a string . We consider the various "find" functions that enable us to find a substring in a string (searching the string either forwards or backwards), and we show how to find either the first occurrence or last occurrence of a character selected from a string of characters, and how to find the first occurrence or last occurrence of a character that is not in a selected string of characters. We show how to replace, erase and insert characters in a string and how to convert a string object to a C-style char * string.

***Chapter 19*, *Bits, Characters, C Strings and* structs**, begins by comparing C++ structures to classes, then defining and using C-like structures. We show how to declare structures, initialize structures and pass structures to functions. C++'s powerful bit-manipulation capabilities enable you to write programs that exercise lower-level hardware capabilities. This helps programs process bit strings, set individual bits and store information more compactly. Such capabilities, often found only in low-level assembly languages, are valued by programmers writing system software, such as operating systems and networking software. We discuss C-style char * string manipulation in Chapter 8 , where we present the most popular string-manipulation functions. In Chapter 19 , we continue our presentation of characters and C-style char * strings. We present the various character-manipulation capabilities of the <cctype> library—such as the ability to test a character to determine whether it is a digit, an alphabetic character, an alphanumeric character, a hexadecimal digit, a lowercase letter or an uppercase letter. We present the remaining string-manipulation functions of the various string-related libraries.

***Chapter 20*, *Standard Template Library (STL)***, discusses the STL's powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures. The STL offers proof of concept for generic programming with templates—introduced in Chapter 14 . This chapter discusses the STL's three key components—containers (templatized data structures), iterators and algorithms. Containers are data structures capable of storing objects of any type. We'll see that there are three container categories—first-class containers, adapters and near containers. Iterators, which have similar properties to those of pointers, are used by programs to manipulate the container elements. In fact, standard arrays can be manipulated as STL containers, using pointers as iterators. Manipulating containers with iterators is convenient and provides tremendous expressive power when combined with STL algorithms—in some cases, reducing many lines of code to a single statement. STL algorithms are functions that perform common data manipulations such as searching, sorting and comparing elements (or entire containers). Most of these use iterators to access container elements.

***Chapter 21*, *Boost Libraries, Technical Report 1 and C++0x***, focuses on the future of C++. We introduce the Boost Libraries, a collection of free, open source C++ libraries. The Boost libraries are carefully designed to work well with the C++ Standard Library. We then discuss Technical Report 1 (TR1), a description of proposed changes and additions to the Standard Library. Many of the libraries in TR1 were derived from libraries currently in Boost. The chapter briefly describes the TR1 libraries. We provide in-depth code examples for two of the most useful libraries, Boost.Regex and Boost.Smart_ptr. The Boost.Regex library provides support for regular expressions. We demonstrate how to use the library to search a string for matches to a regular expression, validate data, replace parts of a string and split a string into tokens. The Boost.Smart_ptr library provides smart pointers to help manage dynamically allocated memory. We discuss the two types of smart pointers included in TR1—shared_ptr and weak_ptr . We provide examples to demonstrate how these can be used to avoid common memory management errors. This chapter also discusses the upcoming release of the new standard for C++.

***Chapter 22*, *Other Topics***, is a collection of miscellaneous C++ topics. We discuss one more cast operator—const_cast. This operator, static_cast (Chapter 5), dynamic_cast (Chapter 13) and reinterpret_cast (Chapter 17 ), provide a more robust mechanism for converting between types than do the original cast operators C++ inherited from C (which are now deprecated). We discuss namespaces, a feature particularly crucial for software developers who build substantial systems. Namespaces prevent naming collisions, which can hinder such large software efforts. We discuss keyword mutable , which allows a member of a const object to be changed. Previously, this was accomplished by "casting away const-ness", which is considered a dangerous practice. We also discuss pointer-to-member operators .* and ->*, multiple inheritance (including the problem of "diamond inheritance") and virtual base classes.

***Appendix A*****, Operator Precedence and Associativity Chart***, presents the complete set of C++ operator symbols, in which each operator appears on a line by itself with its operator symbol, its name and its associativity.

***Appendix B, ASCII Character Set***. All the programs in this book use the ASCII character set, which is presented in this appendix.

***Appendix C, Fundamental Types***, lists C++'s fundamental types.

***Appendix D, Preprocessor***, discusses the preprocessor's directives. The appendix includes more complete information on the #include directive, which causes a copy of a specified file to be included in place of the directive before the file is compiled and the #define directive that creates symbolic constants and macros. The appendix explains conditional compilation, which enables you to control the execution of preprocessor directives and the compilation of program code. The # operator that converts its operand to a string and the ## operator that concatenates two tokens are discussed. The various predefined preprocessor symbolic constants (__LINE__, __FILE__, __DATE__, __STDC__, __TIME__ and __TIMESTAMP__) are presented. Finally, macro assert of the header file<cassert> is discussed, which is valuable in program testing, debugging, verification and validation.

***Appendix E, ATM Case Study Code***, contains the implementation of our case study on object-oriented design with the UML. This appendix is discussed in the tour of the case study (presented shortly).

***Appendix F, UML 2: Additional Diagram Types***, overviews the UML 2 diagram types that are not found in the OOD/UML Case Study.

***Appendix G, Using the Visual Studio Debugger***, demonstrates key features of the Visual Studio Debugger, which allows a programmer to monitor the execution of applications to locate and remove logic errors. The appendix presents step-by-step instructions, so you learn how to use the debugger in a hands-on manner.

***Appendix H, Using the GNU C++ Debugger***, demonstrates key features of the GNU C++ Debugger. The appendix presents step-by-step instructions, so you learn how to use the debugger in a hands-on manner.

***Bibliography***. The Bibliography lists many books and articles for further reading on C++ and object-oriented programming.

***Index***. The comprehensive index enables you to locate by keyword any term or concept throughout the text.

**Object-Oriented Design of an ATM with the UML: A Tour of the Optional Software Engineering Case Study**

In this section, we tour the book's optional case study of object-oriented design with the UML. This tour previews the contents of the nine Software Engineering Case Study sections (in Chapters 1–7, 9 and 13 ). After completing this case study, you'll be thoroughly familiar with a carefully developed and reviewed object-oriented design and implementation for a significant C++ application.

The design presented in the ATM case study was developed at Deitel & Associates, Inc. and scrutinized by a distinguished developmental review team of industry professionals and academics. Real ATM systems used by banks and their customers worldwide are based on more sophisticated designs that take into consideration many more issues than we have addressed here. Our primary goal throughout the design process was to create a simple design that would be clear to OOD and UML novices, while still demonstrating key OOD concepts and the related UML modeling techniques.

***Section 1.10, Software Engineering Case Study: Introduction to Object Technology and the UML***—introduces the object-oriented design case study with the UML. The section introduces the basic concepts and terminology of object technology, including classes, objects, encapsulation, inheritance and polymorphism. We discuss the history of the UML. This is the only required section of the case study.

***Section 2.7, (Optional) Software Engineering Case Study: Examining the ATM Requirements***

***Specification—***discusses a *requirements specification* that specifies the requirements for a system that we'll design and implement—the software for a simple automated teller machine (ATM). We investigate the structure and behavior of object-oriented systems in general. We discuss how the UML will facilitate the design process in subsequent Software Engineering Case Study sections by providing several additional types of diagrams to model our system. We discuss the interaction between the ATM system specified by the requirements specification and its user. Specifically, we investigate the scenarios that may occur between the user and the system itself—these are called *use cases*. We model these interactions, using *use case diagrams* of the UML.

***Section 3.11*****, *(Optional) Software Engineering Case Study: Identifying the Classes in the ATM Requirements Specification—*** begins to design the ATM system. We identify its classes, or "building blocks," by extracting the nouns and noun phrases from the requirements specification. We arrange these classes into a UML class diagram that describes the class structure of our simulation. The class diagram also describes relationships, known as *associations* , among classes.

***Section 4.11*****, *(Optional) Software Engineering Case Study: Identifying Class Attributes in the ATM System—***focuses on the attributes of the classes discussed in Section 3.11 . A class contains both *attributes* (data) and *operations* (behaviors). As we'll see in later sections, changes in an object's attributes often affect the object's behavior. To determine the attributes for the classes in our case study, we extract the adjectives describing the nouns and noun phrases (which defined our classes) from the requirements specification, then place the attributes in the class diagram we created in Section 3.11.

***Section 5.10*****, *(Optional) Software Engineering Case Study: Identifying Objects' States and Activities in the ATM System—*** discusses how an object, at any given time, occupies a specific condition called a *state*. A *state transition* occurs when that object receives a message to change state. The UML provides the *state machine diagram* , which identifies the set of possible states that an object may occupy and models that object's state transitions. An object also has an *activity*—the work it performs in its lifetime. The UML provides the *activity diagram* —a flowchart that models an object's activity. In this section, we use both types of diagrams to begin modeling specific behavioral aspects of our ATM system, such as how the ATM carries out a withdrawal transaction and how the ATM responds when the user is authenticated.

***Section 6.22*****, *(Optional) Software Engineering Case Study: Identifying Class Operations in the ATM System—*** identifies the operations, or services, of our classes. We extract from the requirements specification the verbs and verb phrases that specify the operations for each class. We then modify the class diagram of Section 3.11 to include each operation with its associated class. At this point in the case study, we will have gathered all information possible from the requirements specification. However, as future chapters introduce such topics as inheritance, we'll modify our classes and diagrams.

***Section 7.12*****, *(Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM System—*** provides a "rough sketch" of the model for our ATM system. In this section, we see how it works. We investigate the behavior of the simulation by discussing *collaborations* —messages that objects send to each other to communicate. The class operations that we discovered in Section 6.22 turn out to be the collaborations among the objects in our system. We determine the collaborations, then collect them into a *communication diagram* —the UML diagram for modeling collaborations. This diagram reveals which objects collaborate and when. We present a communication diagram of the collaborations among objects to perform an ATM balance inquiry. We then present the UML *sequence diagram* for modeling interactions in a system. This diagram emphasizes the chronological ordering of messages. A sequence diagram models how objects in the system interact to carry out withdrawal and deposit transactions.

***Section 9.11*****, *(Optional) Software Engineering Case Study: Starting to Program the Classes of the ATM System—*** takes a break from designing the system's behavior. We begin the implementation process to emphasize the material discussed in Chapter 9 . Using the UML class diagram of Section 3.11 and the attributes and operations discussed in Section 4.11 and Section 6.22 , we show how to implement a class in C++ from a design. We do not implement all classes—because we have not completed the design process. Working from our UML diagrams, we create code for the Withdrawal class.

***Section 13.10*****, *(Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System—*** continues our discussion of object-oriented programming. We consider inheritance—classes sharing common characteristics may inherit attributes and operations from a "base" class. In this section, we investigate how our ATM system can benefit from using inheritance. We document our discoveries in a class diagram that models inheritance relationships—the UML refers to these relationships as *generalizations*. We modify the class diagram of Section 3.11 by using inheritance to group classes with similar characteristics. This section concludes the design of the model portion of our simulation. We fully implement this model in 877 lines of C++ code in Appendix E.

***Appendix E*, ATM Case Study Code—** The majority of the case study involves designing the model (i.e., the data and logic) of the ATM system. In this appendix, we implement that model in C++. Using all the UML diagrams we created, we present the C++ classes necessary to implement the model. We apply the concepts of object-oriented design with the UML and object-oriented programming in C++ that you learned in the chapters. By the end of this appendix, you'll have completed the design and implementation of a real-world system, and should feel confident tackling larger systems.

***Appendix F*, UML 2: Additional Diagram Types—** Overviews the UML 2 diagram types that are not found in the OOD/UML Case Study.

### Compilers and Other Resources

Many C++ development tools are available. We wrote *C++ for Programmers* primarily using Microsoft's free Visual C++ Express Edition (www.microsoft.com/express/vc/) and the free GNU C++ at gcc.gnu.org, which is already installed on most Linux systems and can be installed on Mac OS X systems as well. Apple includes GNU C++ in their Xcode development tools, which Max OS X users can download from developer.apple.com/tools/xcode.

Additional resources and software downloads are available in our C++ Resource Center:

www.deitel.com/cplusplus/

and at the website for this book:

www.deitel.com/books/cppfp/

For a list of other C++ compilers that are available free for download, visit:

www.thefreecountry.com/developercity/ccompilers.shtml

www.compilers.net

### Warnings and Error Messages on Older C++ Compilers

The programs in this book are designed to be used with compilers that support standard C++. However, there are variations among compilers that may cause occasional warnings or errors. In addition, though the standard specifies various situations that require errors to be generated, it does not specify the messages that compilers should issue. Warnings and error messages vary among compilers.

Some older C++ compilers generate error or warning messages in places where newer compilers do not. Although most of the examples in this book will work with these older compilers, there are a few examples that need minor modifications to work with older compilers.

### Notes Regarding `using` Declarations and C Standard Library Functions

The C++ Standard Library includes the functions from the C Standard Library. According to the C++ standard document, the contents of the header files that come from the C Standard Library are part of the "`std`" namespace. Some compilers (old and new) generate error messages when `using` declarations are encountered for C functions.

### The Deitel Online Resource Centers

Our website provides Resource Centers (www.deitel.com/ResourceCenters.html) on various topics including programming languages, software, Web 2.0, Internet business and open source projects. The Resource Centers evolve out of the research we do for our books and business endeavors. We've found many (mostly free) exceptional resources including tutorials, documentation, software downloads, articles, blogs, videos, code samples, books, e-books and more. We help you wade through the vast amount of content on the Internet by providing links to the most valuable resources. Each week we announce our latest Resource Centers in our newsletter, the *Deitel® Buzz Online* (www.deitel.com/newsletter/subscribe.html ). The following Resource Centers may be of interest to you as you read *C++ for Programmers*:

- C++
- Visual C++ 2008
- C++ Boost Libraries
- C++ Game Programming
- Code Search Engines and Code Sites
- Computer Game Programming
- Computing Jobs
- Open Source
- Programming Projects
- Eclipse
- Linux
- .NET
- Windows Vista

### *Deitel® Buzz Online* Free E-mail Newsletter

Each week, the *Deitel® Buzz Online* newsletter announces our latest Resource Centers and includes commentary on industry trends and developments, links to free articles and resources from our published books and upcoming publications, product-release schedules, errata, challenges, anecdotes, information on our corporate instructor-led training courses and more. It's also a good way for you to keep posted about issues related to *C++ for Programmers*. To subscribe, visit

www.deitel.com/newsletter/subscribe.html

### Deitel® *LiveLessons* Self-Paced Video Training

The Deitel® *LiveLessons* products are self-paced video training. Each collection provides approximately 14+ hours of an instructor guiding you through programming training.

Your instructor, Paul Deitel, has personally taught programming at organizations ranging from IBM to Sun Microsystems to NASA. With the powerful videos included in our *LiveLessons* products, you'll learn at your own pace as Paul guides you through programming fundamentals, object-oriented programming and additional topics.

Deitel® *LiveLessons* products are based on its corresponding best-selling books and Paul's extensive experience presenting hundreds corporate training seminars. To view sample videos, visit

www.deitel.com/books/livelessons/

The *Java Fundamentals I and II LiveLessons* are available now. For announcements about upcoming Deitel *LiveLessons* products, including *C++ Fundamentals*, *C# 2008 Fundamentals* and *JavaScript Fundamentals* , subscribe to the *Deitel*® *Buzz Online* email newsletter at www.deitel.com/newsletter/subscribe.html.

## Deitel® *Dive-Into*® Series Instructor-Led Training

With our corporate, on-site, instructor-led *Dive-Into*® *Series* programming training courses (Fig. 2 ), professionals can learn C++, Java, C, Visual Basic, Visual C#, Visual C++, Python, and Internet and web programming from the internationally recognized professionals at Deitel & Associates, Inc. Our authors, teaching staff and contract instructors have taught over 1,000,000 people in more than 100 countries how to program in almost every major programming language through:

- *Deitel Developer Series* professional books
- *How to Program Series* textbooks
- University teaching
- Professional seminars
- Interactive multimedia CD-ROM Cyber Classrooms, Complete Training Courses and *LiveLessons* Video Training
- Satellite broadcasts

**Fig. 2. Deitel *Dive Into*® Series programming training courses.**

| **Deitel *Dive Into*® Series Programming Training Courses** |
|---|
| Java |
| Intro to Java for Non-Programmers: Part 1 |
| Intro to Java for Non-Programmers: Part 2 |
| Java for Visual Basic, C or COBOL Programmers |
| Java for C++ or C# Programmers |
| Advanced Java |
| C++ |
| Intro to C++ for Non-Programmers: Part 1 |
| Intro to C++ for Non-Programmers: Part 2 |
| C++ and Object Oriented Programming |
| C |
| Intro to C for Non-Programmers: Part 1 |
| Intro to C for Non-Programmers: Part 2 |
| C for Programmers |
| Visual C# 2008 |

## Acknowledgments

It is a great pleasure to acknowledge the efforts of many people whose names may not appear on the cover, but whose hard work, cooperation, friendship and understanding were crucial to the production of the book. Many people at Deitel & Associates, Inc. devoted long hours to this project—thanks especially to Abbey Deitel and Barbara Deitel.

We'd also like to thank one of the participants in our Honors Internship program who contributed to this publication—Greg Ayer, a computer science major at Northeastern University.

We are fortunate to have worked on this project with the talented and dedicated team of publishing professionals at Prentice

Hall. We appreciate the extraordinary efforts of Marcia Horton, Editorial Director of Prentice Hall's Engineering and Computer Science Division, Mark Taub, Editor-in-Chief of Prentice Hall Professional, and John Fuller, Managing Editor of Prentice Hall Professional. Carole Snyder, Lisa Bailey and Dolores Mars did a remarkable job recruiting the book's large review team and managing the review process. Sandra Schroeder designed the book's cover. Scott Disanno and Robert Engelhardt managed the book's production.

This book was adapted from our book *C++ How to Program*, *6/e* . We wish to acknowledge the efforts of our reviewers on that book. Adhering to a tight time schedule, they scrutinized the text and the programs, providing countless suggestions for improving the accuracy and completeness of the presentation.

### C++ How to Program, 6/e *Reviewers*

***Industry and Academic Reviewers:*** Dr. Richard Albright (Goldey-Beacom College), William B. Higdon (University of Indianapolis), Howard Hinnant (Apple), Anne B. Horton (Lockheed Martin), Terrell Hull (Logicalis Integration Solutions), Rex Jaeschke (Independent Consultant), Maria Jump (The University of Texas at Austin), Geoffrey S. Knauth (GNU), Don Kostuch (Independent Consultant), Colin Laplace (Freelance Software Consultant), Stephan T. Lavavej (Microsoft), Amar Raheja (California State Polytechnic University, Pomona), G. Anthony Reina (University of Maryland University College, Europe), Daveed Vandevoorde (C++ Standards Committee), Jeffrey Wiener (DEKA Research & Development Corporation, New Hampshire Community Technical College), and Chad Willwerth (University of Washington, Tacoma). ***Boost/C++Ox Reviewers:*** Edward Brey (Kohler Co.), Jeff Garland (Boost.org), Douglas Gregor (Indiana University), and Björn Karlsson (Author of *Beyond the C++ Standard Library: An Introduction to Boost*, Addison-Wesley/Readsoft, Inc.).

These reviewers scrutinized every aspect of the text and made countless suggestions for improving the accuracy and completeness of the presentation.

Well, there you have it! Welcome to the exciting world of C++ and object-oriented programming. We hope you enjoy this look at contemporary computer programming.

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

deitel@deitel.com

We'll respond promptly, and post corrections and clarifications on:

www.deitel.com/books/cppfp/

We hope you enjoy reading *C++ for Programmers* as much as we enjoyed writing it!

*Paul J. Deitel*

*Dr. Harvey M. Deitel*

## About the Authors

***Paul J. Deitel*** , CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT's Sloan School of Management, where he studied Information Technology. Through Deitel & Associates, Inc., he has delivered C++, Java, C, C# and Visual Basic courses to industry, government and military clients, including Cisco, IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA at the Kennedy Space Center, White Sands Missile Range, the National Severe Storm Laboratory, Rogue Wave Software, Boeing, Stratus, Hyperion Software, Adra Systems, Entergy, CableData Systems, Nortel Networks, Puma, iRobot, Invensys and many more. He has lectured on C++ and Java for the Boston Chapter of the Association for Computing Machinery, and on .NET technologies for ITESM in Monterrey, Mexico. He and his father, Dr. Harvey M. Deitel, are the world's best-selling programming language textbook authors.

***Dr. Harvey M. Deitel*** , Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has 47 years of academic and industry experience in the computer field. Dr. Deitel earned B.S. and M.S. degrees from the MIT and a Ph.D. from Boston University. He has 20 years of college teaching experience, including earning tenure and serving as the Chairman of the

Computer Science Department at Boston College before founding Deitel & Associates, Inc., with his son, Paul J. Deitel. He and Paul are the co-authors of several dozen books and multimedia packages and they are writing many more. The Deitels' texts have earned international recognition with translations published in Japanese, German, Russian, Spanish, Traditional Chinese, Simplified Chinese, Korean, French, Polish, Italian, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of professional seminars to major corporations, academic institutions, government organizations and the military.

**About Deitel & Associates, Inc**

Deitel & Associates, Inc., is an internationally recognized corporate training and content-creation organization specializing in computer programming languages, Internet and web software technology, object technology education and Internet business development through its Internet Business Initiative. The company provides instructor-led professional courses on major programming languages and platforms, such as C++, Java, C, C#, Visual C++, Visual Basic, XML, Perl, Python, object technology and Internet and web programming. The founders of Deitel & Associates, Inc., are Paul J. Deitel and Dr. Harvey M. De-itel. The company's clients include many of the world's largest companies, government agencies, branches of the military, and academic institutions. Through its 32-year publishing partnership with Prentice Hall, Deitel & Associates, Inc. publishes leading-edge programming professional books, textbooks, *LiveLessons* video courses, interactive multimedia *Cyber Classrooms* , web-based training courses and e-content for popular course management systems. Deitel & Associates, Inc., and the authors can be reached via e-mail at:

deitel@deitel.com

To learn more about Deitel & Associates, Inc., its publications and its *Dive-Into*® Series Corporate Training curriculum offered on-site at clients worldwide, visit:

www.deitel.com

and subscribe to the free *Deitel*® *Buzz Online* e-mail newsletter at:

www.deitel.com/newsletter/subscribe.html

Check out the growing list of online Deitel Resource Centers at:

www.deitel.com/resourcecenters.html

Individuals wishing to purchase Deitel publications can do so through:

www.deitel.com/books/index.html

Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Prentice Hall. For more information, visit

www.prenhall.com/mischtm/support.html#order

# Before You Begin

Please follow the instructions in this section to download the book's examples before you begin using this book.

## Downloading the *C++ for Programmers* Example Code

The examples for *C++ for Programmers* can be downloaded as a ZIP archive file from www.deitel.com/books/cppfp/. *After you register and log in*, click the link for the examples under Download Code Examples and Other Premium Content for Registered Users. Save the ZIP file to a location that you'll remember. Extract the example files to your hard disk using a ZIP file extractor program, such as WinZip (www.winzip.com).

## Installing/Choosing a Compiler

If you have a computer running Windows XP or Windows Vista, you can install Visual C++ Express (www.microsoft.com/express/vc/) to edit, compile, execute, debug and modify your programs. Follow the on-screen instructions to install Visual C++ Express. We recommend that you use the default installation options and select the option to install the documentation as well.

If your computer has Linux or Mac OS X, then you probably already have the GNU C++ command-line compiler installed. There are many other C++ compilers and IDEs. We provide links to various free C++ development tools for Windows, Linux and Mac OS X platforms in our C++ Resource Center at

www.deitel.com/cplusplus/

This Resource Center also links to online tutorials that will help you get started with various C++ development tools.

You are now ready to begin using *C++ for Programmers*. We hope you enjoy the book! If you have any questions, please feel free to email us at deitel@deitel.com. We'll respond promptly.

# 1. Introduction

**Objectives**

In this chapter you'll learn:

- Object-technology concepts, such as classes, objects, attributes, behaviors, encapsulation and inheritance.
- A typical C++ program development environment.
- The history of the industry-standard object-oriented system modeling language, the UML.
- The history of the Internet and the World Wide Web, and the Web 2.0 phenomenon.
- To test-drive C++ applications in two popular C++ environments—GNU C++ running on Linux and Microsoft's Visual C++® on Windows®.
- What open source is, and two popular C++ open source libraries—Ogre for graphics and game programming, and Boost for broadly enhancing the capabilities of the C++ Standard Library.

The chief merit of language is clearness.

—***Galen***

Our life is frittered away by detail. . . . Simplify, simplify.

—***Henry David Thoreau***

He had a wonderful talent for packing thought close, and rendering it portable.

—***Thomas B. Macaulay***

Man is still the most extraordinary computer of all.

—***John F. Kennedy***

**Outline**

## 1.1. Introduction

Welcome to C++! We've worked hard to create what we hope you'll find to be an informative, entertaining and challenging learning experience. C++ is a powerful computer programming language that is appropriate for experienced programmers to use in building substantial information systems. *C++ for Programmers*, is an effective learning tool for each of these audiences.

The book emphasizes achieving program clarity through the proven techniques of object-oriented programming. This is an "early classes and objects" book. We teach C++ features in the context of complete working C++ programs and show the outputs produced when those programs are run on a computer—we call this the live-code approach. You may download the example programs from www.deitel.com/books/cppfp/.

The early chapters introduce the fundamentals of C++, providing a solid foundation for the deeper treatment of C++ in the later chapters. Experienced programmers tend to read the early chapters quickly, then find the treatment of C++ in the remainder of the book rigorous and challenging.

C++ is one of today's most popular software development languages. This book discusses the version of C++ standardized in the United States through the American National Standards Institute (ANSI) and worldwide through the International Organization for Standardization (ISO).

To keep up to date with C++ developments at Deitel & Associates, please register for our free e-mail newsletter, the *Deitel® Buzz Online,* at

www.deitel.com/newsletter/subscribe.html

Please check out our growing list of C++ and related Resource Centers at

www.deitel.com/ResourceCenters.html

Some Resource Centers that will be valuable to you as you read this book are C++, C++ Game Programming, C++ Boost Libraries, Code Search Engines and Code Sites, Computer Game Programming, Programming Projects, Eclipse, Linux, Open Source and Windows Vista. Each week we announce our latest Resource Centers in the newsletter. Errata and updates for this book are posted at www.deitel.com/books/cppfp/.

You are embarking on a challenging and rewarding path. As you proceed, if you have any questions, please send e-mail to deitel@deitel.com. We'll respond promptly. We hope that you'll enjoy learning with *C++ for Programmers*.

## 1.2. History of C and C++

C++ evolved from C, which evolved from two previous programming languages, BCPL and B. BCPL was developed in 1967 by Martin Richards as a language for writing operating systems software and compilers for operating systems. Ken Thompson modeled many features in his language B after their counterparts in BCPL and he used B to create early versions of the UNIX operating system at Bell Laboratories in 1970.

The C language was evolved from B by Dennis Ritchie at Bell Laboratories. C uses many important concepts of BCPL and B. C initially became widely known as the development language of the UNIX operating system. Today, most operating systems are written in C and/or C++. C is available for most computers and is hardware independent. With careful design, it is possible to write C programs that are portable to most computers.

The widespread use of C with various hardware platforms unfortunately led to many variations. This was a serious problem for program developers, who needed to write portable programs that would run on several platforms. 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*.

C99 is the latest C standard. It was developed to evolve the C language to keep pace with today's powerful hardware and with increasingly demanding user requirements. The C99 Standard is more capable (than earlier C Standards) of competing with languages like Fortran for mathematical applications. C99 capabilities include the long long type for 64-bit machines, complex numbers for engineering applications and greater support of floating-point arithmetic. C99 also makes C more consistent with C++ by enabling polymorphism through type-generic mathematical functions and through the creation of a defined boolean type. For more information on C and C99, see our book *C How to Program, Fifth Edition* and our C Resource Center (located at www.deitel.com/C/).

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

You'll be introduced to the basic concepts and terminology of object technology in Section 1.10. Objects are essentially reusable software components that model items in the real world. Software developers are discovering that a modular, object-oriented design and implementation approach can make them much more productive than can previous popular programming techniques. Object-oriented programs are easier to understand, correct and modify. You'll begin developing customized, reusable classes and objects in Chapter 3 , Introduction to Classes and Objects. This book is object oriented, where appropriate, from the start and throughout the text. This gets you "thinking about objects" immediately and mastering these concepts more completely.

We also provide an optional automated teller machine (ATM) case study in the Software Engineering Case Study sections of Chapters 1–7, 9 and 13, and Appendix E, 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 first design experience intended for the novice object-oriented designer/programmer. Our goal is to help you develop an object-oriented design to complement the object-oriented programming concepts you learn in this chapter and begin implementing in Chapter 3.

## 1.3. C++ Standard Library

C++ programs consist of pieces calledclasses and functions. You can program each piece that you may need to form a C++ program. However, most C++ programmers take advantage of the rich collections of existing 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. Throughout the book, we discuss many of these classes and functions. P. J. Plauger's book, *The Standard C Library* (Upper Saddle River, NJ: Prentice Hall PTR, 1992), is a must read for programmers who need a deep understanding of the ANSI C library functions that are included in C++, how to implement them and how to use them to write portable code. The standard class libraries generally are provided by compiler vendors. Many special-purpose class libraries are supplied by independent software vendors.

Software Engineering Observation 1.1

*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.*

We include many Software Engineering Observations throughout the book to explain concepts that affect and improve the overall architecture and quality of software systems. We also highlight other kinds of tips, including Good Programming Practices (to help you write programs that are clearer, more understandable, more maintainable and easier to test and debug—or remove programming errors),Common Programming Errors (problems to watch out for and avoid), Performance Tips (techniques for writing programs that run faster and use less memory), Portability Tips (techniques to help you write programs that can run, with little or no modification, on a variety of computers—these tips also include general observations about how C++ achieves its high degree of portability) and Error-Prevention Tips (techniques for removing programming errors—also known as bugs—from your programs and, more important, techniques for writing bug-free programs in the first place).

Performance Tip 1.1

*Using C++ Standard Library functions and classes instead of writing your own versions can improve program performance, because they are 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 are included in every C++ implementation.*

## 1.4. Key Software Trend: Object Technology

One of the authors, Harvey Deitel, remembers the great frustration felt in the 1960s by software development organizations, especially those working on large-scale projects. During his undergraduate years, he had the privilege of working summers at a leading computer vendor on the teams developing timesharing, virtual memory operating systems. This was a great experience for a college student. But, in the summer of 1967, reality set in when the company "decommitted" from producing as a commercial product the particular system on which hundreds of people had been working for many years. It was difficult to get this software right—software is "complex stuff."

Improvements to software technology did emerge, with the benefits of structured programming (and the related disciplines of structured systems analysis and design) being realized in the 1970s. Not until object-oriented programming became widely used in the 1990s, though, did software developers feel they had the necessary tools for making major strides in the software development process.

Actually, object technology dates back to the mid 1960s. The C++ programming language, developed at AT&T by Bjarne Stroustrup in the early 1980s, is based on two languages—C, which initially was developed at AT&T to implement the UNIX operating system in the early 1970s, and Simula 67, a simulation programming language developed in Europe and released in 1967. C++ absorbed the features of C and added Simula's capabilities for creating and manipulating objects. Neither C nor C++ was originally intended for wide use beyond the AT&T research laboratories. But grass roots support rapidly developed for each.

What are objects and why are they special? Actually, object technology is a packaging scheme that helps us create meaningful software units. These can be large and are highly focused on particular applications areas. There are date objects, time objects, paycheck objects, invoice objects, audio objects, video objects, file objects, record objects and so on. In fact, almost any noun can be reasonably represented as an object.

We live in a world of objects. Just look around you. There are cars, planes, people, animals, buildings, traffic lights, elevators and the like. Before object-oriented languages appeared, procedural programming languages (such as Fortran, COBOL, Pascal, BASIC and C) were focused on actions (verbs) rather than on things or objects (nouns). Programmers living in a world of objects programmed primarily using verbs. This made it awkward to write programs. Now, with the availability of popular object-oriented languages such as C++ and Java, programmers continue to live in an object-oriented world and can program in an object-oriented manner. This is a more natural process than procedural programming and has resulted in significant productivity gains.

A key problem with procedural programming is that the program units do not effectively mirror real-world entities, so these units are not particularly reusable. It's not unusual for programmers to "start fresh" on each new project and have to write similar software "from scratch." This wastes time and money, as people repeatedly "reinvent the wheel." With object technology, the software entities created (called classes ), if properly designed, tend to be reusable on future projects. Using libraries of reusable componentry can greatly reduce effort required to implement certain kinds of systems (compared to the effort that would be required to reinvent these capabilities on new projects).

Software Engineering Observation 1.2

*Extensive class libraries of reusable software components are available on the Internet. Many of these libraries are free.*

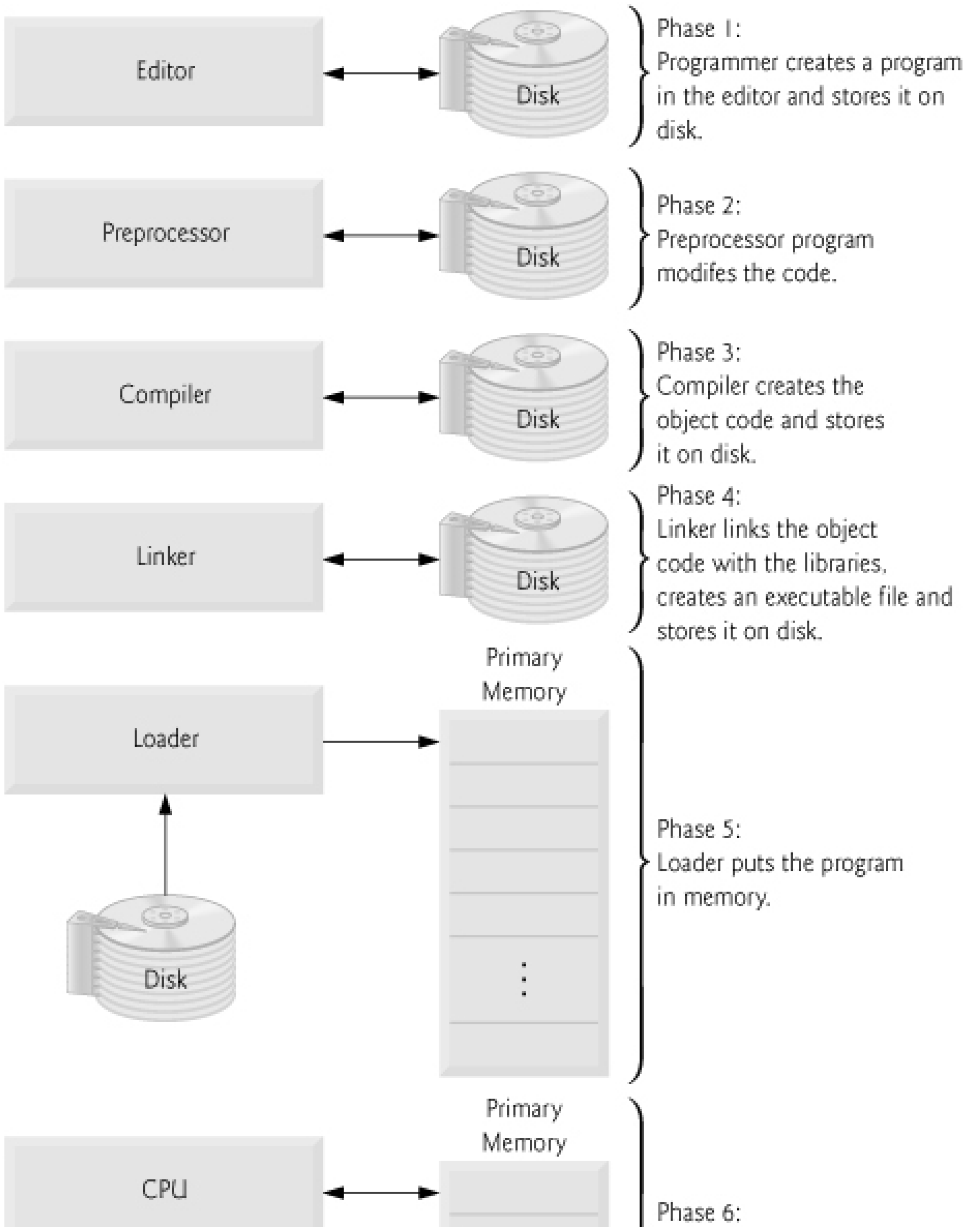
Some organizations report that the key benefit object-oriented programming gives them is not software reuse but, rather, that the software they produce is more understandable, better organized and easier to maintain, modify and debug. This can be significant, because perhaps as much as 80 percent of software costs are associated not with the original efforts to develop the software, but with the continued evolution and maintenance of that software throughout its lifetime.

Whatever the perceived benefits, it's clear that object-oriented programming will be the key programming methodology for the next several decades.

## 1.5. Typical C++ Development Environment

Let's consider the steps in creating and executing a C++ application using a C++ development environment (illustrated in Fig. 1.1 ). 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.

**Fig. 1.1. Typical C++ environment.**


Editor
Disk
Phase 1:
Programmer creates a program in the editor and stores it on disk.
Preprocessor
Disk
Phase 2:
Preprocessor program modifes the code.
Compiler
Disk
Phase 3:
Compiler creates the object code and stores it on disk.
Linker
Disk
Phase 4:
Linker links the object code with the libraries, creates an executable file and stores it on disk.
Primary Memory
Loader
Disk
Phase 5:
Loader puts the program in memory.
Primary Memory
CPU
Phase 6:

CPU takes each instruction and executes it, possibly storing new data values as the program executes.

### Phase 1: Creating a Program

Phase 1 consists of editing a file with an editor . You type a C++ program 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 filename extensions.

Two editors widely used on UNIX systems are vi and emacs . C++ software packages for Microsoft Windows such as Microsoft Visual C++ and cross-platform tools such as Eclipse 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.

### Phases 2 and 3: Preprocessing and Compiling a C++ Program

In phase 2, you give the command to compile the program. 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 preprocessor 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 preprocessor directives are discussed in the early chapters; a detailed discussion of preprocessor features appears in Appendix D , Preprocessor. In phase 3, the compiler translates the C++ program into object code.

### 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. 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 image (with no missing pieces). If the program compiles and links correctly, an executable image is produced.

### Phase 5: Loading

Before a program can be executed, it must first be placed in memory. 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.

### Phase 6: Execution

Finally, the computer executes the program.

## Problems That May Occur at Execution Time

Each of the preceding phases can fail because of various errors that we discuss throughout the book. This would cause the C++ program to display an error message. If this occurs, you would have to return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine that the corrections fix the problem(s).

Most programs in C++ input and/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. It is common for users to assign `cout` to a device other than the screen while keeping `cerr` assigned to the screen, so that normal outputs are separated from errors.

Common Programming Error 1.1

*Errors such as division by zero occur as a program runs, so they are 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. [*Note: *On some systems, divide-by-zero is not a fatal error. Please see your system documentation.]*

## 1.6. Notes About C++ and *C++ for Programmers*

Experienced C++ programmers sometimes take pride in being able to create weird, contorted, convoluted uses of the language. This is a poor programming practice. It makes programs more difficult to read, more likely to behave strangely, more difficult to test and debug, and more difficult to adapt to changing requirements. The following is our first Good Programming Practice.

Good Programming Practice 1.1

*Write your C++ programs in a simple and straightforward manner. This is sometimes referred to as KIS ("keep it simple"). Do not "stretch" the language by trying bizarre usages.*

You have heard that C and C++ are portable languages, and that programs written in C and C++ can run on many different computers. *Portability is an elusive goal.* The ANSI C standard document contains a lengthy list of portability issues, and complete books have been written that discuss portability.

Portability Tip 1.2

*Although it's possible to write portable programs, there are many problems among different C and C++ compilers and different computers that can make portability difficult to achieve. Writing programs in C and C++ does not guarantee portability. You often will need to deal directly with compiler and computer variations. As a group, these are sometimes called* ***platform*** *variations.*

We have audited our presentation against the ISO/IEC C++ standard document for completeness and accuracy. However, C++ is a rich language, and there are some features we have not covered. If you need additional technical details on C++, you may want to read the C++ standard document, which can be ordered from ANSI at

webstore.ansi.org

The title of the document is "Information Technology – Programming Languages – C++" and its document number is INCITS/ISO/IEC 14882-2003.

We have included an extensive bibliography of books and papers on C++ and object-oriented programming. We also list many websites relating to C++ and object-oriented programming in our C++ Resource Center at www.deitel.com/cplusplus/ . We list several websites in Section 1.12 , including links to free C++ compilers, resource sites, some fun C++ games and game programming tutorials.

## 1.7. 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:* This 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 in two ways—using the Windows XP Command Prompt and using a shell on Linux (similar to a Windows Command Prompt ). The application runs similarly on both platforms. Many development environments are available in which readers can compile, build and run C++ applications, such as Eclipse, GNU C++, Microsoft Visual C++, etc.

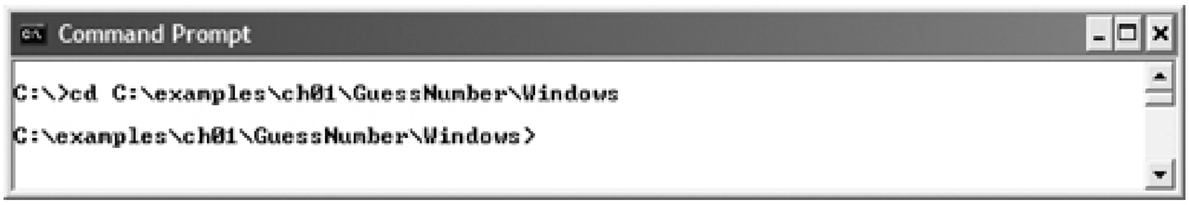
In the following steps, you'll run the application and enter various numbers to guess the correct number. Throughout the book, we use fonts to distinguish between features you see on the screen (e.g., the Command Prompt ) and elements that are not directly related to the screen. Our convention is to emphasize screen features like titles and menus (e.g., the File menu) in a semibold sans-serif Helvetica font and to 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 have noticed, the defining occurrence of each key term is set in bold italic. For the figures in this section, we highlight the user input required by each step and point out significant parts of the application. To make these features more visible, we have 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 , 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 XP Command Prompt

1. Checking your setup. Read the Before You Begin section at the beginning of this book to ensure that you've copied the book's examples to your hard drive.

2. Locating the completed application. Open a Command Prompt window. For readers using Windows 95, 98 or 2000, select Start > Programs > Accessories > Command Prompt. For Windows XP users, select Start > All Programs > Accessories > Command Prompt. To change to your completed GuessNumber application directory, type ***cd C:\examples\ch01\GuessNumber\Windows***, then press *Enter* (Fig. 1.2). The command cd is used to change directories.

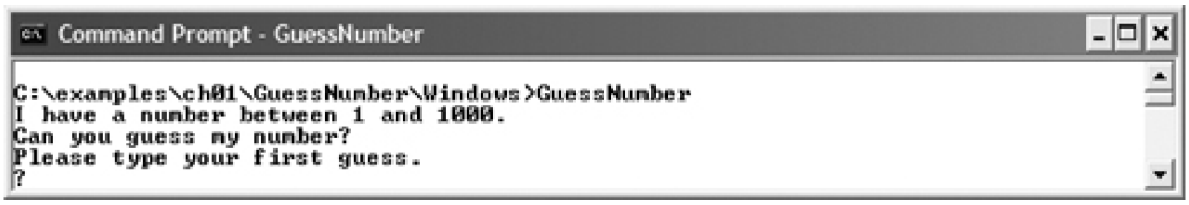
**Fig. 1.2. Opening a Command Prompt window and changing the directory.**



```
C:\>cd C:\examples\ch01\GuessNumber\Windows

C:\examples\ch01\GuessNumber\Windows>
```

3. Running the *GuessNumber* application. Now that you are in the directory that contains the GuessNumber application, type the command GuessNumber (Fig. 1.3) and press *Enter*. [*Note:* GuessNumber.exe is the actual name of the application; however, Windows assumes the .exe extension by default.]

**Fig. 1.3. Running the GuessNumber application.**

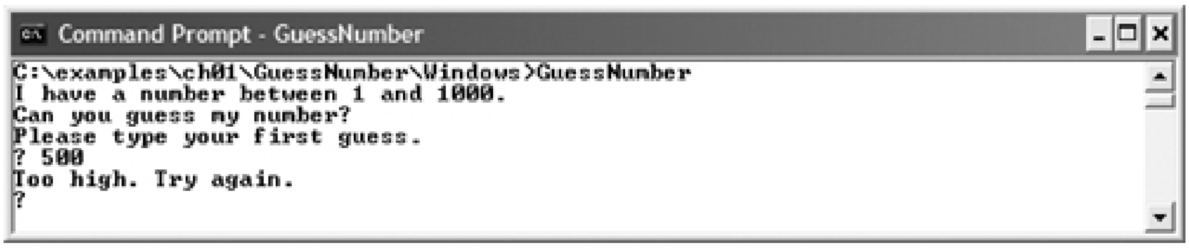


```
C:\examples\ch01\GuessNumber\Windows>GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

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. 1.3). At the prompt, enter 500 (Fig. 1.4)

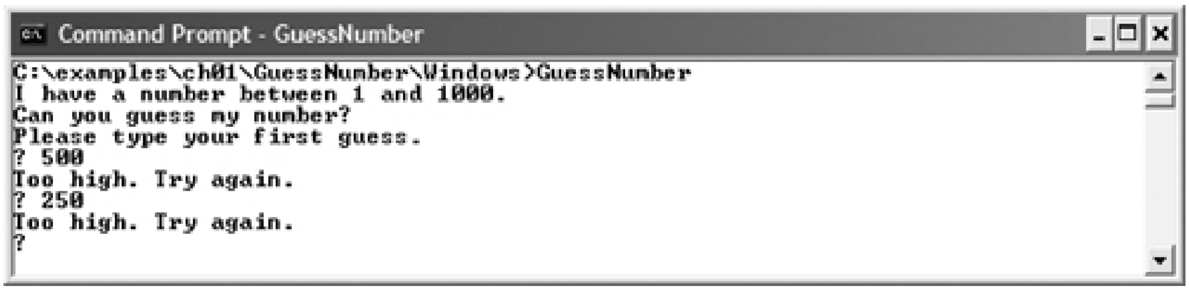
**Fig. 1.4. Entering your first guess.**



```
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.
?
```

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. 1.5). 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.

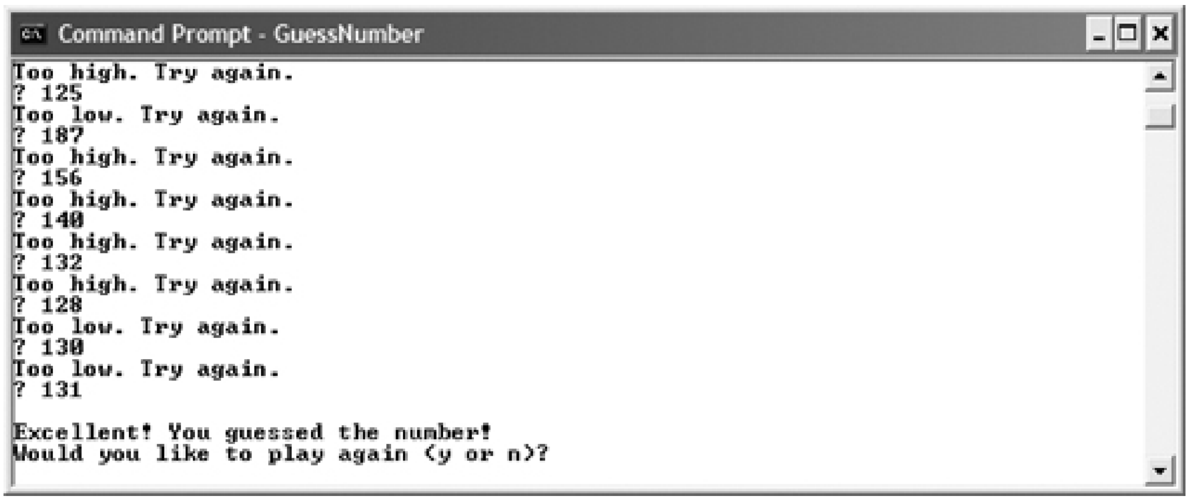
**Fig. 1.5. Entering a second guess and receiving feedback.**



```
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.
?
```

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. 1.6).

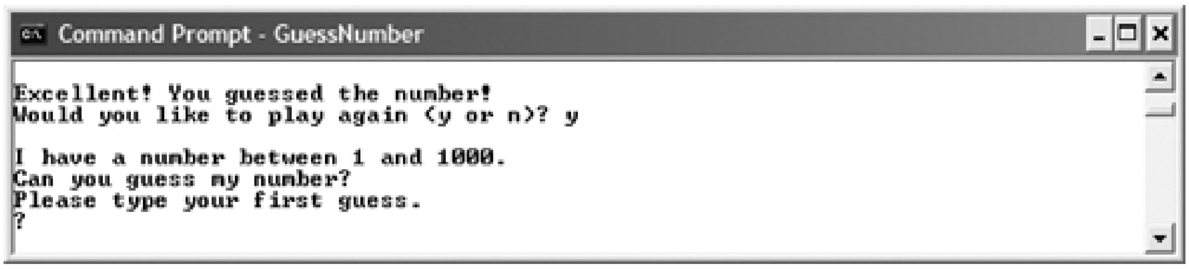
**Fig. 1.6. Entering additional guesses and guessing the correct number.**



```
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)?
```

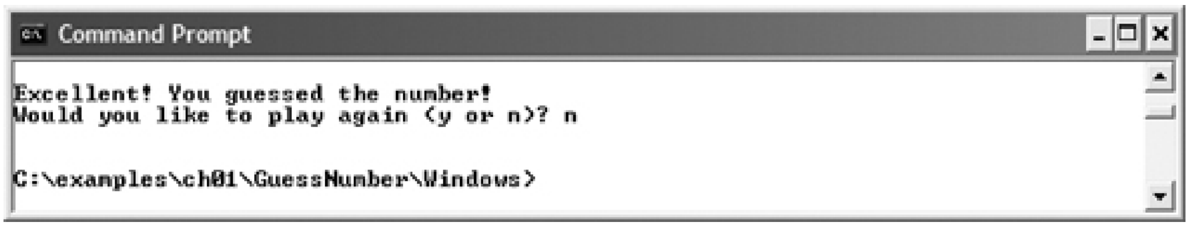
7. Playing the game again or exiting the application. After you guess correctly, the application asks if you would like to play another game (Fig. 1.6). At the "Would you like to play again (y or n)?" prompt, entering the one character y causes the application to choose a new number and displays the message "Please type your first guess." followed by a question mark prompt (Fig. 1.7) 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. 1.8). Each time you execute this application from the beginning (i.e., *Step 3*), it will choose the same numbers for you to guess.

**Fig. 1.7. Playing the game again.**



```
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. 1.8. Exiting the game.**



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

C:\examples\ch01\GuessNumber\Windows>
```

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

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

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. 1.9) by typing

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

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

**Fig. 1.9. Changing to the GuessNumber application's directory after logging in to your account.**

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

**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. 1.10 . This command compiles the application and produces an executable file called `GuessNumber`.

**Fig. 1.10. Compiling the GuessNumber application using the `g++` command.**

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

**3.** Running the *GuessNumber* application. To run the executable file `GuessNumber`, type `./GuessNumber` at the next prompt, then press *Enter* (Fig. 1.11).

**Fig. 1.11. Running the GuessNumber application.**

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

**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. 1.11). At the prompt, enter `500` (Fig. 1.12). [*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.]

**Fig. 1.12. Entering an initial 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.
?
```

**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. 1.12). At the next prompt, enter `250` (Fig. 1.13). This time the application displays `"Too low. Try again."` , because the value you entered is less than the correct guess.

**Fig. 1.13. Entering a second guess and receiving feedback.**

```
~/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.
?
```

6. Entering additional guesses. Continue to play the game (Fig. 1.14) by entering values until you guess the correct number. When you guess correctly, the application displays "Excellent! You guessed the number." (Fig. 1.14).

**Fig. 1.14. Entering additional guesses and guessing the correct 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)?
```

7. Playing the game again or exiting the application. After you guess the correct number, the application asks if you would 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. 1.15) 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. 1.16). Each time you execute this application from the beginning (i.e., *Step 3*), it will choose the same numbers for you to guess.

Fig. 1.15. Playing the game again.

```
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. 1.16. Exiting the game.

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

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

## 1.8. Software Technologies

In this section, we discuss a number of software engineering buzzwords that you'll hear in the software development community. We've created Resource Centers on most of these topics, with many more on the way. You can find our complete list of Resource Centers at www.deitel.com/ResourceCenters.html.

Agile Software Development is a set of methodologies that try to get software implemented quickly with fewer resources than previous methodologies. Check out the Agile Alliance (www.agilealliance.org ) and the Agile Manifesto (www.agilemanifesto.org).

Refactoring involves reworking code to make it clearer and easier to maintain while preserving its functionality. It's widely employed with agile development methodologies. Many refactoring tools are available to do major portions of the reworking automatically. Check out our Resource Center on refactoring.

Design patterns are proven architectures for constructing flexible and maintainable object-oriented software. The field of design patterns tries to enumerate those recurring patterns, encouraging software designers to reuse them to develop better quality software with less time, money and effort.

Game programming. The computer game business is larger than the first-run movie business. College courses and even majors are now devoted to the sophisticated software techniques used in game programming. Check out our Resource Centers on Game Programming, C++ Game Programming and Programming Projects.

Open source software is a style of developing software in contrast to proprietary development 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 generally gets scrutinized by a much larger audience than proprietary software, so bugs get removed faster. Open source also encourages more innovation. Sun recently announced that it is open sourcing Java. Some organizations you'll hear a lot about in the open source community are the Eclipse Foundation (the Eclipse IDE is popular for C++ and Java software development), the Mozilla Foundation (creators of the Firefox browser), the Apache Software Foundation (creators of the Apache web server) and SourceForge (which provides the tools for managing open source projects and currently has over 150,000 open source projects under development).

Linux is an open source operating system and one of the greatest successes of the open source movement. MySQL is an open source database management system. PHP is the most popular open source server-side "scripting" language for developing Internet-based applications. LAMP is an acronym for the set of open source technologies that many developers used to build web applications—it stands for Linux, Apache, MySQL and PHP (or Perl or Python—two other languages used for similar purposes).

Software has generally been viewed as a product; most software still is offered this way. If you want to run an application, you buy a software package from a software vendor. You then install that software on your computer and run it as needed. As new versions of the software appear, you upgrade your software, often at significant expense. This process can become cumbersome for organizations with tens of thousands of systems that must be maintained on a diverse array of computer equipment. With Software as a Service (SaaS) the software runs on servers elsewhere on the Internet. When those servers are updated, all clients worldwide see the new capabilities; no local installation is needed. You access the service through a browser—these are quite portable, so you can run the same applications on different kinds of computers from anywhere in the world. Salesforce.com , Google, and Microsoft's Office Live and Windows Live all offer SaaS.

## 1.9. Future of C++: Open Source Boost Libraries, TR1 and C++0x

Bjarne Stroustrup, the creator of C++, has expressed his vision for the future of C++. The main goals for the new standard are to make C++ easier to learn, improve library building capabilities, and increase compatibility with the C programming language.

Chapter 21 considers the future of C++—we introduce the Boost C++ Libraries, Technical Report 1 (TR1) and C++0x. The Boost C++ Libraries are free, open source libraries created by members of the C++ community. Boost has grown to over 70 libraries, with more being added regularly. Today there are thousands of programmers in the Boost open source community. Boost provides C++ programmers with useful, well-designed 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. We overview the libraries included in TR1 and provide code examples for the "regular expression" and "smart pointer" libraries.

Regular expressions are used to match specific character patterns in text. They can be used to validate data to ensure that it is 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, which we present in Chapter 8, Pointers and Pointer-Based Strings. Smart pointers help you avoid errors by providing additional functionality to standard pointers. This functionality typically strengthens the process of memory allocation and deallocation.

Technical Report 1 describes the proposed changes to the C++ Standard Library, many of which are based on current Boost libraries. These libraries add useful functionality to C++. The C++ Standards Committee is currently revising the C++ Standard. The last standard was published in 1998. Work on the new standard, currently referred to as C++0x, began in 2003. The new standard is likely to be released in 2009. It will include changes to the core language and, most likely, many of the libraries in TR1.

## 1.10. Software Engineering Case Study: Introduction to Object Technology and the UML

Now we begin our study of object orientation. Chapters 1–7, 9 and 13 all end with a brief Software Engineering Case Study section in which we present a carefully paced introduction to object orientation. Our goal here is to help you develop an object-oriented way of thinking and to introduce you to the Unified Modeling Language™ (UML®) —a graphical language that allows people who design object-oriented software systems to use an industry-standard notation to represent them.

In this required section, we introduce basic object-oriented concepts and terminology. The optional sections in Chapters 2–7, 9 and 13 present an object-oriented design and implementation of the software for a simple automated teller machine (ATM) system. The Software Engineering Case Study sections at the ends of Chapters 2–7

- analyze a typical requirements specification that describes a software system (the ATM) to be built
- determine the objects required to implement that system
- determine the attributes the objects will have
- determine the behaviors these objects will exhibit
- specify how the objects interact with one another to meet the system requirements

The Software Engineering Case Study sections at the ends of Chapters 9 and 13 modify and enhance the design presented in Chapters 2–7. Appendix E contains a complete, working C++ implementation of the object-oriented ATM system.

Although our case study is a scaled-down version of an industry-level problem, we nevertheless cover many common industry practices. You'll experience a solid introduction to object-oriented design with the UML. Also, you'll sharpen your code-reading skills by touring the complete, carefully written and well-documented C++ implementation of the ATM.

### Basic Object Technology Concepts

We begin our introduction to object orientation with some key terminology. Everywhere you look in the real world you see objects —people, animals, plants, cars, planes, buildings, computers, monitors and so on. Humans think in terms of objects. Telephones, houses, traffic lights, microwave ovens and water coolers are just a few more objects we see around us every day.

We sometimes divide objects into two categories: animate and inanimate. Animate objects are "alive" in some sense—they move around and do things. Inanimate objects do not move on their own. Objects of both types, however, have some things in common. They all have attributes (e.g., size, shape, color and weight), and they all exhibit behaviors (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleeps, crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water). We'll study the kinds of attributes and behaviors that software objects have.

Humans learn about existing objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults, and between humans and chimpanzees.

Object-oriented design (OOD) models software in terms similar to those that people use to describe real-world objects. It takes advantage of class relationships, where objects of a certain class, such as a class of vehicles, have the same characteristics—cars, trucks, little red wagons and roller skates have much in common. OOD takes advantage of inheritance relationships, where new classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. An object of class "convertible" certainly has the characteristics of the more general class "automobile," but more specifically, the roof goes up and down.

Object-oriented design provides a natural and intuitive way to view the software design process—namely, modeling objects by their attributes, behaviors and interrelationships just as we describe real-world objects. OOD also models communication between objects. Just as people send messages to one another (e.g., a sergeant commands a soldier to stand at attention), objects also communicate via messages. A bank account object may receive a message to decrease its balance by a certain amount because the customer has withdrawn that amount of money.

OOD encapsulates (i.e., wraps) attributes and operations (behaviors) into objects—an object's attributes and operations are intimately tied together. Objects have the property of information hiding. This means that objects may know how to communicate with one another across well-defined interfaces, but normally they are not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves. We can drive a car effectively, for instance, without knowing the details of how engines, transmissions, brakes and exhaust systems work internally—as long as we know how to use the accelerator pedal, the brake pedal, the steering wheel and so on. Information hiding, as we'll see, is crucial to good software engineering.

Languages like C++ are object oriented. Programming in such a language is called object-oriented programming (OOP), and it allows computer programmers to implement object-oriented designs as working software systems. Languages like C, on the other hand, are procedural, so programming tends to be action oriented. In C, the unit of programming is the function. In C++, the unit of programming is the class from which objects are eventually instantiated (an OOP term for "created"). C++ classes contain functions that implement operations and data that implements attributes.

C programmers concentrate on writing functions. Programmers group actions that perform some common task into functions, and group functions to form programs. Data is certainly important in C, but the view is that data exists primarily in support of the actions that functions perform. The verbs in a system specification help the C programmer determine the set of functions that will work together to implement the system.

## Classes, Data Members and Member Functions

C++ programmers concentrate on creating their own user-defined types called classes. Each class contains data as well as the set of functions that manipulate that data and provide services to clients (i.e., other classes or functions that use the class). The data components of a class are called data members. For example, a bank account class might include an account number and a balance. The function components of a class are called member functions (typically called methods in other object-oriented programming languages such as Java). For example, a bank account class might include member functions to make a deposit (increasing the balance), make a withdrawal (decreasing the balance) and inquire what the current balance is. You use built-in types (and other user-defined types) as the "building blocks" for constructing new user-defined types (classes). The nouns in a system specification help the C++ programmer determine the set of classes from which objects are created that work together to implement the system.

Classes are to objects as blueprints are to houses—a class is a "plan" for building an object of the class. Just as we can build many houses from one blueprint, we can instantiate (create) many objects from one class. You cannot cook meals in the kitchen of a blueprint; you can cook meals in the kitchen of a house. You cannot sleep in the bedroom of a blueprint; you can sleep in the bedroom of a house.

Classes can have relationships with other classes. For example, in an object-oriented design of a bank, the "bank teller" class needs to relate to other classes, such as the "customer" class, the "cash drawer" class, the "safe" class, and so on. These relationships are called associations.

Packaging software as classes makes it possible for future software systems to reuse the classes. Groups of related classes are often packaged as reusable components. Just as realtors often say that the three most important factors affecting the price of real estate are "location, location and location," some people in the software development community say that the three most important factors affecting the future of software development are "reuse, reuse and reuse."

Software Engineering Observation 1.3

*Reuse of existing classes when building new classes and programs saves time, money and effort. Reuse also helps programmers build more reliable and effective systems, because existing classes and components often have gone through extensive testing, debugging and performance tuning.*

Indeed, with object technology, you can build much of the new software you'll need by combining existing classes, just as automobile manufacturers combine interchangeable parts. Each new class you create will have the potential to become a valuable software asset that you and other programmers can reuse to speed and enhance the quality of future software development efforts.

## Introduction to Object-Oriented Analysis and Design (OOAD)

To create the best solutions, you should follow a detailed process for analyzing your project's requirements (i.e., determining *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 would go through this process and carefully review the design (or 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 is called an object-oriented analysis and design (OOAD) process. Experienced programmers know that analysis and design can save many hours by helping them to avoid an ill-planned system-development approach that has to be abandoned part of the way through its implementation, possibly wasting considerable time, money and effort.

Ideally, members of a group should agree on a strictly defined process for solving their problem and a uniform way of communicating the results of that process to one another. 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), was developed in the mid-1990s under the initial direction of three software methodologists—Grady Booch, James Rumbaugh and Ivar Jacobson.

## History of the UML

In the 1980s, increasing numbers of organizations began using OOP to build their applications, and a need developed for a standard OOAD process. Many methodologists—including Booch, Rumbaugh and Jacobson—individually produced and promoted separate processes to satisfy this need. Each process had its own notation, or "language" (in the form of graphical diagrams), to convey the results of analysis and design.

By the early 1990s, different organizations, and even divisions within the same organization, were using their own unique processes and notations. At the same time, these organizations also wanted to use software tools that would support their particular processes. Software vendors found it difficult to provide tools for so many processes. A standard notation and standard processes were needed.

In 1994, James Rumbaugh joined Grady Booch at Rational Software Corporation (now a division of IBM), and the two began working to unify their popular processes. They soon were joined by Ivar Jacobson. In 1996, the group released early versions of the UML to the software engineering community and requested feedback. Around the same time, an organization known as the Object Management Group™ (OMG™) invited submissions for a common modeling language. The OMG (www.omg.org) is a nonprofit organization that promotes the standardization of object-oriented technologies by issuing guidelines and specifications, such as the UML. Several corporations—among them HP, IBM, Microsoft, Oracle and Rational Software—had already recognized the need for a common modeling language. In response to the OMG's request for proposals, these companies formed UML Partners—the consortium that developed the UML version 1.1 and submitted it to the OMG. The OMG accepted the proposal and, in 1997, assumed responsibility for the continuing maintenance and revision of the UML. The UML version 2 now available marks the first major revision of the UML since the 1997 version 1.1 standard. We present UML 2 terminology and notation throughout this book.

### What Is the UML?

The UML is now the most widely used graphical representation scheme for modeling object-oriented systems. It has indeed unified the various popular notational schemes. Those who design systems use the language (in the form of diagrams) to model their systems.

An attractive feature of the UML is its flexibility. The UML is extensible (i.e., capable of being enhanced with new features) and is independent of any particular OOAD process. UML modelers are free to use various processes in designing systems, but all developers can now express their designs with one standard set of graphical notations.

In our Software Engineering Case Study sections, we present a simple, concise subset of the UML. We then use this subset to guide you through a complete object-oriented design experience with the UML.

### UML Web Resources

For more information about the UML, refer to the websites listed below. For additional UML sites, refer to the web resources listed at the end of Section 2.7.

www.uml.org

This UML resource page from the Object Management Group (OMG) provides specification documents for the UML and other object-oriented technologies.

www.ibm.com/software/rational/uml

This is the UML resource page for IBM Rational—the successor to the Rational Software Corporation (the company that created the UML).

### Recommended Readings

The following books provide information about object-oriented design with the UML:

Ambler, S. *The Object Primer: Agile Model-Driven Development with UML 2.0, Third Edition*. New York: Cambridge University Press, 2005.

Arlow, J., and I. Neustadt. *UML and the Unified Process: Practical Object-Oriented Analysis and Design, Second Edition*. Boston: Addison-Wesley Professional, 2006.

Fowler, M. *UML Distilled, Third Edition: A Brief Guide to the Standard Object Modeling Language*. Boston: Addison-Wesley Professional, 2004.

Rumbaugh, J., I. Jacobson and G. Booch. *The Unified Modeling Language User Guide, Second Edition*. Boston: Addison-Wesley Professional, 2006.

## Section 1.10 Self-Review Exercises

**1.1** List three examples of real-world objects that we did not mention. For each object, list several attributes and behaviors.

**1.2** Pseudocode is __________.

- **a.** another term for OOAD
- **b.** a programming language used to display UML diagrams
- **c.** an informal means of expressing program logic
- **d.** a graphical representation scheme for modeling object-oriented systems

**1.3** The UML is used primarily to __________.

- **a.** test object-oriented systems
- **b.** design object-oriented systems
- **c.** implement object-oriented systems
- **d.** Both a and b

## Answers to Section 1.10 Self-Review Exercises

**1.1** [*Note:* Answers may vary.] a) A television's attributes include the size of the screen, the number of colors it can display, its current channel and its current volume. A television turns on and off, changes channels, displays video and plays sounds. b) A coffee maker's attributes include the maximum volume of water it can hold, the time required to brew a pot of coffee and the temperature of the heating plate under the coffee pot. A coffee maker turns on and off, brews coffee and heats coffee. c) A turtle's attributes include its age, the size of its shell and its weight. A turtle walks, retreats into its shell, emerges from its shell and eats vegetation.

**1.2** c.

**1.3** b.

## 1.11. Wrap-Up

This chapter discussed the history of C++. We discussed the different types of programming languages, their history and which programming languages are most widely used. We also discussed the C++ Standard Library which contains reusable classes and functions that help C++ programmers create portable C++ programs.

We presented basic object technology concepts, including classes, objects, attributes, behaviors, encapsulation and inheritance. You also learned about the history and purpose of the UML—the industry-standard graphical language for modeling object-oriented software systems.

You learned the typical steps for creating and executing a C++ application. You "test-drove" a sample C++ application.

We discussed several key software technologies and concepts, including open source, and looked to the future of C++. In later chapters, we'll present two open source libraries—Ogre for graphics and game programming, and Boost for broadly enhancing the C++ Standard Library's capabilities.

In the next chapter, you'll create your first C++ applications. You'll see several examples that demonstrate how programs display messages on the screen and obtain information from the user at the keyboard for processing.

## 1.12. Web Resources

This section provides many web resources that will be useful to you as you learn C++. The sites include C++ resources, C++ development tools and some links to fun games built with C++. This section also lists our own websites where you can find downloads and resources associated with this book.

### Deitel & Associates Websites

www.deitel.com/books/cppfp/

The Deitel & Associates *C++ for Programmers* site. Here you'll find links to the book's examples and other resources, such as our *Dive Into™ guides* that help you get started with several C++ integrated development environments (IDEs).

www.deitel.com/cplusplus/

www.deitel.com/cplusplusgameprogramming/

www.deitel.com/cplusplusboostlibraries/

www.deitel.com/codesearchengines/

www.deitel.com/programmingprojects/

www.deitel.com/visualcplusplus/

Our C++ and related Resource Centers on www.deitel.com. Start your search here for resources, downloads, tutorials, documentation, books, e-books, journals, articles, blogs, RSS feeds and more that will help you develop C++ applications.

www.deitel.com

Please check the Deitel & Associates site for updates, corrections and additional resources for all Deitel publications.

www.deitel.com/newsletter/subscribe.html

Please visit this site to subscribe for the *Deitel® Buzz Online* e-mail newsletter to follow the Deitel & Associates publishing program, including updates and errata to *C++ for Programmers*.

### Compilers and Development Tools

www.thefreecountry.com/developercity/ccompilers.shtml

This site lists free C and C++ compilers for a variety of operating systems.

msdn.microsoft.com/vstudio/express/visualc/default.aspx

The *Microsoft Visual C++ Express* site provides a free download of *Visual C++ Express* edition, product information, overviews and supplemental materials for Visual C++.

www.codegear.com/products/cppbuilder

This is a link to the *Code Gear C++Builder* site.

www.compilers.net

*Compilers.net* is designed to help users locate compilers.

developer.intel.com/software/products/compilers/cwin/index.htm

An evaluation download of the *Intel C++ compiler* is available at this site.

## Resources

www.hal9k.com/cug

The *C/C++ Users Group (CUG)* site contains C++ resources, journals, shareware and freeware.

www.devx.com

*DevX* is a comprehensive resource for programmers that provides the latest news, tools and techniques for various programming languages. The *C++ Zone* offers tips, discussion forums, technical help and online newsletters.

www.acm.org/crossroads/xrds3-2/ovp32.html

*The Association for Computing Machinery (ACM)* site offers a comprehensive listing of C++ resources, including recommended texts, journals and magazines, published standards, newsletters, FAQs and newsgroups.

www.accu.informika.ru/resources/public/terse/cpp.htm

*The Association of C & C++ Users (ACCU)* site contains links to C++ tutorials, articles, developer information, discussions and book reviews.

www.cuj.com

The *C/C++ User's Journal* is an online magazine that contains articles, tutorials and downloads. The site features news about C++, forums and links to information about development tools.

www.research.att.com/~bs/homepage.html

This is the site for Bjarne Stroustrup, designer of the C++ programming language. This site provides a list of C++ resources, FAQs and other useful C++ information.

## Games and Game Programming

www.codearchive.com/list.php?go=0708

This site has several C++ games available for download.

www.mathtools.net/C_C__/Games/

This site includes links to numerous games built with C++. The source code for most of the games is available for download.

www.gametutorials.com/gtstore/c-3-c-tutorials.aspx

This site has tutorials on game programming in C++. Each tutorial includes a description of the game and a list of the methods and functions used in the tutorial.

# 2. Introduction to C++ Programming

**Objectives**

In this chapter you'll learn:

- To write simple computer programs in C++.
- To write simple input and output statements.
- To use fundamental types.
- To use arithmetic operators.
- The precedence of arithmetic operators.
- To write simple decision-making statements.

What's in a name? that which we call a rose By any other name would smell as sweet.

—***William Shakespeare***

When faced with a decision, I always ask, "What would be the most fun?"

—***Peggy Walker***

"Take some more tea," the March Hare said to Alice, very earnestly. "I've had nothing yet," Alice replied in an offended tone: "so I can't take more." "You mean you can't take less," said the Hatter: "it's very easy to take more than nothing."

—***Lewis Carroll***

High thoughts must have high language.

—***Aristophanes***

**Outline**

2.8 Wrap-Up

## 2.1. Introduction

In this chapter, we present five examples that demonstrate how your programs can display messages and obtain information from the user for processing. The first three examples display messages on the screen. The next obtains two numbers from a user, calculates their sum and displays the result. The accompanying discussion shows you how to perform various arithmetic calculations and save their results for later use. The fifth example demonstrates decision-making fundamentals by comparing two numbers, then displaying messages based on the comparison results.

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

C++ uses notations that may appear strange to nonprogrammers. We now consider a simple program that prints a line of text (Fig. 2.1).

**Fig. 2.1. Text-printing program.**

```
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
12 } // end function main
```

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

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. 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 C's style in which a comment—possibly containing many lines—begins with /* and ends with */.]

Line 3

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

is a preprocessor directive , which is a message to the C++ preprocessor (introduced in Section 1.5). 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 file <iostream> . This file must be included for any program that outputs data to the screen or inputs data from the keyboard using C++-style stream input/output. We discuss header files in more detail in Chapter 6 , Functions and an Introduction to Recursion, and explain the contents of <iostream> in Chapter 15, Stream Input/Output.

Common Programming Error 2.1

*Forgetting to include the* `<iostream>` *header file in a program that inputs data from the keyboard or outputs data to the screen causes the compiler to issue an error message, because it cannot recognize references to the stream components (e.g.,* `cout`*).*

Line 4 is simply a blank line. You use blank lines, space characters and tab characters (i.e., "tabs") to make programs easier to read. Together, these characters are known as white space . Whitespace characters are normally ignored by the compiler.

Line 5

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

is another single-line comment.

Line 6

```
int main()
```

is a part of every C++ program. The parentheses after `main` indicate that `main` is a function . C++ programs typically consist of one or more functions and classes (as you'll see in Chapter 3 ). Exactly one function in every program must be `main`. Figure 2.1 contains only one function. C++ programs begin executing at function `main`, even if `main` is not the first function 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 . You'll see how to create your own functions in Section 3.5 . We discuss functions in greater depth in Chapter 6. For now, simply include `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 12) must end each function's body. Line 8

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

prints the string of characters contained between the double quotation marks. White-space 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 . Every C++ statement must end with a semicolon (also known as the statement terminator). Preprocessor directives (like `#include` ) do not end with a semicolon. 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. We discuss `std::cout`'s many features in detail in Chapter 15.

Notice that we placed `std::` before `cout` . This is required when we use names that we've brought into the program by the preprocessor 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 22 , 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—in Fig. 2.9 , we introduce the using declaration, which will enable us to omit std:: before each use of a name in the std namespace.

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. The right operand'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 cursor to move to the beginning of the next line on the screen. Some common escape sequences are listed in Fig. 2.2.

**Fig. 2.2. Escape sequences.**

| 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. Use to print a single quote character. |
| \" | Double quote. Used to print a double quote character. |

Common Programming Error 2.2

*Omitting the semicolon at the end of a C++ statement is a syntax error. (Again, preprocessor directives do not end in a semicolon.)*

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. In Chapter 6 we discuss functions in detail, and the reasons for including this statement will become clear. For now, simply include this statement in each program, or the compiler may produce a warning on some systems. The right brace, }, (line 12) indicates the end of function main.

Good Programming Practice 2.1

*Many programmers make the last character printed by a function a newline (\n). This ensures that the function will leave the screen cursor positioned at the beginning of a new line. Conventions of this nature encourage software reusability—a key goal in software development.*

Good Programming Practice 2.2

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

Good Programming Practice 2.3

*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 recommend using either 1/4-inch tab stops or (preferably) three spaces to form a level of indent.*

## 2.3. Modifying Our First C++ Program

This section continues our introduction to C++ programming with two examples, showing how to modify the program in 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 white background in the code table 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 the second stream insertion (line 9) begins printing on the same line immediately following the space.

**Fig. 2.3. Printing a line of text with multiple statements.**

```
 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
 5  // function main begins program execution
 6  int main()
 7  {
 8     std::cout << "Welcome ";
 9     std::cout << "to C++!\n";
10
11     return 0; // indicate that program ended successfully
12
13  } // end function main
```

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

### 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.

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

```
// Fig. 2.4: fig02_04.cpp
// Printing multiple lines of text with a single statement.
#include<iostream>// allows program to output data to the screen

// function main begins program execution
int main()
{
  std::cout <<"Welcome\nto\n\nC++!\n";

   return0; // indicate that program ended successfully

 } // end function main
```

```
Welcome
to

C++!
```

## 2.4. Another C++ Program: Adding Integers

Our next program uses the input stream object std::cin and the stream extraction operator, >>, to obtain 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. Note that we highlight the user's input in bold.

The comments in lines 1 and 2 state the name of the file and the purpose of the program. The C++ preprocessor directive

```
#include <iostream> // allows program to perform input and output
```

in line 3 includes the contents of the <iostream> header file in the program.

The program begins execution with function main (line 6). The left brace (line 7) marks the beginning of main's body and the corresponding right brace (line 25) marks the end of main.

Lines 9–11

```
int number1; // first integer to add
int number2; // second integer to add
int sum; // sum of number1 and number2
```

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, i.e., whole numbers such as 7, –11, 0 and 31914. 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 as follows:

```
int number1, number2, sum;
```

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

We'll soon discuss the data type double for specifying real numbers, and the data 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 often called fundamental types or built-in types. Fundamental-type names are keywords and therefore must appear in all lowercase letters. Appendix C contains the complete list of fundamental types.

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 impose some restrictions on the length of identifiers. Use identifiers of 31 characters or fewer to ensure portability.*

Good Programming Practice 2.4

*Avoid identifiers that begin with underscores and double underscores, because C++ compilers may use names like that for their own purposes internally. This will prevent names you choose from being confused with names the compilers choose.*

Error-Prevention Tip 2.1

*Languages like C++ are "moving targets." As they evolve, more keywords could be added to the language. Avoid using "loaded" words like "object" as identifiers. Even though "object" is not currently a keyword in C++, it could become one; therefore, future compiling with new compilers could break existing code.*

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

**Fig. 2.5. Addition program that displays the sum of two integers entered at the keyboard.**

```
 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
 4
 5  // function main begins program execution
 6  int main()
 7  {
 8     // variable declarations
 9     int number1; // first integer to add
10     int number2; // second integer to add
11     int sum; // sum of number1 and number2
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
23     return 0; // indicate that program ended successfully
24
25  } // end function main
```

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

```
int number1; // first integer to add
```

could have been placed immediately before line 14

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

Line 13

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

prints the string Enter first integer: on the screen. We like to pronounce the preceding statement as "std::cout *gets* the character string "Enter first integer: "." Line 14

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

uses the 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."

Error-Prevention Tip 2.2

*Programs should validate the correctness of all input values to prevent erroneous information from affecting a program's calculations.*

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.

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.

The assignment statement in line 19

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

calculates the sum of the variables number1 and number2 and assigns the result to variable sum using the assignment operator =. The = operator and the + operator are called binary operators because each has two operands. In the case of the + operator, the operands are number1 and number2. In the case of the preceding = operator, the operands are sum and the value of the expression number1 + number2.

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 so-called 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.

Note that 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 . It is unnecessary to have multiple statements to output multiple pieces of data.

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 Chapters 9 and 10 ). 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 11).

## 2.5. Arithmetic

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

**Fig. 2.6. Arithmetic operators.**

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

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`. Note that any fractional part in integer division is discarded (i.e., 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).

Common Programming Error 2.3

*Attempting to use the modulus operator (`%`) with noninteger operands is a compilation error.*

### 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 sequence determined by the following rules of operator precedence, which are generally the same as those followed in algebra:

1. Operators in expressions contained within pairs of parentheses are evaluated first. Parentheses are said to be 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. [*Note:* As in algebra, it is acceptable to place unnecessary parentheses in an expression to make the expression clearer. These are called redundant parentheses.]

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 said to be 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 set of rules of operator precedence defines 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, in the expression

```
a + b + c
```

the addition operators (`+`) associate from left to right, so `a + b` is calculated first, then `c` is added to that sum to determine the value of the whole expression. We'll see that some operators associate from right to left. Figure 2.7 summarizes these rules of operator precedence. This table will be expanded as additional C++ operators are introduced. A complete precedence chart is included in Appendix A.

**Fig. 2.7. Precedence of arithmetic operators.**

| **Operator(s)** | **Operation(s)** | **Order of evaluation (precedence)** |
|---|---|---|
| `()` | Parentheses | Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right. |
| `*`<br>`/`<br>`%` | Multiplication, Division, Modulus | Evaluated second. If there are several, they are evaluated left to right. |
| `+`<br>`-` | Addition Subtraction | Evaluated last. If there are several, they are evaluated left to right. |

Good Programming Practice 2.5

*Using redundant parentheses in complex arithmetic expressions can make the expressions clearer.*

## 2.6. Decision Making: Equality and Relational Operators

This section introduces a simple version of C++'s if statement that allows a program to take alternative action based on the truth or falsity of some condition . If the condition is met, i.e., the condition is true, the statement in the body of the if statement is executed. If the condition is not met, i.e., 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 equality operators and relational operators summarized in Fig. 2.8 . 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.

**Fig. 2.8. Equality and relational operators.**

| **Standard algebraic equality or relational operator** | **C++ equality or relational 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 |

Common Programming Error 2.4

*Confusing the equality operator == with the assignment operator = results in logic errors. The equality operator should be read "is equal to," and the assignment operator should be read "gets" or "gets the value of" or "is assigned the value of." Some people prefer to read the equality operator as "double equals." As we discuss in Section 5.9 , confusing these operators may not necessarily cause an easy-to-recognize syntax error, but may cause extremely subtle logic errors.*

The following example 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. Figure 2.9 shows the program and the input/ output dialogs of three sample executions.

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. Once we insert these using declarations, we can write cout instead of std::cout, cin instead of std::cin and endl instead of std::endl, respectively, in the remainder of the program. [*Note:* From this point forward in the book, each example contains one or more using declarations.]

Good Programming Practice 2.6

*Place using declarations immediately after the #include to which they refer.*

Lines 13–14

```
int number1; // first integer to compare
int number2; // second integer to compare
```

declare the variables used in the program. Remember that variables may be declared in one declaration or in separate declarations.

The program uses cascaded stream extraction operations (line 17) to input two integers. Remember that we are 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.

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.

Notice that each if statement in Fig. 2.9 has a single statement in its body and that 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 is called a compound statement or a block).

**Fig. 2.9. Comparing integers using if statements, relational operators and equality operators.**

```
// Fig. 2.9: fig02_09.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; // first integer to compare
   int number2; // second integer to compare

   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;

   return 0; // indicate that program ended successfully

} // 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
```

Common Programming Error 2.5

*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.*

Figure 2.10 shows the precedence and associativity of the operators introduced in this chapter. The operators are shown top to bottom in decreasing order of precedence. Notice that 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`.

**Fig. 2.10. Precedence and associativity of the operators discussed so far.**

| **Operators** | | | | **Associativity** | **Type** |
|---|---|---|---|---|---|
| () | | | | left to right | 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 |

## 2.7. (Optional) Software Engineering Case Study: Examining the ATM Requirements Specification

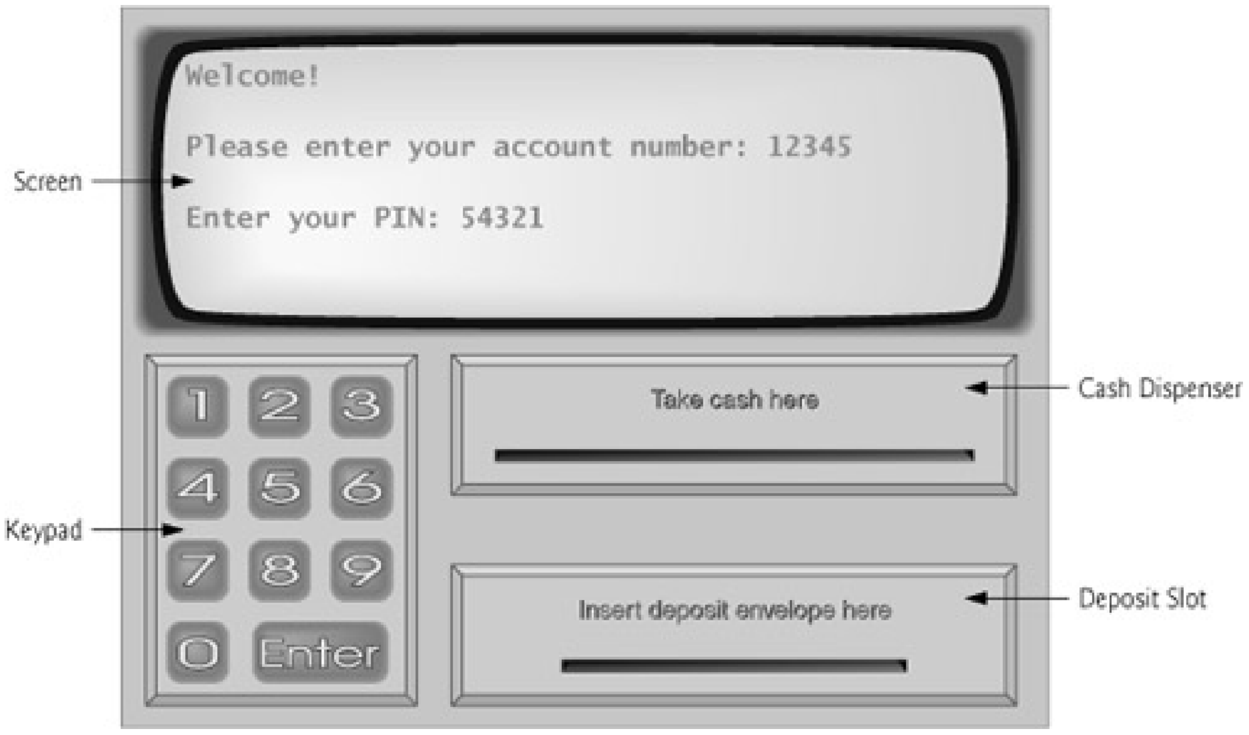
Now we begin our optional object-oriented design and implementation case study. The Software Engineering Case Study sections at the ends of this and the next several chapters will ease you into object orientation. We'll develop software for a simple automated teller machine (ATM) system, providing you with a concise, carefully paced, complete design and implementation experience. In Chapters 3–7, 9 and 13, we'll perform the various steps of an object-oriented design (OOD) process using the UML, while relating these steps to the object-oriented concepts discussed in the chapters. Appendix E implements the ATM using the techniques of object-oriented programming (OOP) in C++. We present the complete case study solution. This is not an exercise; rather, it is an end-to-end learning experience that concludes with a detailed walkthrough of the C++ code that implements our design. It will acquaint you with the kinds of substantial problems encountered in industry and their potential solutions.

We begin our design process by presenting a requirements specification that specifies the overall purpose of the ATM system and *what* it must do. Throughout the case study, we refer to the requirements specification to determine precisely what functionality the system must include.

### Requirements Specification

A local bank intends to install a new automated teller machine (ATM) to allow users (i.e., bank customers) to perform basic financial transactions (Fig. 2.11). Each user can have only one account at the bank. ATM users should be able to view their account balance, withdraw cash (i.e., take money out of an account) and deposit funds (i.e., place money into an account).

**Fig. 2.11. Automated teller machine user interface.**



The user interface of the automated teller machine contains the following hardware components:

- a screen that displays messages to the user
- a keypad that receives numeric input from the user
- a cash dispenser that dispenses cash to the user and
- a deposit slot that receives deposit envelopes from the user.

The cash dispenser begins each day loaded with 500 $20 bills. [*Note:* Owing to the limited scope of this case study, certain elements of the ATM described here do not accurately mimic those of a real ATM. For example, a real ATM typically contains a device that reads a user's account number from an ATM card, whereas this ATM asks the user to type an account number using the keypad. A real ATM also usually prints a receipt at the end of a session, but all output from this ATM appears on the screen.]

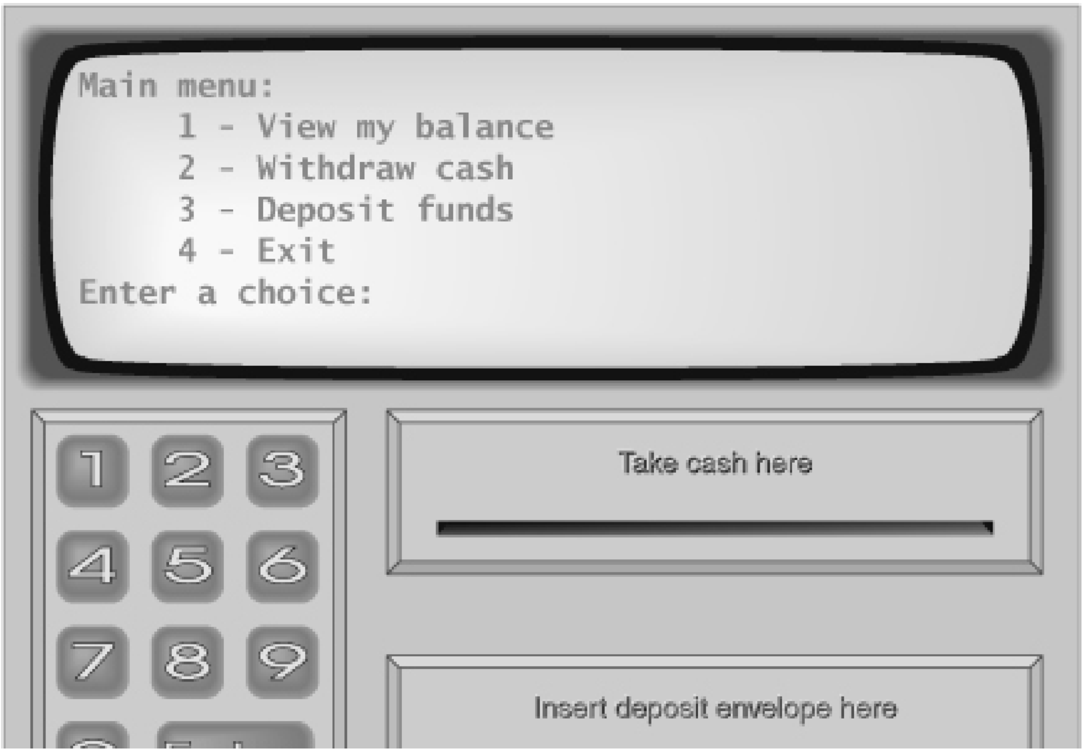
The bank wants you to develop software to perform the financial transactions initiated by bank customers through the ATM. The bank will integrate the software with the ATM's hardware at a later time. The software should encapsulate the functionality of the hardware devices (e.g., cash dispenser, deposit slot) within software components, but it need not concern itself with how these devices perform their duties. The ATM hardware has not been developed yet, so instead of writing your software to run on the ATM, you should develop a first version of the software to run on a personal computer. This version should use the computer's monitor to simulate the ATM's screen, and the computer's keyboard to simulate the ATM's keypad.

An ATM session consists of authenticating a user (i.e., proving the user's identity) based on an account number and personal identification number (PIN), followed by creating and executing financial transactions. To authenticate a user and perform transactions, the ATM must interact with the bank's account information database. [*Note:* A database is an organized collection of data stored on a computer.] For each bank account, the database stores an account number, a PIN and a balance indicating the amount of money in the account. [*Note:* For simplicity, we assume that the bank plans to build only one ATM, so we do not need to worry about multiple ATMs accessing this database at the same time. Furthermore, we assume that the bank does not make any changes to the information in the database while a user is accessing the ATM. Also, any business system like an ATM faces reasonably complicated security issues that go well beyond the scope of a first- or second-semester computer science course. We make the simplifying assumption, however, that the bank trusts the ATM to access and manipulate the information in the database without significant security measures.]

Upon first approaching the ATM, the user should experience the following sequence of events (shown in Fig. 2.11):

1. The screen displays a welcome message and prompts the user to enter an account number.
2. The user enters a five-digit account number, using the keypad.
3. The screen prompts the user to enter the PIN (personal identification number) associated with the specified account number.
4. The user enters a five-digit PIN, using the keypad.
5. If the user enters a valid account number and the correct PIN for that account, the screen displays the main menu (Fig. 2.12). If the user enters an invalid account number or an incorrect PIN, the screen displays an appropriate message, then the ATM returns to *Step 1* to restart the authentication process.

**Fig. 2.12. ATM main menu.**


```
Main menu:
     1 - View my balance
     2 - Withdraw cash
     3 - Deposit funds
     4 - Exit
Enter a choice:
```

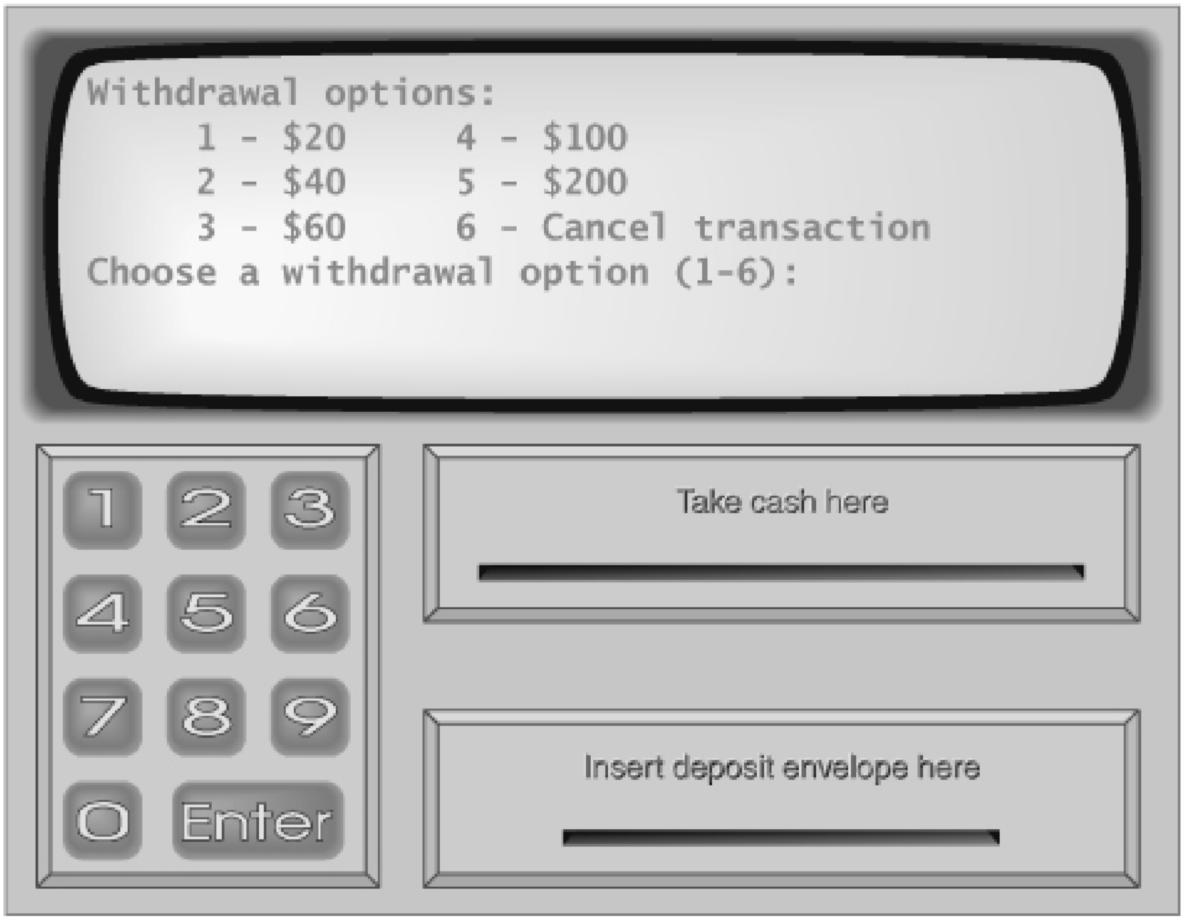
After the ATM authenticates the user, the main menu (Fig. 2.12) displays a numbered option for each of the three types of transactions: balance inquiry (option 1), withdrawal (option 2) and deposit (option 3). The main menu also displays an option that allows the user to exit the system (option 4). The user then chooses either to perform a transaction (by entering 1, 2 or 3) or to exit the system (by entering 4). If the user enters an invalid option, the screen displays an error message, then redisplays to the main menu.

If the user enters 1 to make a balance inquiry, the screen displays the user's account balance. To do so, the ATM must retrieve the balance from the bank's database.

The following actions occur when the user enters 2 to make a withdrawal:

1. The screen displays a menu (shown in Fig. 2.13 ) containing standard withdrawal amounts: $20 (option 1), $40 (option 2), $60 (option 3), $100 (option 4) and $200 (option 5). The menu also contains an option to allow the user to cancel the transaction (option 6).

**Fig. 2.13. ATM withdrawal menu.**



```
Withdrawal options:
     1 - $20      4 - $100
     2 - $40      5 - $200
     3 - $60      6 - Cancel transaction
Choose a withdrawal option (1-6):
```

2. The user enters a menu selection (1–6) using the keypad.

3. If the withdrawal amount chosen is greater than the user's account balance, the screen displays a message stating this and telling the user to choose a smaller amount. The ATM then returns to *Step 1* . If the withdrawal amount chosen is less than or equal to the user's account balance (i.e., an acceptable withdrawal amount), the ATM proceeds to *Step 4* . If the user chooses to cancel the transaction (option 6), the ATM displays the main menu (Fig. 2.12) and waits for user input.

4. If the cash dispenser contains enough cash to satisfy the request, the ATM proceeds to *Step 5* . Otherwise, the screen displays a message indicating the problem and telling the user to choose a smaller withdrawal amount. The ATM then

returns to *Step 1*.

5. The ATM debits (i.e., subtracts) the withdrawal amount from the user's account balance in the bank's database.

6. The cash dispenser dispenses the desired amount of money to the user.

7. The screen displays a message reminding the user to take the money.

The following actions occur when the user enters 3 (while the main menu is displayed) to make a deposit:

1. The screen prompts the user to enter a deposit amount or to type 0 (zero) to cancel the transaction.

2. The user enters a deposit amount or 0, using the keypad. [*Note:* The keypad does not contain a decimal point or a dollar sign, so the user cannot type a real dollar amount (e.g., $1.25). Instead, the user must enter a deposit amount as a number of cents (e.g., 125). The ATM then divides this number by 100 to obtain a number representing a dollar amount (e.g., 125 ÷ 100 = 1.25).]

3. If the user specifies a deposit amount, the ATM proceeds to *Step 4*. If the user chooses to cancel the transaction (by entering 0), the ATM displays the main menu (Fig. 2.12) and waits for user input.

4. The screen displays a message telling the user to insert a deposit envelope into the deposit slot.

5. If the deposit slot receives a deposit envelope within two minutes, the ATM credits (i.e., adds) the deposit amount to the user's account balance in the bank's database. [*Note:* This money is not immediately available for withdrawal. The bank first must physically verify the amount of cash in the deposit envelope, and any checks in the envelope must clear (i.e., money must be transferred from the check writer's account to the check recipient's account). When either of these events occurs, the bank appropriately updates the user's balance stored in its database. This occurs independently of the ATM system.] If the deposit slot does not receive a deposit envelope within this time period, the screen displays a message that the system has canceled the transaction due to inactivity. The ATM then displays the main menu and waits for user input.

After the system successfully executes a transaction, the system should redisplay the main menu (Fig. 2.12) so that the user can perform additional transactions. If the user chooses to exit the system (option 4), the screen should display a thank you message, then display the welcome message for the next user.

## Analyzing the ATM System

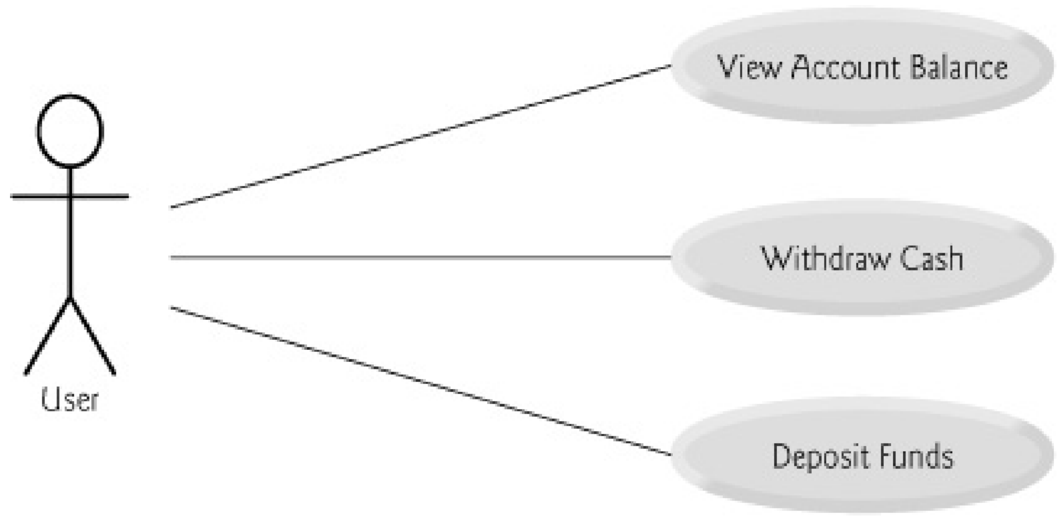
The preceding statement is a simplified example of a requirements specification. Typically, such a document is the result of a detailed process of requirements gathering that might include interviews with potential users of the system and specialists in fields related to the system. For example, a systems analyst who is hired to prepare a requirements specification for banking software (e.g., the ATM system described here) might interview financial experts to gain a better understanding of *what* the software must do. The analyst would use the information gained to compile a list of system requirements to guide systems designers.

The process of requirements gathering is a key task of the first stage of the software life cycle. The software life cycle specifies the stages through which software evolves from the time it is first conceived to the time it is retired from use. These stages typically include: analysis, design, implementation, testing and debugging, deployment, maintenance and retirement. Several software life-cycle models exist, each with its own preferences and specifications for when and how often software engineers should perform each of these stages. Waterfall models perform each stage once in succession, whereas iterative models may repeat one or more stages several times throughout a product's life cycle.

The analysis stage of the software life cycle focuses on defining the problem to be solved. When designing any system, one must certainly *solve the problem right*, but of equal importance, one must *solve the right problem*. Systems analysts collect the requirements that indicate the specific problem to solve. Our requirements specification describes our ATM system in sufficient detail that you do not need to go through an extensive analysis stage—it has been done for you.

To capture what a proposed system should do, developers often employ a technique known as use case modeling. This process identifies the use cases of the system, each of which represents a different capability that the system provides to its clients. For example, ATMs typically have several use cases, such as "View Account Balance," "Withdraw Cash," "Deposit Funds," "Transfer Funds Between Accounts" and "Buy Postage Stamps." The simplified ATM system we build in this case study allows only the first three of these use cases (Fig. 2.14).

**Fig. 2.14. Use case diagram for the ATM system from the User's perspective.**



Each use case describes a typical scenario in which the user uses the system. You have already read descriptions of the ATM system's use cases in the requirements specification; the lists of steps required to perform each type of transaction (i.e., balance inquiry, withdrawal and deposit) actually described the three use cases of our ATM—"View Account Balance," "Withdraw Cash" and "Deposit Funds."

### Use Case Diagrams

We now introduce the first of several UML diagrams in our ATM case study. We create a use case diagram to model the interactions between a system's clients (in this case study, bank customers) and the system. The goal is to show the kinds of interactions users have with a system without providing the details—these are provided in other UML diagrams (which we present throughout the case study). Use case diagrams are often accompanied by informal text that describes the use cases in more detail—like the text that appears in the requirements specification. Use case diagrams are produced during the analysis stage of the software life cycle. In larger systems, use case diagrams are simple but indispensable tools that help system designers remain focused on satisfying the users' needs.

Figure 2.14 shows the use case diagram for our ATM system. The stick figure represents an actor , which defines the roles that an external entity—such as a person or another system—plays when interacting with the system. For our automated teller machine, the actor is a User who can view an account balance, withdraw cash and deposit funds from the ATM. The User is not an actual person, but instead comprises the roles that a real person—when playing the part of a User—can play while interacting with the ATM. Note that a use case diagram can include multiple actors. For example, the use case diagram for a real bank's ATM system might also include an actor named Administrator who refills the cash dispenser each day.

We identify the actor in our system by examining the requirements specification, which states, "ATM users should be able to view their account balance, withdraw cash and deposit funds." The actor in each of the three use cases is the User who interacts with the ATM. An external entity—a real person—plays the part of the User to perform financial transactions. Figure 2.14 shows one actor, whose name, User, appears below the actor in the diagram. The UML models each use case as an oval connected to an actor with a solid line.

Software engineers (more precisely, systems analysts) must analyze the requirements specification or a set of use cases and design the system before programmers implement it. During the analysis stage, systems analysts focus on understanding the requirements specification to produce a high-level specification that describes *what* the system is supposed to do. The output of the design stage—a design specification—should specify clearly *how* the system should be constructed to satisfy these requirements. In the next several Software Engineering Case Study sections, we perform the steps of a simple object-oriented design (OOD) process on the ATM system to produce a design specification containing a collection of UML diagrams and supporting text. Recall that the UML is designed for use with any OOD process. Many such processes exist, the best known of which is the Rational Unified Process™ (RUP) developed by Rational Software Corporation (now a division of IBM). RUP is a rich process intended for designing "industrial strength" applications. For this case study, we present our own simplified design process.

## Designing the ATM System

We now begin the design stage of our ATM system. A system is a set of components that interact to solve a problem. For example, to perform the ATM system's designated tasks, our ATM system has a user interface (Fig. 2.11), contains software that executes financial transactions and interacts with a database of bank account information. System structure describes the system's objects and their interrelationships. System behavior describes how the system changes as its objects interact with one another. Every system has both structure and behavior—designers must specify both. There are several distinct types of system structures and behaviors. For example, the interactions among objects in the system differ from those between the user and the system, yet both constitute a portion of the system behavior.

The UML 2 specifies 13 diagram types for documenting the models of systems. Each models a distinct characteristic of a system's structure or behavior—six diagrams relate to system structure; the remaining seven relate to system behavior. We list here only the six types of diagrams used in our case study—one of these (class diagrams) models system structure—the remaining five model system behavior. We overview the remaining seven UML diagram types in Appendix F, UML 2: Additional Diagram Types.

1. Use case diagrams, such as the one in Fig. 2.14, model the interactions between a system and its external entities (actors) in terms of use cases (system capabilities, such as "View Account Balance," "Withdraw Cash" and "Deposit Funds").

2. Class diagrams, which you'll study in Section 3.11, model the classes, or "building blocks," used in a system. Each noun or "thing" described in the requirements specification is a candidate to be a class in the system (e.g., "account," "keypad"). Class diagrams help us specify the structural relationships between parts of the system. For example, the ATM system class diagram will specify that the ATM is physically composed of a screen, a keypad, a cash dispenser and a deposit slot.

3. State machine diagrams, which you'll study in Section 5.10, model the ways in which an object changes state. An object's state is indicated by the values of all the object's attributes at a given time. When an object changes state, that object may behave differently in the system. For example, after validating a user's PIN, the ATM transitions from the "user not authenticated" state to the "user authenticated" state, at which point the ATM allows the user to perform financial transactions (e.g., view account balance, withdraw cash, deposit funds).

4. Activity diagrams, which you'll also study in Section 5.10, model an object's activity—the object's workflow (sequence of events) during program execution. An activity diagram models the actions the object performs and specifies the order in which it performs these actions. For example, an activity diagram shows that the ATM must obtain the balance of the user's account (from the bank's account information database) before the screen can display the balance to the user.

5. Communication diagrams (called collaboration diagrams in earlier versions of the UML) model the interactions among objects in a system, with an emphasis on *what* interactions occur. You'll see in Section 7.12 that these diagrams show which objects must interact to perform an ATM transaction. For example, the ATM must communicate with the bank's account information database to retrieve an account balance.

6. Sequence diagrams also model the interactions among the objects in a system, but unlike communication diagrams, they emphasize *when* interactions occur. You'll see in Section 7.12 that these diagrams help show the order in which interactions occur in executing a financial transaction. For example, the screen prompts the user to enter a withdrawal amount before cash is dispensed.

In Section 3.11, we continue designing our ATM system by identifying the classes from the requirements specification. We accomplish this by extracting key nouns and noun phrases from the requirements specification. Using these classes, we develop our first draft of the class diagram that models the structure of our ATM system.

## Web Resources

The following URLs provide information on object-oriented design with the UML.

www.objectsbydesign.com/books/booklist.html

Lists books on the UML and object-oriented design.

www-306.ibm.com/software/rational/offerings/design.html

Provides information about IBM Rational software available for designing systems. Provides downloads of 30-day trial versions of several products, such as IBM Rational Application Developer.

www.borland.com/us/products/together/index.html

Provides a free 30-day license to download a trial version of Borland® Together® ControlCenter™—a software-development tool that supports the UML.

argouml.tigris.org

Contains information and downloads for ArgoUML, a free open-source UML tool written in Java.

www.objectsbydesign.com/tools/umltools_byCompany.html

Lists software tools that use the UML, such as IBM Rational Rose, Embarcadero Describe, Sparx Systems Enterprise Architect, I-Logix Rhapsody and Gentleware Poseidon for UML.

www.ootips.org/ood-principles.html

Provides answers to the question, "What Makes a Good Object-Oriented Design?"

parlezuml.com/tutorials/umlforjava.htm

Provides a UML tutorial for Java developers that presents UML diagrams side by side with the Java code that implements them.

www.cetus-links.org/oo_uml.html

Introduces the UML and provides links to numerous UML resources.

www.agilemodeling.com/essays/umlDiagrams.htm

Provides in-depth descriptions and tutorials on each of the 13 UML-2 diagram types.

### Recommended Readings

The following books provide information on object-oriented design with the UML.

Booch, G. *Object-Oriented Analysis and Design with Applications*. 3rd ed. Boston: Addison-Wesley, 2004.

Eriksson, H., et al. *UML 2 Toolkit*. Hoboken, NJ: John Wiley & Sons, 2003.

Fowler, M. *UML Distilled*. 3rd ed. Boston: Addison-Wesley Professional, 2004.

Kruchten, P. *The Rational Unified Process: An Introduction*. Boston: Addison-Wesley, 2004.

Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2002.

Roques, P. *UML in Practice: The Art of Modeling Software Systems Demonstrated Through Worked Examples and Solutions*. Hoboken, NJ: John Wiley & Sons, 2004.

Rosenberg, D., and K. Scott. *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*. Reading, MA: Addison-Wesley, 2001.

Rumbaugh, J., I. Jacobson and G. Booch. *The Complete UML Training Course*. Upper Saddle River, NJ: Prentice Hall, 2000.

Rumbaugh, J., I. Jacobson and G. Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson and G. Booch. *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.

Schneider, G. and J. Winters. *Applying Use Cases: A Practical Guide*. 2nd ed. Boston: Addison-Wesley Professional, 2002.

## Software Engineering Case Study Self-Review Exercises

**2.1** Suppose we enabled a user of our ATM system to transfer money between two bank accounts. Modify the use case diagram of Fig. 2.14 to reflect this change.

**2.2** ________model the interactions among objects in a system with an emphasis on *when* these interactions occur.

- **a.** Class diagrams
- **b.** Sequence diagrams
- **c.** Communication diagrams
- **d.** Activity diagrams

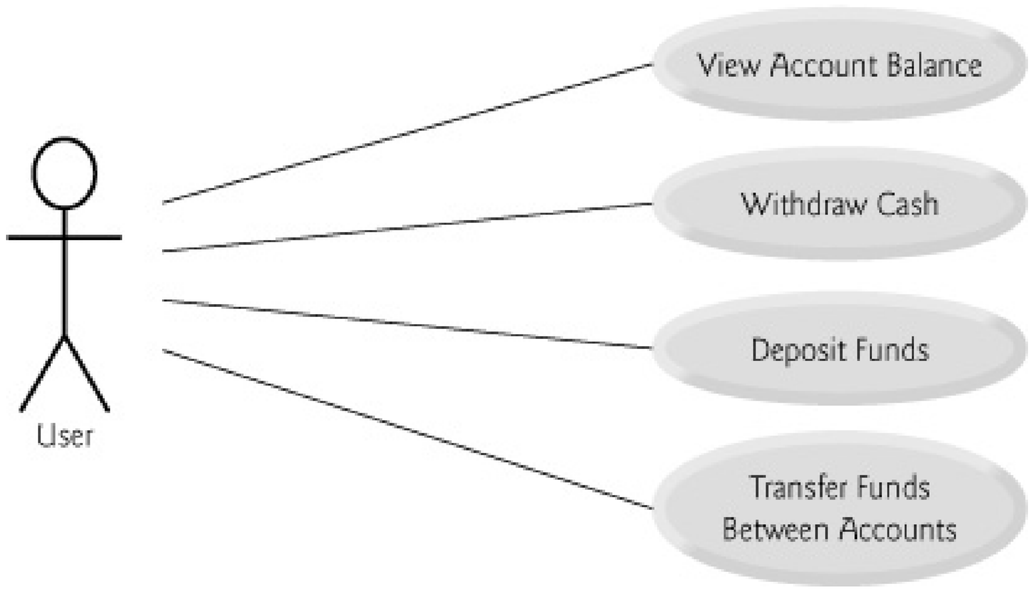
**2.3** Which of the following choices lists stages of a typical software life cycle in sequential order?

- **a.** design, analysis, implementation, testing
- **b.** design, analysis, testing, implementation
- **c.** analysis, design, testing, implementation
- **d.** analysis, design, implementation, testing

## Answers to Software Engineering Case Study Self-Review Exercises

**2.1** Figure 2.15 shows a use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

**Fig. 2.15. Use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.**



**2.2** b.

**2.3** d.

## 2.8. 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 see how to implement your own classes and use objects of those classes in applications.

# 3. Introduction to Classes and Objects

**Objectives**

In this chapter you'll learn:

- What classes, objects, member functions and data members are.
- How to define a class and use it to create an object.
- How to define member functions in a class to implement the class's behaviors.
- How to declare data members in a class to implement the class's attributes.
- How to call a member function of an object to make that member function perform its task.
- The differences between data members of a class and local variables of a function.
- How to use a constructor to ensure that an object's data is initialized when the object is created.
- How to engineer a class to separate its interface from its implementation and encourage reuse.

Nothing can have value without being an object of utility.

—***Karl Marx***

Your public servants serve you right.

—***Adlai E. Stevenson***

Knowing how to answer one who speaks, To reply to one who sends a message.

—***Amenemope***

You'll see something new. Two things. And I call them Thing One and Thing Two.

—***Dr. Theodor Seuss Geisel***

**Outline**

## 3.1. Introduction

In Chapter 2 , you created simple programs that displayed messages to the user, obtained information from the user, performed calculations and made decisions. In this chapter, you'll begin writing programs that employ the basic concepts of object-oriented programming that we introduced in Section 1.10. 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++.

First, we motivate the notion of classes with a real-world example. Then we present a carefully paced sequence of seven 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. This case study is enhanced over the next several chapters, culminating with the version presented in Chapter 7 , Arrays and Vectors. We also introduce the C++ standard library class string in this chapter.

## 3.2. Classes, Objects, Member Functions and Data Members

Let's begin with a simple analogy to help you reinforce your understanding from Section 1.10 of classes and their contents. Suppose you want to drive a car and make it go faster by pressing down on its accelerator pedal. What must happen before you can do this? Well, before you can drive a car, someone has to design it and build it. A car typically begins as engineering drawings, similar to the blueprints used to design a house. These drawings include the design for an accelerator pedal that the driver will use to make the car go faster. In a sense, the pedal "hides" the complex mechanisms that actually make the car go faster, just as the brake pedal "hides" the mechanisms that slow the car, the steering wheel "hides" the mechanisms that turn the car and so on. This enables people with little or no knowledge of how cars are engineered to drive a car easily, simply by using the accelerator pedal, the brake pedal, the steering wheel, the transmission shifting mechanism and other such simple and user-friendly "interfaces" to the car's complex internal mechanisms.

Unfortunately, you cannot drive the engineering drawings of a car—before you can drive a car, it must be built from the engineering drawings that describe it. A completed car will have an actual accelerator pedal to make the car go faster. But even that's not enough—the car will not accelerate on its own, so the driver must press the accelerator pedal to tell the car to go faster.

### Classes and Member Functions

Now let's use our car example to introduce the key object-oriented programming concepts of this section. Performing a task in a program requires a function (such as main, as described in Chapter 2 ). The function describes the mechanisms that actually perform its tasks. The function hides from its user the complex tasks that it performs, just as the accelerator pedal of a car hides from the driver the complex mechanisms of making the car go faster. In C++, we begin by creating a program unit called a class to house a function, just as a car's engineering drawings house the design of an accelerator pedal. Recall from Section 1.10 that a function belonging to a class is called a member function. In a class, you provide one or more member functions that are designed to perform the class's tasks. For example, a class that represents a bank account might contain one member function to deposit money into the account, another to withdraw money from the account and a third to inquire what the current account balance is.

### Objects

Just as you cannot drive an engineering drawing of a car, you cannot "drive" a class. Just as someone has to build a car from its engineering drawings before you can actually drive the car, you must create an object of a class before you can get a program to perform the tasks the class describes. That is one reason C++ is known as an object-oriented programming language. Note also that just as *many* cars can be built from the same engineering drawing, *many* objects can be built from the same class.

### Requesting an Object's Services via Member-Function Calls

When you drive a car, pressing its gas pedal sends a message to the car to perform a task—that is, make the car go faster. Similarly, you send messages to an object—each message is known as a member-function call and tells a member function of the object to perform its task. This is often called requesting a service from an object

### Attributes and Data Members

Thus far, we have used the car analogy to introduce classes, objects and member functions. In addition to the capabilities a car provides, it also has many attributes, such as its color, the number of doors, the amount of gas in its tank, its current speed and its total miles driven (i.e., its odometer reading). Like the car's capabilities, these attributes are represented as part of a car's design in its engineering diagrams. As you drive a car, these attributes are always associated 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. Similarly, an object has attributes that are carried with the object as it is 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.

## 3.3. Overview of the Chapter Examples

The remainder of this chapter presents seven simple examples that demonstrate the concepts we introduced in the context of the car analogy. These examples, summarized below, incrementally build a GradeBook class to demonstrate these concepts:

1. The first example presents a GradeBook class with one member function that simply displays a welcome message when it is called. We show how to create an object of that class and call the member function so that it displays the welcome message.

2. The second example modifies the first by allowing the member function to receive a course name as a so-called argument. Then, the member function displays the course name as part of the welcome message.

3. The third example shows how to store the course name in a GradeBook object. For this version of the class, we also show how to use member functions to set the course name in the object and get the course name from the object.

4. The fourth example demonstrates how the data in a GradeBook object can be initialized when the object is created—the initialization is performed by a special member function called the class's constructor. This example also demonstrates that each GradeBook object maintains its own course name data member.

5. The fifth example modifies the fourth by demonstrating how to place class GradeBook into a separate file to enable software reusability.

6. The sixth example modifies the fifth by demonstrating the good software-engineering principle of separating the interface of the class from its implementation. This makes the class easier to modify without affecting any clients of the class's objects —that is, any classes or functions that call the member functions of the class's objects from outside the objects.

7. The last example enhances class GradeBook by introducing data validation, which ensures that data in an object adheres to a particular format or is in a proper value range. For example, a Date object would require a month value in the range 1–12. In this GradeBook example, the member function that sets the course name for a GradeBook object ensures that the course name is 25 characters or fewer. If not, the member function uses only the first 25 characters of the course name and displays a warning message.

Note that the GradeBook examples in this chapter do not actually process or store grades. We begin processing grades with class GradeBook in Chapter 4 and we store grades in a GradeBook object in Chapter 7, Arrays and Vectors.

## 3.4. Defining a Class with a Member Function

We begin with an example (Fig. 3.1) that consists of class GradeBook (lines 9–17), which represents a grade book that an instructor can use to maintain student test scores, and a main function (lines 20–25) that creates a GradeBook object. Function main uses this object and its member function to display a message on the screen welcoming the instructor to the grade-book program.

**Fig. 3.1. Define class GradeBook with a member function displayMessage, create a GradeBook object, and call its displayMessage function.**

```
 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 std::cout;
 6 using std::endl;
 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()
14    {
15       cout << "Welcome to the Grade Book!" << endl;
16    } // end function displayMessage
17 }; // end class GradeBook
18
19 // function main begins program execution
20 int main()
21 {
22    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
23    myGradeBook.displayMessage(); // call object's displayMessage function
24    return 0; // indicate successful termination
25 } // end main
```

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

First we describe how to define a class and a member function. Then we explain how an object is created and how to call a member function of an object. The first few examples contain function main and the GradeBook class it uses in the same file. Later in the chapter, we introduce more sophisticated ways to structure your programs to achieve better software engineering.

## Class GradeBook

Before function main (lines 20–25) can create an object of class GradeBook, we must tell the compiler what member functions and data members belong to the class. The GradeBook class definition (lines 9–17) contains a member function called displayMessage (lines 13–16) that displays a message on the screen (line 15). Recall that a class is like a blueprint—so we need to make an object of class GradeBook (line 22) and call its displayMessage member function (line 23) to get line 15 to execute and display the welcome message. We'll soon explain lines 22–23 in detail.

The class definition begins in line 9 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 camel case, because the pattern of uppercase and lowercase letters resembles the silhouette of a camel.

Every class's body is enclosed in a pair of left and right braces ({ and }), as in lines 10 and 17. The class definition terminates with a semicolon (line 17).

Common Programming Error 3.1

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

Recall that main is called automatically when you execute a program. As you'll soon see, you must call member function displayMessage explicitly to tell it to perform its task.

Line 11 contains the access-specifier label public:. The keyword public is an access specifier. Lines 13–16 define member function displayMessage. This member function appears after access specifier public: to indicate that the function is "available to the public"—it can be called by other functions in the program (such as main), and by member functions of other classes. Access specifiers are always followed by a colon (:). For the remainder of the text, when we refer to the access specifier public, we'll omit the colon as we did in this sentence. Section 3.6 introduces a second access specifier, private.

Each function in a program performs a task and may return a value when it completes its task—for example, a function might perform a calculation, then return the result of that calculation. 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 13, 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, 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 returns a value.

The name of the member function, displayMessage, follows the return type. By convention, function names begin with a lowercase first letter and all subsequent words in the name begin with a capital letter. The parentheses after the member function name indicate that this is a function. An empty set of parentheses, as shown in line 13, 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 3.5. Line 13 is commonly referred to as the function header. Every function's body is delimited by left and right braces ({ and }), as in lines 14 and 16.

The body of a function contains statements that perform the function's task. In this case, member function displayMessage contains one statement (line 15) that displays the message "Welcome to the Grade Book!". After this statement executes, the function has completed its task.

Common Programming Error 3.2

*Returning a value from a function whose return type has been declared void is a compilation error.*

Common Programming Error 3.3

*Defining a function inside another function is a syntax error.*

### Testing Class GradeBook

Next, we'd like to use class GradeBook in a program. As you learned in Chapter 2, function main (lines 20–25) begins the execution of every program.

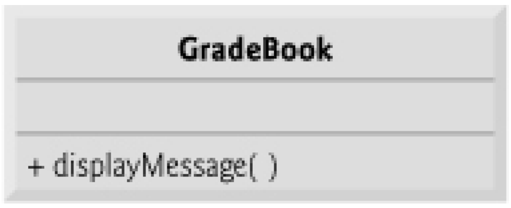
In this program, we'd like to call class GradeBook's displayMessage member function to display the welcome message. Typically, you cannot call a member function of a class until you create an object of that class. (As you'll see in Section 10.7, static member functions are an exception.) Line 22 creates an object of class GradeBook called myGradeBook. Note that the variable's type is GradeBook—the class we defined in lines 9–17. When we declare variables of type int, as we did in Chapter 2, the compiler knows what int is—it's a fundamental type. In line 22, 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 9–17). If we omitted these lines, the compiler would issue an error message (such as "'GradeBook': undeclared identifier" in Microsoft Visual C++ or "'GradeBook': undeclared" in GNU C++). 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 language.

Line 23 calls the member function displayMessage (defined in lines 13–16) 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 23, "myGradeBook." indicates that main should use the GradeBook object that was created in line 22. The empty parentheses in line 13 indicate that member function displayMessage does not require additional data to perform its task. (In Section 3.5, you'll see how to pass data to a function.) When displayMessage completes its task, function main continues executing in line 24, which indicates that main performed its tasks successfully. This is the end of main, so the program terminates.

### UML Class Diagram for Class GradeBook

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 have any attributes. (Section 3.6 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 13 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.**



## 3.5. Defining a Member Function with a Parameter

In our car analogy from Section 3.2, 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 14–23) with a displayMessage member function (lines 18–22) that displays the course name as part of the welcome message. The new version of displayMessage requires a parameter (courseName in line 18) that represents the course name to output.

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

```
// Fig. 3.3: fig03_03.cpp
// Define class GradeBook with a member function that takes a parameter;
// Create a GradeBook object and call its displayMessage function.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <string> // program uses C++ standard string class
using std::string;
using std::getline;

// GradeBook class definition
class GradeBook
{
public:
   // function that displays a welcome message to the GradeBook user
   void displayMessage( string courseName )
   {
      cout << "Welcome to the grade book for\n" << courseName << "!"
         << endl;
   } // end function displayMessage
}; // end class GradeBook

// function main begins program execution
int main()
{
   string nameOfCourse; // string of characters to store the course name
   GradeBook myGradeBook; // create a GradeBook object named myGradeBook

   // prompt for and input course name
   cout << "Please enter the course name:" << endl;
   getline( cin, nameOfCourse ); // read a course name with blanks
   cout << endl; // output a blank line

   // call myGradeBook's displayMessage function
   // and pass nameOfCourse as an argument
   myGradeBook.displayMessage( nameOfCourse );
   return 0; // indicate successful termination
} // end main
```

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

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

Before discussing the new features of class GradeBook, let's see how the new class is used in main (lines 26–40). Line 28 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 file <string> , and the name string, like cout, belongs to namespace std . To enable line 28 to compile, line 9 includes the <string> header file. Note that the using declaration in line 10 allows us to simply write string in line 28 rather than std::string. For now, you can think of string variables like variables of other types such as int. You'll see additional string capabilities in Section 3.10.

Line 29 creates an object of class GradeBook named myGradeBook . Line 32 prompts the user to enter a course name. Line 33 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. (Recall that we highlight user-supplied input in bold.) When cin is used with the stream extraction operator, it 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 33 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. Note that when you press *Enter* while typing program input, a newline is inserted in the input stream. Also note that the <string> header file must be included in the program to use function getline and that the name getline belongs to namespace std.

Line 38 calls myGradeBook's displayMessage member function. The nameOfCourse variable in parentheses is the argument that is passed to member function displayMessage so that it can perform its task. The value of variable nameOfCourse in main becomes the value of member function displayMessage's parameter courseName in line 18. When you execute this program, notice that 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 that a 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 13) to indicate that a function does not require any parameters. Member function displayMessage 's parameter list (Fig. 3.3 , line 18) declares that the function requires one parameter. Each parameter must specify a type and an identifier. In this case, 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 is passed to the function in the function call (line 38 in main). Lines 20–21 display parameter courseName 's value as part of the welcome message. Note that the parameter variable's name (line 18) can be the same as or different from the argument variable's name (line 38)—you'll see why in Chapter 6, Functions and an Introduction to Recursion.

A function can specify multiple parameters by separating each parameter from the next with a comma (we'll see an example in Figs. 6.3–6.4 ). 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 see 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).

Common Programming Error 3.4

*Placing a semicolon after the right parenthesis enclosing the parameter list of a function definition is a syntax error.*

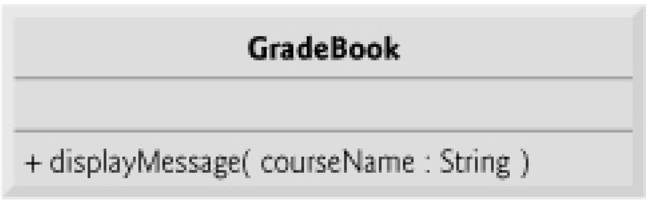
Common Programming Error 3.5

*Defining a function parameter again as a local variable in the function is a compilation error.*

### 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 is 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 18–22) has a string parameter named courseName, so Fig. 3.4 lists courseName : String between the parentheses following the operation name displayMessage. Note that 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 displayMessage operation with a courseName parameter of UML type String.**



## 3.6. Data Members, *set* Functions and *get* 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 immediately following closing right brace (}). 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 is 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.) Recall from Section 3.2 that an object has attributes that are carried with it as it is used in a program. Such attributes exist throughout the life of the object.

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 copy of its attributes in memory. The example in this section demonstrates a GradeBook class that contains a courseName data member to represent a particular GradeBook object's course name.

### GradeBook Class with a Data Member, a *set* Function and a *get* Function

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 at any time during 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. Defining and testing class GradeBook with a data member and *set* and *get* functions.**

```
// 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>
using std::cout;
using std::cin;
using std::endl;

#include <string> // program uses C++ standard string class
using std::string;
using std::getline;

// 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()
   {
      return courseName; // return the object's courseName
   } // end function getCourseName

   // function that displays a welcome message
   void displayMessage()
   {
      // this statement calls getCourseName to get the
      // name of the course this GradeBook represents
      cout << "Welcome to the grade book for\n" << getCourseName() << "!"
         << endl;
   } // end function displayMessage
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook

// function main begins program execution
int main()
{
   string nameOfCourse; // string of characters to store the course name
   GradeBook myGradeBook; // create a GradeBook object named myGradeBook

   // display initial value of courseName
   cout << "Initial course name is: " << myGradeBook.getCourseName()
      << endl;

   // prompt for, input and set course name
   cout << "\nPlease enter the course name:" << endl;
54    getline( cin, nameOfCourse ); // read a course name with blanks
55    myGradeBook.setCourseName( nameOfCourse ); // set the course name
56
57    cout << endl; // outputs a blank line
58    myGradeBook.displayMessage(); // display message with new course name
59    return 0; // indicate successful termination
60 } // 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!
```

Good Programming Practice 3.1

*Place a blank line between member-function definitions to enhance program readability.*

A typical instructor teaches multiple courses, each with its own course name. Line 39 declares that courseName is a variable of type string . Because the variable is declared in the class definition (lines 15–40) but outside the bodies of the class's member-function definitions (lines 19–22, 25–28 and 31–37), the variable is a data member. Every instance (i.e., object) of class GradeBook contains one copy of each of the class's data members—if there are two GradeBook objects, each has its own copy of 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 (in this case, class GradeBook ) 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 access-specifier label private: (line 38). Like public, keyword private is an access specifier. Variables or functions declared after access specifier private (and before the next access specifier) are accessible only to member functions of the class for which they are declared. Thus, data member courseName can be used only in member functions setCourseName, getCourseName and displayMessage of (every object of) class GradeBook. Data member courseName, because it is private, cannot be accessed by functions outside the class (such as main ) or by member functions of other classes in the program. Attempting to access data member courseName in one of these program locations with an expression such as myGradeBook.courseName would result in a compilation error containing a message similar to

```
cannot access private member declared in class 'GradeBook'
```

Software Engineering Observation 3.1

*As a rule of thumb, data members should be declared* private *and member functions should be declared* public. *(We'll see that it is appropriate to declare certain member functions* private *, if they are to be accessed only by other member functions of the class.)*

Common Programming Error 3.6

*An attempt by a function, which is not a member of a particular class (or a friend of that class, as we'll see in Chapter 10 , Classes: A Deeper Look, Part 2), 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 are private. The access specifiers public and private may be repeated, but this is unnecessary and can be confusing.

Good Programming Practice 3.2

*Despite the fact that the* public *and* private *access specifiers may be repeated and intermixed, list all the* public *members of a class first in one group and then list all the* private *members in another group. This focuses the client's attention on the class's* public *interface, rather than on the class's implementation.*

Good Programming Practice 3.3

*If you choose to list the* private *members first in a class definition, explicitly use the* private *access specifier despite the fact that* private *is assumed by default. This improves program clarity.*

Declaring data members with access specifier private is known as data hiding. When a program creates (instantiates) 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 (and displayMessage could do so if necessary).

Software Engineering Observation 3.2

*You'll see in Chapter 10 that functions and classes declared by a class to be friends can access the private members of the class.*

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.*

### Member Functions setCourseName and getCourseName

Member function setCourseName (defined in lines 19–22) 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 55 of main). Line 21 assigns name to data member courseName. In this example, setCourseName does not attempt to 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 basic validation techniques in Section 3.10.

Member function getCourseName (defined in lines 25–28) returns a particular GradeBook object's courseName. 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 returns 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 back 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 assings 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 to variable biggest the value 114.

Common Programming Error 3.7

*Forgetting to return a value from a function that is supposed to return a value is a compilation error.*

Note that the statements in lines 21 and 27 each use variable courseName (line 39) even though it was not declared in any of the member functions. We can use courseName in the member functions of class GradeBook because courseName is a data member of the class. Also note that the order in which member functions are defined does not determine when they are called at execution time. So member function getCourseName could be defined before member function setCourseName.

### Member Function displayMessage

Member function displayMessage (lines 31–37) 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 35–36 output a welcome message that includes the value of data member courseName. Line 35 calls member function getCourseName to obtain the value of courseName. Note that member function displayMessage could also access data member courseName directly, just as member functions setCourseName and getCourseName do. We explain shortly why we choose to call member function getCourseName to obtain the value of courseName.

### Testing Class GradeBook

The main function (lines 43–60) creates one object of class GradeBook and uses each of its member functions. Line 46 creates a GradeBook object named myGradeBook. Lines 49–50 display the initial course name by calling the object's getCourseName member function. Note that 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 53 prompts the user to enter a course name. Local string variable nameOfCourse (declared in line 45) is set to the course name entered by the user, which is obtained by the call to the getline function (line 54). Line 55 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 19) of member function setCourseName (lines 19–22). Then the parameter's value is assigned to data member courseName (line 21). Line 57 skips a line in the output; then line 58 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 we'll see in Chapter 10 ). So a client of an object—that is, any class or function 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 (Fig. 3.5, lines 43–60) 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. The names of these member functions 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. Note that *set* functions are also sometimes called mutators (because they mutate, or change, values), and *get* functions are also sometimes 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 is 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 18, 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, although these member functions *can* access the private data directly. In Fig. 3.5, member functions setCourseName and getCourseName are public member functions, so they are 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 is easier to maintain and less likely to stop working). 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 decide that 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 would continue to display the complete course title "CS101 Introduction to C++ Programming," because it is unaffected by the change to the class's data members. The benefits of calling a *set* function from another member function of a class will become clear when we discuss validation in Section 3.10.

Good Programming Practice 3.4

*Always try to localize the effects of changes to a class's data members by accessing and manipulating the data members through their* get *and* set *functions. Changes to the name of a data member or the data type used to store a data member then affect only the corresponding* get *and* set *functions, but not the callers of those functions.*

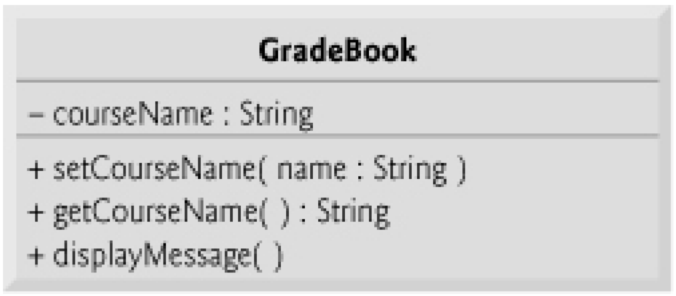
Software Engineering Observation 3.3

*The class designer need not provide* set *or* get *functions for each* private *data item; these capabilities should be provided only when appropriate. If a service is useful to the client code, that service should typically be provided in the class's* public *interface.*

## 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. The minus sign in the UML is equivalent to the private access specifier in C++. Class GradeBook contains three public member functions, so the class diagram lists three operations in the third compartment. Recall that the plus (+) sign before each operation name indicates that the operation is public in C++. 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 (Fig. 3.5) has a string return type in C++, so the class diagram shows a String return type in the UML. Note that operations setCourseName and displayMessage do not return values (i.e., they return void), so the UML class diagram does not specify a return type after the parentheses of these operations. The UML does not use void as C++ does when a function does not return a value.

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



## 3.7. Initializing Objects with Constructors

As mentioned in Section 3.6 , 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 a constructor 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 . The term "constructor" is often abbreviated as "ctor" in the literature—we generally avoid abbreviations.

C++ requires a constructor call for each object that is created, which helps ensure that each object is initialized before it is used in a program. The constructor call occurs implicitly when the object is created. If a class does not explicitly include a constructor, the compiler provides a default constructor—that is, a constructor with no parameters. For example, when line 46 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. [*Note:* 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. You'll learn more about initializing data members that are objects of other classes in Section 10.3.]

In the example of Fig. 3.7, we specify a course name for a GradeBook object when the object is created (line 49). In this case, the argument "CS101 Introduction to C++ Programming" is passed to the GradeBook object's constructor (lines 17–20) 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.

**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.**

```
// Fig. 3.7: fig03_07.cpp
// Instantiating multiple objects of the GradeBook class and using
// the GradeBook constructor to specify the course name
// when each GradeBook object is created.
#include <iostream>
using std::cout;
using std::endl;

#include <string> // program uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
public:
   // constructor initializes courseName with string supplied as argument
   GradeBook( string name )
   {
      setCourseName( name ); // call set function to initialize courseName
   } // end GradeBook constructor

   // function to set the course name
   void setCourseName( string name )
   {
      courseName = name; // store the course name in the object
   } // end function setCourseName

   // function to get the course name
   string getCourseName()
   {
      return courseName; // return object's courseName
   } // end function getCourseName

   // display a welcome message to the GradeBook user
   void displayMessage()
   {
      // call getCourseName to get the courseName
      cout << "Welcome to the grade book for\n" << getCourseName()
         << "!" << endl;
   } // end function displayMessage
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook

// function main begins program execution
int main()
{
   // create two GradeBook objects
   GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
   GradeBook gradeBook2( "CS102 Data Structures in C++" );

   // display initial value of courseName for each GradeBook
   cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
```

```
      << "\ngradeBook2 created for course: "    << gradeBook2.getCourseName()
      << endl;
   return 0; // indicate successful termination
} // end main
```

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

### Defining a Constructor

Lines 17–20 of Fig. 3.7 define a constructor for class GradeBook. Notice that 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 49–50). Line 17 indicates that class GradeBook's constructor has a string parameter called name . Note that line 17 does not specify a return type, because constructors cannot return values (or even void).

Line 19 in the constructor's body passes the constructor's parameter name to member function setCourseName, which assigns a value to data member courseName. The setCourseName member function (lines 23–26) simply assigns its parameter name to the data member courseName , so you might be wondering why we bother making the call to setCourseName in line 19—the constructor certainly could perform the assignment courseName = name. In Section 3.10, we modify setCourseName to perform validation (ensuring that, in this case, the courseName is 25 or fewer characters in length). At that point the benefits of calling setCourseName from the constructor will become clear. Note that both the constructor (line 17) and the setCourseName function (line 23) 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 46–57 of Fig. 3.7 define the main function that tests class GradeBook and demonstrates initializing GradeBook objects using a constructor. Line 49 in function main creates and initializes a GradeBook object called gradeBook1. When this line executes, the GradeBook constructor (lines 17–20) is called (implicitly by C++) with the argument "CS101 Introduction to C++ Programming" to initialize gradeBook1's course name. Line 50 repeats this process for the GradeBook object called gradeBook2, this time passing the argument "CS102 Data Structures in C++" to initialize gradeBook2's course name. Lines 53–54 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 copy of data member courseName.

### Two Ways to Provide a Default Constructor for a Class

Any constructor that takes no arguments is called a default constructor. A class gets a default constructor in one of two ways:

1. The compiler implicitly creates a default constructor in a class that does not define a constructor. Such a default constructor does not initialize the class's data members, but does call the default constructor for each

data member that is an object of another class. [*Note:* An uninitialized variable typically contains a "garbage" value (e.g., an uninitialized int variable might contain -858993460 , which is likely to be an incorrect value for that variable in most programs).]

2. You explicitly define a constructor that takes no arguments. Such a default constructor will perform the initialization specified by you and will call the default constructor for each data member that is an object of another class.

If you define a constructor with arguments, C++ will not implicitly create a default constructor for that class. Note that 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 a constructor to ensure that your class's data members are initialized with meaningful values when each new object of your class is created.*

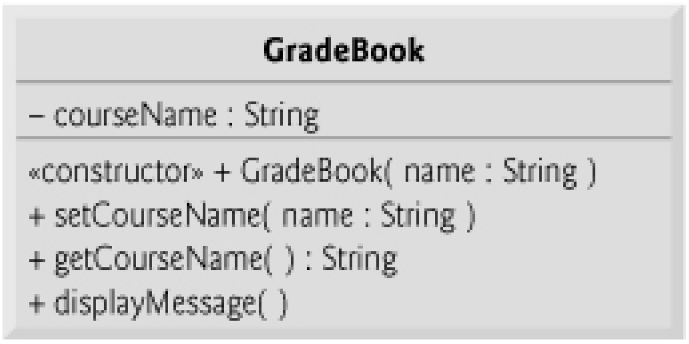
Software Engineering Observation 3.4

*Data members can be initialized in a constructor of the class, or their values may be* set *later after the object is created. However, it is a good software engineering practice to ensure that an object is fully initialized before the client code invokes the object's member functions. In general, 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 class GradeBook 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. It is customary to 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.**


GradeBook
– courseName : String
«constructor» + GradeBook( name : String )
+ setCourseName( name : String )
+ getCourseName( ) : String
+ displayMessage( )

## 3.8. Placing a Class in a Separate File for Reusability

We have developed class GradeBook as far as we need to for now from a programming perspective, so let's consider some software engineering issues. One of the benefits of creating class definitions is that, when packaged properly, our classes can be reused by programmers—potentially worldwide. For example, we can reuse C++ Standard Library type string in any C++ program by including the header file <string> in the program (and, as we'll see, by being able to link to the library's object code).

Unfortunately, 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. When they attempt to compile their programs, the compiler will indicate an error. For example, attempting to compile a program with two main functions in Microsoft Visual C++ 2008 produces the error

```
error C2084: function 'int main(void)' already has a body
```

when the compiler tries to compile the second main function it encounters. Similarly, the GNU C++ compiler produces the error

```
redefinition of 'int main()'
```

These errors indicate that a program already has a main function. 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.

### Header Files

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 is 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 file. Programs use #include preprocessor directives to include header files and take advantage of reusable software components, such as type string provided in the C++ Standard Library and user-defined types like class GradeBook.

In our next example, we separate 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 file in Fig. 3.9, notice that it contains only the GradeBook class definition (lines 11–41) and lines 3–8, which allow class GradeBook to use cout, endl and type string. The main function that uses class GradeBook is defined in the source-code file fig03_10.cpp (Fig. 3.10) in lines 10–21. 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 see how a source-code file with main can use the class definition found in a header file to create objects of a class.

**Fig. 3.9. GradeBook class definition.**

```
// Fig. 3.9: GradeBook.h
// GradeBook class definition in a separate file from main.
#include <iostream>
using std::cout;
using std::endl;

#include <string> // class GradeBook uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
public:
   // constructor initializes courseName with string supplied as argument
   GradeBook( string name )
   {
      setCourseName( name ); // call set function to initialize courseName
   } // end GradeBook constructor

   // function to set the course name
   void setCourseName( string name )
   {
      courseName = name; // store the course name in the object
   } // end function setCourseName

   // function to get the course name
   string getCourseName()
   {
      return courseName; // return object's courseName
   } // end function getCourseName

   // display a welcome message to the GradeBook user
   void displayMessage()
   {
      // call getCourseName to get the courseName
      cout << "Welcome to the grade book for\n" << getCourseName()
         << "!" << endl;
   } // end function displayMessage
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook
```

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

```
// Fig. 3.10: fig03_10.cpp
// Including class GradeBook from file GradeBook.h for use in main.
#include <iostream>
using std::cout;
using std::endl;

#include "GradeBook.h"   // include definition of class GradeBook

// function main begins program execution
int main()
{
   // create two GradeBook objects
   GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
   GradeBook gradeBook2( "CS102 Data Structures in C++" );

   // display initial value of courseName for each GradeBook
   cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
      << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
      << endl;
   return 0; // indicate successful termination
} // end main
```

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

### Including a Header File That Contains a User-Defined Class

A header file such as GradeBook.h (Fig. 3.9) cannot be used to begin program execution, because it does not contain a main function. If you try to compile and link GradeBook.h by itself to create an executable application, Microsoft Visual C++ 2005 produces the linker error message:

```
error LNK2019: unresolved external symbol _main referenced in
function _mainCRTStartup
```

To compile and link with GNU C++ on Linux, you must first include the header file in a .cpp source-code file, then GNU C++ produces a linker error message containing:

```
undefined reference to 'main'
```

This error indicates that the linker could not locate the program's 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.

Recall from Section 3.4 that, while the compiler knows what fundamental data types like int are, the compiler does not know what a GradeBook is because it is a user-defined type. In fact, the compiler does not 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 file. This enables the compiler to determine the amount of memory that it must reserve for each object of the class and ensure that a program calls the class's member functions correctly.

To create GradeBook objects gradeBook1 and gradeBook2 in lines 13–14 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 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 copy of the class's data members, because their contents can vary among objects (such as two different BankAccount objects having two different balance data members). 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 amount of memory required to store the class's data members. By including GradeBook.h in line 7, we give the compiler access to the information it needs (Fig. 3.9, line 40) to determine the size of a GradeBook object and to determine whether objects of the class are used correctly (in lines 13–14 and 17–18 of Fig. 3.10).

Line 7 instructs the C++ preprocessor to replace the directive with a copy of the contents of GradeBook.h (i.e., the GradeBook class definition) *before* the program is compiled. When the source-code file fig03_10.cpp is compiled, it now contains the GradeBook class definition (because of the #include), and the compiler is able to determine how to create GradeBook objects and see that their member functions are called correctly. Now that the class definition is in a header file (without a main function), we can include that header in *any* program that needs to reuse our GradeBook class.

### How Header Files Are Located

Notice that the name of the GradeBook.h header file in line 7 of Fig. 3.10 is enclosed in quotes (" ") rather than angle brackets (<>). Normally, a program's source-code files and user-defined header files are placed in the same directory. When the preprocessor encounters a header file name in quotes (e.g., "GradeBook.h"), the preprocessor attempts to locate the header file in the same directory as the file in which the #include directive appears. If the preprocessor cannot find the header file in that directory, it searches for it in the same location(s) as the C++ Standard Library header files. When the preprocessor encounters a header file name in angle brackets (e.g., <iostream>), it assumes that the header is part of the C++ Standard Library and does not look in the directory of the program that is being preprocessed.

Error-Prevention Tip 3.3

*To ensure that the preprocessor can locate header files correctly, #include preprocessor directives should place the names of user-defined header files in quotes (e.g., "GradeBook.h") and place the names of C++ Standard Library header files in angle brackets (e.g., <iostream>).*

### Additional Software Engineering Issues

Now that class GradeBook is defined in a header file, the class is reusable. Unfortunately, placing a class definition in a header file as in Fig. 3.9 still reveals the entire implementation of the class to the class's clients—GradeBook.h is simply a text file that anyone can open and read. Conventional software engineering wisdom says that to use an object of a class, the client code needs to know only what member functions to call, what arguments to provide to each member function and what return type to expect from each member function. The client code does not need to know how those functions are implemented.

If client code does know how a class is implemented, the client-code programmer might write client code based on the class's implementation details. Ideally, if that implementation changes, the class's clients should not have to change. Hiding the class's implementation details makes it easier to change the class's implementation while minimizing, and hopefully eliminating, changes to client code.

In Section 3.9 , we show how to break up the GradeBook class into two files so that

1. the class is reusable,
2. the clients of the class know what member functions the class provides, how to call them and what return types to expect, and
3. the clients do not know how the class's member functions are implemented.

## 3.9. Separating Interface from Implementation

In the preceding section, we showed how to promote software reusability by separating a class definition from the client code (e.g., function main ) that uses the class. We now introduce another fundamental principle of good software engineering—separating interface from implementation

### Interface of a Class

Interfaces define and standardize the ways in which things such as people and systems interact with one another. For example, a radio's controls serve as an interface between the radio's users and its internal components. The controls allow users to perform a limited set of operations (such as changing the station, adjusting the volume, and choosing between AM and FM stations). Various radios may implement these operations differently—some provide push buttons, some provide dials and some support voice commands. The interface specifies *what* operations a radio permits users to perform but does not specify *how* the operations are implemented inside the radio.

Similarly, the interface of a class describes *what* services a class's clients can use and how to *request* those services, but not *how* the class carries out the services. A class's interface consists of the class's public member functions (also known as the class's public services). For example, class GradeBook's interface (Fig. 3.9) contains a constructor and member functions setCourseName, getCourseName and displayMessage. GradeBook's clients (e.g., main in Fig. 3.10 ) use these functions to request the class's services. As you'll soon see, you can specify a class's interface by writing a class definition that lists only the member-function names, return types and parameter types.

### Separating the Interface from the Implementation

In our prior examples, each class definition contained the complete definitions of the class's public member functions and the declarations of its private data members. However, it is better software engineering to define member functions outside the class definition, so that their implementation details can be hidden from the client code. This practice ensures that programmers do not write client code that depends on the class's implementation details. If they were to do so, the client code would be more likely to "break" if the class's implementation changed.

The program of Figs. 3.11–3.13 separates class GradeBook 's interface from its implementation by splitting the class definition of Fig. 3.9 into two files—the header file GradeBook.h (Fig. 3.11 ) in which class GradeBook is defined, and the source-code file GradeBook.cpp (Fig. 3.12) in which GradeBook 's member functions are defined. By convention, member-function definitions are placed in a source-code file of the same base name (e.g., GradeBook) as the class's header file but with a cpp filename extension. The source-code file fig03_13.cpp (Fig. 3.13) defines function main (the client code). The code and output of Fig. 3.13 are identical to that of Fig. 3.10. Figure 3.14 shows how this three-file program is compiled from the perspectives of the GradeBook class programmer and the client-code programmer—we'll explain this figure in detail.

**Fig. 3.11. GradeBook class definition containing function prototypes that specify the interface of the class.**

```
// Fig. 3.11: GradeBook.h
// GradeBook class definition. This file presents GradeBook's public
// interface without revealing the implementations of GradeBook's member
// functions, which are defined in GradeBook.cpp.
#include <string> // class GradeBook uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
public:
   GradeBook( string ); // constructor that initializes courseName
   void setCourseName( string ); // function that sets the course name
   string getCourseName(); // function that gets the course name
   void displayMessage(); // function that displays a welcome message
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook
```

**Fig. 3.12.** **GradeBook** **member-function definitions represent the implementation of class** **GradeBook.**

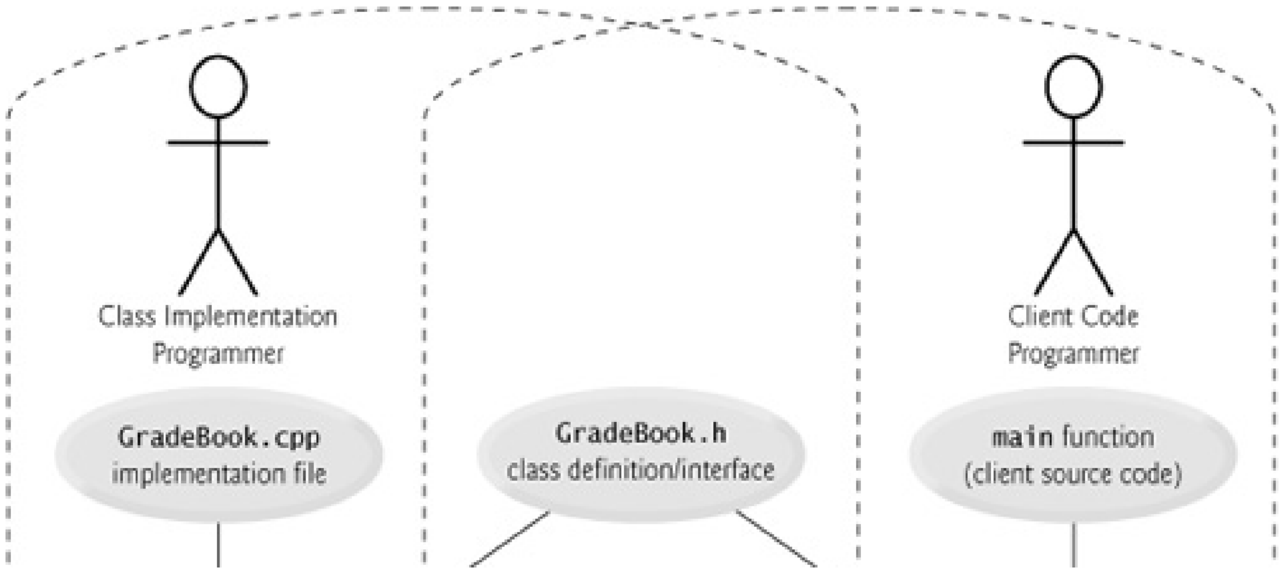
```
// Fig. 3.12: GradeBook.cpp
// GradeBook member-function definitions. This file contains
// implementations of the member functions prototyped in GradeBook.h.
#include <iostream>
using std::cout;
using std::endl;

#include "GradeBook.h" // include definition of class GradeBook

// constructor initializes courseName with string supplied as argument
GradeBook::GradeBook( string name )
{
   setCourseName( name ); // call set function to initialize courseName
} // end GradeBook constructor

// function to set the course name
void GradeBook::setCourseName( string name )
{
   courseName = name; // store the course name in the object
} // end function setCourseName

// function to get the course name
string GradeBook::getCourseName()
{
   return courseName; // return object's courseName
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   // call getCourseName to get the courseName
   cout << "Welcome to the grade book for\n" << getCourseName()
      << "!" << endl;
} // end function displayMessage
```

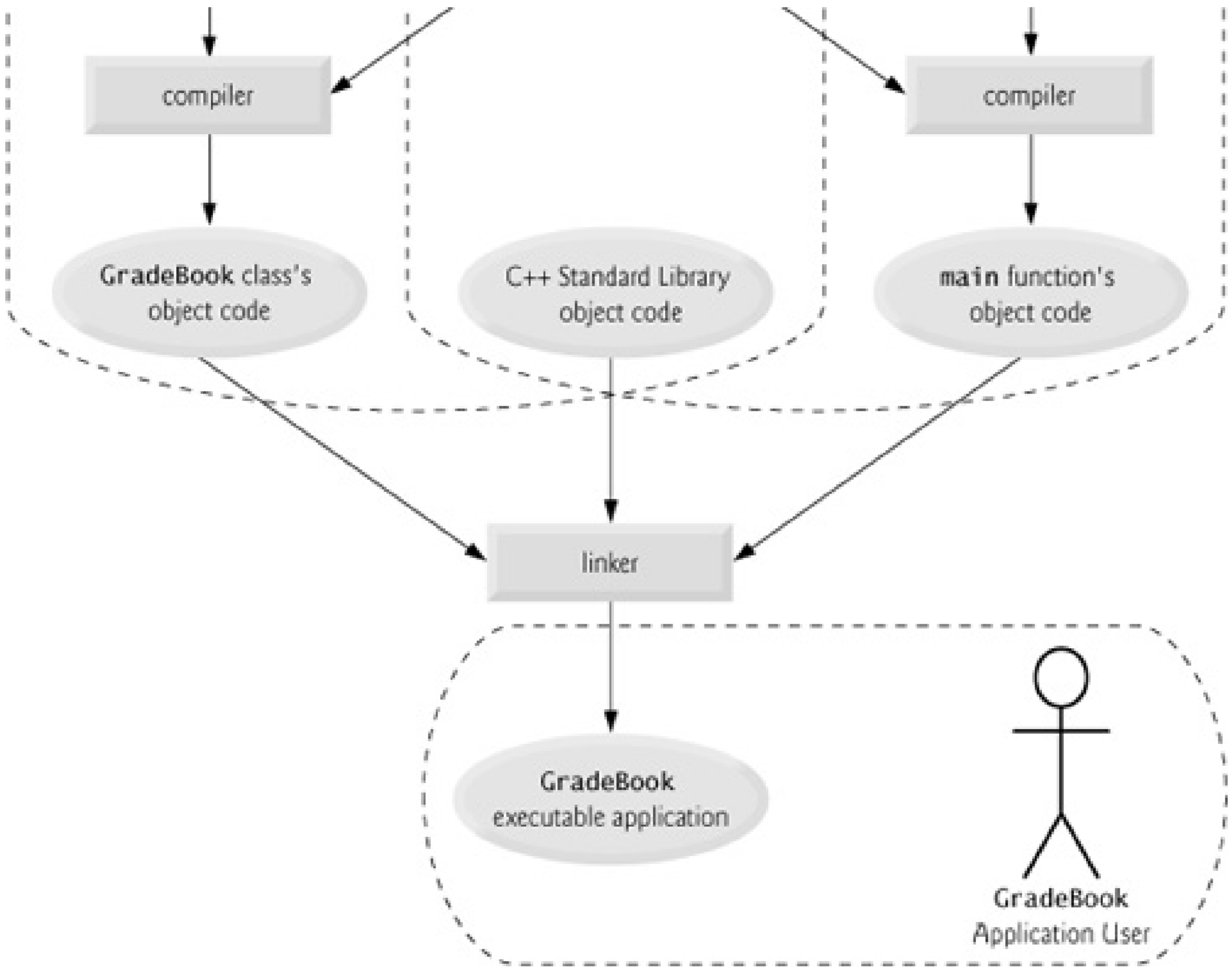
**Fig. 3.13. GradeBook class demonstration after separating its interface from its implementation.**

```
// Fig. 3.13: fig03_13.cpp
// GradeBook class demonstration after separating
// its interface from its implementation.
#include <iostream>
using std::cout;
using std::endl;

#include "GradeBook.h" // include definition of class GradeBook

// function main begins program execution
int main()
{
   // create two GradeBook objects
   GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
   GradeBook gradeBook2( "CS102 Data Structures in C++" );

   // display initial value of courseName for each GradeBook
   cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
      << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
      << endl;
   return 0; // indicate successful termination
} // end main
```

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

**Fig. 3.14. Compilation and linking process that produces an executable application.**


Class Implementation Programmer
GradeBook.cpp implementation file
GradeBook.h class definition/interface
Client Code Programmer
main function (client source code)


compiler
compiler
GradeBook class's object code
C++ Standard Library object code
main function's object code
linker
GradeBook executable application
GradeBook Application User

GradeBook.h: **Defining a Class's Interface with Function Prototypes**

Header file GradeBook.h (Fig. 3.11) contains another version of GradeBook's class definition (lines 9–18). This version is similar to the one in Fig. 3.9, but the function definitions in Fig. 3.9 are replaced here with function prototypes (lines 12–15) that describe the class's public interface without revealing the class's member-function implementations. A function prototype is a declaration of a function that tells the compiler the function's name, its return type and the types of its parameters. Note that the header file still specifies the class's private data member (line 17) as well. Again, the compiler must know the data members of the class to determine how much memory to reserve for each object of the class. Including the header file GradeBook.h in the client code (line 8 of Fig. 3.13) provides the compiler with the information it needs to ensure that the client code calls the member functions of class GradeBook correctly.

The function prototype in line 12 (Fig. 3.11) indicates that the constructor requires one string parameter. Recall that constructors do not have return types, so no return type appears in the function prototype. Member function setCourseName's function prototype (line 13) indicates that setCourseName requires a string parameter and does not return a value (i.e., its return type is void). Member function getCourseName's function prototype (line 14) indicates that the function does not require parameters and returns a string. Finally, member function displayMessage's function prototype (line 15) specifies that displayMessage does not require parameters and does not return a value. These function prototypes are the same as the corresponding function headers in Fig. 3.9, except that the parameter names (which are optional in prototypes) are not included and each function prototype must end with a semicolon.

Common Programming Error 3.8

*Forgetting the semicolon at the end of a function prototype is a syntax error.*

Good Programming Practice 3.5

*Although parameter names in function prototypes are optional (they are ignored by the compiler), many programmers use these names for documentation purposes.*

Error-Prevention Tip 3.4

*Parameter names in a function prototype (which, again, are ignored by the compiler) can be misleading if wrong or confusing names are used. For this reason, many programmers create function prototypes by copying the first line of the corresponding function definitions (when the source code for the functions is available), then appending a semicolon to the end of each prototype.*

## GradeBook.cpp: Defining Member Functions in a Separate Source-Code File

**GradeBook.cpp** (Fig. 3.12) defines class GradeBook 's member functions, which were declared in lines 12–15 of Fig. 3.11 . The member-function definitions appear in lines 11–34 and are nearly identical to the member-function definitions in lines 15–38 of Fig. 3.9.

Notice that each member-function name in the function headers (lines 11, 17, 23 and 29) is preceded by the class name and ::, which is known as the binary scope resolution operator. This "ties" each member function to the (now separate) GradeBook class definition (Fig. 3.11 ), which declares the class's member functions and data members. Without "GradeBook:: " preceding each function name, these functions would not be recognized by the compiler as member functions of class GradeBook—the compiler would consider them "free" or "loose" functions, like main. Such functions cannot access GradeBook's private data or call the class's member functions, without specifying an object. So, the compiler would not be able to compile these functions. For example, lines 19 and 25 that access variable courseName would cause compilation errors because courseName is not declared as a local variable in each function—the compiler would not know that courseName is already declared as a data member of class GradeBook.

Common Programming Error 3.9

*When defining a class's member functions outside that class, omitting the class name and binary scope resolution operator (* :: *) preceding the function names causes compilation errors.*

To indicate that the member functions in GradeBook.cpp are part of class GradeBook, we must first include the GradeBook.h header file (line 8 of Fig. 3.12). This allows us to access the class name GradeBook in the GradeBook.cpp file. When compiling

GradeBook.cpp, the compiler uses the information in GradeBook.h to ensure that

1. the first line of each member function (lines 11, 17, 23 and 29) matches its prototype in the GradeBook.h file—for example, the compiler ensures that getCourseName accepts no parameters and returns a string, and that
2. each member function knows about the class's data members and other member functions—for example, lines 19 and 25 can access variable courseName because it is declared in GradeBook.h as a data member of class GradeBook, and lines 13 and 32 can call functions setCourseName and getCourseName, respectively, because each is declared as a member function of the class in GradeBook.h (and because these calls conform with the corresponding prototypes).

### Testing Class GradeBook

Figure 3.13 performs the same GradeBook object manipulations as Fig. 3.10. Separating GradeBook's interface from the implementation of its member functions does not affect the way that this client code uses the class. It affects only how the program is compiled and linked, which we discuss in detail shortly.

As in Fig. 3.10, line 8 of Fig. 3.13 includes the GradeBook.h header file so that the compiler can ensure that GradeBook objects are created and manipulated correctly in the client code. Before executing this program, the source-code files in Fig. 3.12 and Fig. 3.13 must both be compiled, then linked together—that is, the member-function calls in the client code need to be tied to the implementations of the class's member functions—a job performed by the linker.

### The Compilation and Linking Process

The diagram in Fig. 3.14 shows the compilation and linking process that results in an executable GradeBook application that can be used by instructors. Often a class's interface and implementation will be created and compiled by one programmer and used by a separate programmer who implements the client code that uses the class. So, the diagram shows what is required by both the class-implementation programmer and the client-code programmer. The dashed lines in the diagram show the pieces required by the class-implementation programmer, the client-code programmer and the GradeBook application user, respectively. [*Note:* Figure 3.14 is not a UML diagram.]

A class-implementation programmer responsible for creating a reusable GradeBook class creates the header file GradeBook.h and the source-code file GradeBook.cpp that #includes the header file, then compiles the source-code file to create GradeBook's object code. To hide class GradeBook's member-function implementation details, the class-implementation programmer would provide the client-code programmer with the header file GradeBook.h (which specifies the class's interface and data members) and the object code for class GradeBook (which contains the machine-language instructions that represent GradeBook's member functions). The client-code programmer is not given GradeBook.cpp, so the client remains unaware of how GradeBook's member functions are implemented.

The client code needs to know only GradeBook's interface to use the class and must be able to link its object code. Since the interface of the class is part of the class definition in the GradeBook.h header file, the client-code programmer must have access to this file and #include it in the client's source-code file. When the client code is compiled, the compiler uses the class definition in GradeBook.h to ensure that the main function creates and manipulates objects of class GradeBook correctly.

To create the executable GradeBook application to be used by instructors, the last step is to link

1. the object code for the main function (i.e., the client code)
2. the object code for class GradeBook's member-function implementations
3. the C++ Standard Library object code for the C++ classes (e.g., string) used by the class-implementation programmer and the client-code programmer.

The linker's output is the executable GradeBook application that instructors can use to manage their students' grades.

For further information on compiling multiple-source-file programs, see your compiler's documentation. We provide links to various C++ compilers in our C++ Resource Center at www.deitel.com/cplusplus/.

## 3.10. Validating Data with *set* Functions

In Section 3.6, we introduced *set* functions for allowing clients of a class to modify the value of a private data member. In Fig. 3.5, class GradeBook defines member function setCourseName to simply assign a value received in its parameter name to data member courseName. This member function does not ensure that the course name adheres to any particular format or follows any other rules regarding what a "valid" course name looks like. As we stated earlier, suppose that a university can print student transcripts containing course names of only 25 characters or less. If the university uses a system containing GradeBook objects to generate the transcripts, we might want class GradeBook to ensure that its data member courseName never contains more than 25 characters. The program of Figs. 3.15–3.17 enhances class GradeBook's member function setCourseName to perform this validation.

### GradeBook **Class Definition**

Notice that GradeBook's class definition (Fig. 3.15)—and hence, its interface—is identical to that of Fig. 3.11 . Since the interface remains unchanged, clients of this class need not be changed when the definition of member function setCourseName is modified. This enables clients to take advantage of the improved GradeBook class simply by linking the client code to the updated GradeBook's object code.

**Fig. 3.15. GradeBook class definition.**

```
// Fig. 3.15: GradeBook.h
// GradeBook class definition presents the public interface of
// the class. Member-function definitions appear in GradeBook.cpp.
#include <string> // program uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
public:
   GradeBook( string ); // constructor that initializes a GradeBook object
   void setCourseName( string ); // function that sets the course name
   string getCourseName(); // function that gets the course name
   void displayMessage(); // function that displays a welcome message
private:
   string courseName; // course name for this GradeBook
}; // end class GradeBook
```

### Validating the Course Name with GradeBook Member Function setCourseName

The enhancement to class GradeBook is in the definition of setCourseName (Fig. 3.16, lines 18–31). The if statement in lines 20–21 determines whether parameter name contains a valid course name (i.e., a string of 25 or fewer characters). If the course name is valid, line 21 stores the course name in data member courseName. Note the expression name.length() in line 20. This is a member-function call just like myGradeBook.displayMessage(). The C++ Standard Library's string class defines a member function length that returns the number of characters in a string object. Parameter name is a string object, so the call name.length() returns the number of characters in name. If this value is less than or equal to 25, name is valid and line 21 executes.

**Fig. 3.16. Member-function definitions for class GradeBook with a *set* function that validates the length of data member courseName.**

```
// Fig. 3.16: GradeBook.cpp
// Implementations of the GradeBook member-function definitions.
// The setCourseName function performs validation.
#include <iostream>
using std::cout;
using std::endl;

#include "GradeBook.h" // include definition of class GradeBook

// constructor initializes courseName with string supplied as argument
GradeBook::GradeBook( string name )
{
   setCourseName( name ); // validate and store courseName
} // end GradeBook constructor

// function that sets the course name;
// ensures that the course name has at most 25 characters
void GradeBook::setCourseName( string name )
{
   if ( name.length() <= 25 ) // if name has 25 or fewer characters
      courseName = name; // store the course name in the object

   if ( name.length() > 25 ) // if name has more than 25 characters
   {
      // set courseName to first 25 characters of parameter name
      courseName = name.substr( 0, 25 ); // start at 0, length of 25

      cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
         << "Limiting courseName to first 25 characters.\n" << endl;
   } // end if
} // end function setCourseName

// function to get the course name
string GradeBook::getCourseName()
{
   return courseName; // return object's courseName
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   // call getCourseName to get the courseName
   cout << "Welcome to the grade book for\n" << getCourseName()
      << "!" << endl;
} // end function displayMessage
```

The if statement in lines 23–30 handles the case in which setCourseName receives an invalid course name (i.e., a name that is more than 25 characters long). Even if parameter name is too long, we still want to leave the GradeBook object in a consistent state —that is, a state in which the object's data member courseName contains a valid value (i.e., a string of 25 characters or less). Thus, we truncate (i.e., shorten) the specified course name and assign the first 25 characters of name to the courseName data member (unfortunately, this could truncate the course name awkwardly). Standard class string provides member function substr (short for "substring") that returns a new string object created by copying part of an existing string object. The call in line 26 (i.e., name.substr( 0, 25 )) passes two integers (0 and 25) to name's member function substr. These arguments indicate the portion of the string name that substr should return. The first argument specifies the starting position in the original string from which characters are copied—the first character in every string is considered to be at position 0. The second argument specifies the number of characters to copy. Therefore, the call in line 26 returns a 25-character substring of name starting at position 0 (i.e., the first 25 characters in name). For example, if name holds the value "CS101 Introduction to Programming in C++", substr returns "CS101 Introduction to Pro". After the call to substr, line 26 assigns the substring returned by substr to data member courseName. In this way, member function setCourseName ensures that courseName is always assigned a string containing 25 or fewer characters. If the member function has to truncate the course name to make it valid, lines 28–29 display a warning message.

Note that the if statement in lines 23–30 contains two body statements—one to set the courseName to the first 25 characters of parameter name and one to print an accompanying message to the user. We want both of these statements to execute when name is too long, so we place them in a pair of braces, {}. Recall from Chapter 2 that this creates a block. You'll learn more about placing multiple statements in the body of a control statement in Chapter 4.

Note that the statement in lines 28–29 could also appear without a stream insertion operator at the start of the second line of the statement, as in:

```
cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
  "Limiting courseName to first 25 characters.\n" << endl;
```

The C++ compiler combines adjacent string literals, even if they appear on separate lines of a program. Thus, in the statement above, the C++ compiler would combine the string literals "\" exceeds maximum length (25).\n" and "Limiting courseName to first 25 characters.\n" into a single string literal that produces output identical to that of lines 28–29 in Fig. 3.16 . This behavior allows you to print lengthy strings by breaking them across lines in your program without including additional stream insertion operations.

## Testing Class GradeBook

Figure 3.17 demonstrates the modified version of class GradeBook (Figs. 3.15–3.16 ) featuring validation. Line 14 creates a GradeBook object named gradeBook1 . Recall that the GradeBook constructor calls setCourseName to initialize data member courseName. In previous versions of the class, the benefit of calling setCourseName in the constructor was not evident. Now, however, the constructor takes advantage of the validation provided by setCourseName. The constructor simply calls setCourseName , rather than duplicating its validation code. When line 14 of Fig. 3.17 passes an initial course name of "CS101 Introduction to Programming in C++" to the GradeBook constructor, the constructor passes this value to setCourseName , where the actual initialization occurs. Because this course name contains more than 25 characters, the body of the second if statement executes, causing courseName to be initialized to the truncated 25-character course name "CS101 Introduction to Pro" (the truncated part is highlighted in bold black in line 14). Notice that the output in Fig. 3.17 contains the warning message output by lines 28–29 of Fig. 3.16 in member function setCourseName. Line 15 creates another GradeBook object called gradeBook2 —the valid course name passed to the constructor is exactly 25 characters.

**Fig. 3.17. Creating and manipulating a GradeBook object in which the course name is limited to 25 characters in length.**

```
// Fig. 3.17: fig03_17.cpp
// Create and manipulate a GradeBook object; illustrate validation.
#include <iostream>
using std::cout;
using std::endl;

#include "GradeBook.h" // include definition of class GradeBook

// function main begins program execution
int main()
{
   // create two GradeBook objects;
   // initial course name of gradeBook1 is too long
   GradeBook gradeBook1( "CS101 Introduction to Programming in C++" );
   GradeBook gradeBook2( "CS102 C++ Data Structures" );

   // display each GradeBook's courseName
   cout << "gradeBook1's initial course name is: "
      << gradeBook1.getCourseName()
      << "\ngradeBook2's initial course name is: "
      << gradeBook2.getCourseName() << endl;

   // modify myGradeBook's courseName (with a valid-length string)
   gradeBook1.setCourseName( "CS101 C++ Programming" );

   // display each GradeBook's courseName
   cout << "\ngradeBook1's course name is: "
      << gradeBook1.getCourseName()
      << "\ngradeBook2's course name is: "
      << gradeBook2.getCourseName() << endl;
   return 0; // indicate successful termination
} // end main
```

```
Name "CS101 Introduction to Programming in C++" exceeds maximum length (25).
Limiting courseName to first 25 characters.

gradeBook1's initial course name is: CS101 Introduction to Pro
gradeBook2's initial course name is: CS102 C++ Data Structures

gradeBook1's course name is: CS101 C++ Programming
gradeBook2's course name is: CS102 C++ Data Structures
```

Lines 18–21 of Fig. 3.17 display the truncated course name for gradeBook1 (we highlight this in bold black in the program output) and the course name for gradeBook2. Line 24 calls gradeBook1's setCourseName member function directly, to change the course name in the GradeBook object to a shorter name that does not need to be truncated. Then, lines 27–30 output the course names for the GradeBook objects again.

### Additional Notes on *Set* Functions

A public *set* function such as setCourseName should carefully scrutinize any attempt to modify the value of a data member (e.g., courseName) to ensure that the new value is appropriate for that data item. For example, an attempt to *set* the day of the month to 37 should be rejected, an attempt to *set* a person's weight to zero or a negative value should be rejected, an attempt to *set* a grade on an exam to 185 (when the proper range is zero to 100) should be rejected, and so on

Software Engineering Observation 3.5

*Making data members private and controlling access, especially write access, to those data members through public member functions helps ensure data integrity.*

Error-Prevention Tip 3.5

*The benefits of data integrity are not automatic simply because data members are made private —you must provide appropriate validity checking and report the errors.*

Software Engineering Observation 3.6

*Member functions that* set *the values of private data should verify that the intended new values are proper; if they are not, the* set *functions should place the private data members into an appropriate state.*

A class's *set* functions can return values to the class's clients indicating that attempts were made to assign invalid data to objects of the class. A client of the class can test the return value of a *set* function to determine whether the client's attempt to modify the object was successful and to take appropriate action. In Chapter 16 , we demonstrate how clients of a class can be notified via the exception-handling mechanism when an attempt is made to modify an object with an inappropriate value. To keep the program of Figs. 3.15–3.17 simple at this early point in the book, setCourseName in Fig. 3.16 just prints an appropriate message on the screen.

## 3.11. (Optional) Software Engineering Case Study: Identifying the Classes in the ATM Requirements Specification

Now we begin designing the ATM system that we introduced in Chapter 2 . In this section, we identify the classes that are needed to build the ATM system by analyzing the nouns and noun phrases that appear in the requirements specification. We introduce UML class diagrams to model the relationships between these classes. This is an important first step in defining the structure of our system.

### Identifying the Classes in a System

We begin our OOD process by identifying the classes required to build the ATM system. We'll eventually describe these classes using UML class diagrams and implement these classes in C++. First, we review the requirements specification of Section 2.7 and find key nouns and noun phrases to help us identify classes that comprise the ATM system. We may decide that some of these nouns and noun phrases are attributes of other classes in the system. We may also conclude that some of the nouns do not correspond to parts of the system and thus should not be modeled at all. Additional classes may become apparent to us as we proceed through the design process.

Figure 3.18 lists the nouns and noun phrases in the requirements specification. We list them from left to right in the order in which they appear in the requirements specification. We list only the singular form of each noun or noun phrase.

**Fig. 3.18. Nouns and noun phrases in the requirements specification.**

| Nouns and noun phrases in the requirements specification | | |
|---|---|---|
| bank | money / fund | account number |
| ATM | screen | PIN |
| user | keypad | bank database |
| customer | cash dispenser | balance inquiry |
| transaction | $20 bill / cash | withdrawal |
| account | deposit slot | deposit |
| balance | deposit envelope | |

We create classes only for the nouns and noun phrases that have significance in the ATM system. We do not need to model "bank" as a class, because the bank is not a part of the ATM system—the bank simply wants us to build the ATM. "Customer" and "user" also represent entities outside of the system—they are important because they interact with our ATM system, but we do not need to model them as classes in the ATM software. Recall that we modeled an ATM user (i.e., a bank customer) as the actor in the use case diagram of Fig. 2.14.

We do not model "$20 bill" or "deposit envelope" as classes. These are physical objects in the real world, but they are not part of what is being automated. We can adequately represent the presence of bills in the system using an attribute of the class that models the cash dispenser. (We assign attributes to classes in Section 4.11.) For example, the cash dispenser maintains a count of the number of bills it contains. The requirements specification doesn't say anything about what the system should do with deposit envelopes after it receives them. We can assume that acknowledging the receipt of an envelope—an operation performed by the class that models the deposit slot—is sufficient to represent the presence of an envelope in the system. (We assign operations to classes in Section 6.22.)

In our simplified ATM system, representing various amounts of "money," including the "balance" of an account, as attributes of other classes seems most appropriate. Likewise, the nouns "account number" and "PIN" represent significant pieces of information in the ATM system. They are important attributes of a bank account. They do not, however, exhibit behaviors. Thus, we can most appropriately model them as attributes of an account class.

Though the requirements specification frequently describes a "transaction" in a general sense, we do not model the broad notion of a financial transaction at this time. Instead, we model the three types of transactions (i.e., "balance inquiry," "withdrawal" and "deposit") as individual classes. These classes possess specific attributes needed for executing the transactions they represent. For example, a withdrawal needs to know the amount of money the user wants to withdraw. A balance inquiry, however, does not require any additional data. Furthermore, the three transaction classes exhibit unique behaviors. A withdrawal includes dispensing cash to the user, whereas a deposit involves receiving deposit envelopes from the user. [*Note:* In Section 13.10 , we "factor out" common features of all transactions into a general "transaction" class using the object-oriented concepts of abstract classes and inheritance.]

We determine the classes for our system based on the remaining nouns and noun phrases from Fig. 3.18 . Each of these refers to one or more of the following:

- ATM
- screen
- keypad
- cash dispenser
- deposit slot
- account
- bank database
- balance inquiry
- withdrawal
- deposit

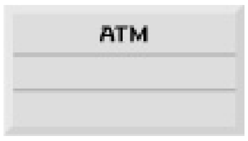
The elements of this list are likely to be classes we'll need to implement our system.

We can now model the classes in our system based on the list we have created. We capitalize class names in the design process—a UML convention—as we'll do when we write the actual C++ code that implements our design. If the name of a class contains more than one word, we run the words together and capitalize the first letter of each word (e.g., MultipleWordName). Using this convention, we create classes ATM, Screen, Keypad, CashDispenser, DepositSlot, Account, BankDatabase, BalanceInquiry, Withdrawal and Deposit . We construct our system using all of these classes as building blocks. Before we begin building the system, however, we must gain a better understanding of how the classes relate to one another.

**Modeling Classes**

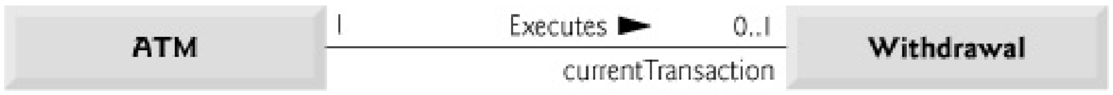
The UML enables us to model, via class diagrams, the classes in the ATM system and their interrelationships. Figure 3.19 represents class ATM . In the UML, each class is modeled as a rectangle with three compartments. The top compartment contains the name of the class, centered horizontally and in boldface. The middle compartment contains the class's attributes. (We discuss attributes in Section 4.11 and Section 5.10 .) The bottom compartment contains the class's operations (discussed in Section 6.22). In Fig. 3.19 the middle and bottom compartments are empty, because we have not yet determined this class's attributes and operations.

**Fig. 3.19. Representing a class in the UML using a class diagram.**



Class diagrams also show the relationships among the classes of the system. Figure 3.20 shows how our classes ATM and Withdrawal relate to one another. For the moment, we choose to model only this subset of classes for simplicity; we present a more complete class diagram later in this section. Notice that the rectangles representing classes in this diagram are not subdivided into compartments. The UML allows the suppression of class attributes and operations in this manner, when appropriate, to create more readable diagrams. Such a diagram is said to be an elided diagram —one in which some information, such as the contents of the second and third compartments, is not modeled. We'll place information in these compartments in Section 4.11 and Section 6.22.

**Fig. 3.20. Class diagram showing an association among classes.**



In Fig. 3.20, the solid line that connects the two classes represents an association —a relationship between classes. The numbers near each end of the line are multiplicity values, which indicate how many objects of each class participate in the association. In this case, following the line from one end to the other reveals that, at any given moment, one ATM object participates in an association with either zero or one Withdrawal objects—zero if the current user is not currently performing a transaction or has requested a different type of transaction, and one if the user has requested a withdrawal. The UML can model many types of multiplicity. Figure 3.21 lists and explains the multiplicity types.

**Fig. 3.21. Multiplicity types.**

| **Symbol** | **Meaning** |
|---|---|
| 0 | None |
| 1 | One |
| *m* | An integer value |
| 0..1 | Zero or one |
| *m, n* | *m* or *n* |
| *m..n* | At least *m* , but not more than *n* |
| * | Any nonnegative integer (zero or more) |
| 0..* | Zero or more (identical to *) |
| 1..* | One or more |

An association can be named. For example, the word Executes above the line connecting classes ATM and Withdrawal in Fig. 3.20 indicates the name of that association. This part of the diagram reads "one object of class ATM executes zero or one objects of class Withdrawal ." Note that association names are directional, as indicated by the filled arrowhead—so it would be improper, for example, to read the preceding association from right to left as "zero or one objects of class Withdrawal execute one object of class ATM."

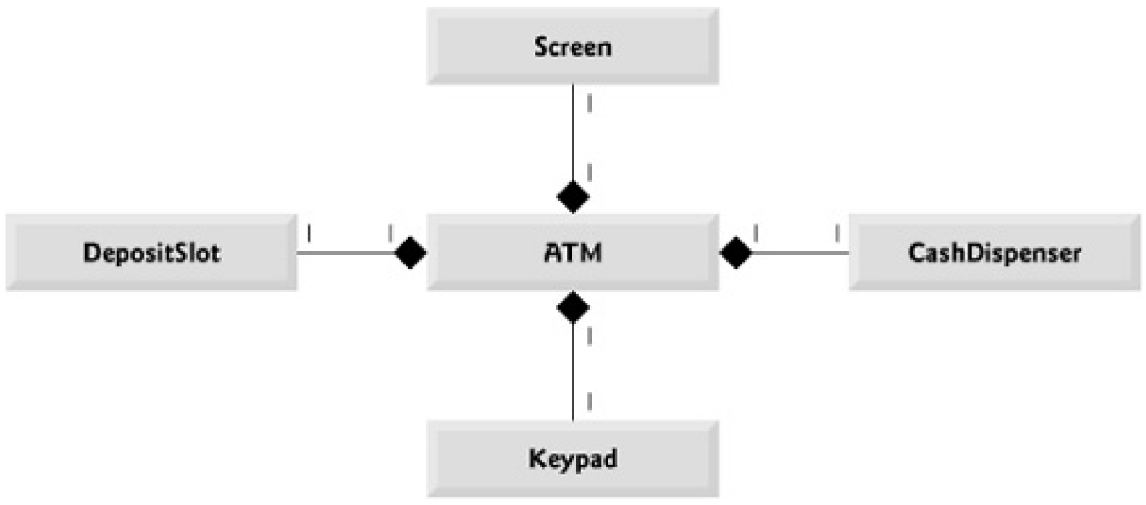
The word currentTransaction at the Withdrawal end of the association line in Fig. 3.20 is a role name, which identifies the role the Withdrawal object plays in its relationship with the ATM . A role name adds meaning to an association between classes by identifying the role a class plays in the context of an association. A class can play several roles in the same system. For example, in a school personnel system, a person may play the role of "professor" when relating to students. The same person may take on the role of "colleague" when participating in a relationship with another professor, and "coach" when coaching student athletes. In Fig. 3.20 , the role name currentTransaction indicates that the Withdrawal object participating in the Executes association with an object of class ATM represents the transaction currently being processed by the ATM. In other contexts, a Withdrawal object may take on other roles (e.g., the previous transaction). Notice that we do not specify a role name for the ATM end of the Executes association. Role names in class diagrams are often omitted when the meaning of an association is clear without them.

In addition to indicating simple relationships, associations can specify more complex relationships, such as objects of one class being composed of objects of other classes. Consider a real-world automated teller machine. What "pieces" does a manufacturer put together to build a working ATM? Our requirements specification tells us that the ATM is composed of a screen, a keypad, a cash dispenser and a deposit slot.

In Fig. 3.22, the solid diamonds attached to the association lines of class ATM indicate that class ATM has a composition relationship with classes Screen, Keypad, CashDispenser and DepositSlot. Composition implies a whole/part relationship. The class that has the composition symbol (the solid diamond) on its end of the association line is the whole (in this case, ATM ), and the classes on the other end of the association lines are the parts—in this case, classes Screen, Keypad, CashDispenser and DepositSlot. The compositions in Fig. 3.22 indicate that an object of class ATM is formed from one object of class Screen , one object of class CashDispenser , one object of class Keypad and one object of class DepositSlot. The ATM "has a" screen, a keypad, a cash dispenser and a deposit slot. The has-a relationship defines composition.

(We'll see in the Software Engineering Case Study section in Chapter 13 that the *is-a* relationship defines inheritance.)

**Fig. 3.22. Class diagram showing composition relationships.**



According to the UML specification, composition relationships have the following properties:

1. Only one class in the relationship can represent the whole (i.e., the diamond can be placed on only one end of the association line). For example, either the screen is part of the ATM or the ATM is part of the screen, but the screen and the ATM cannot both represent the whole in the relationship.

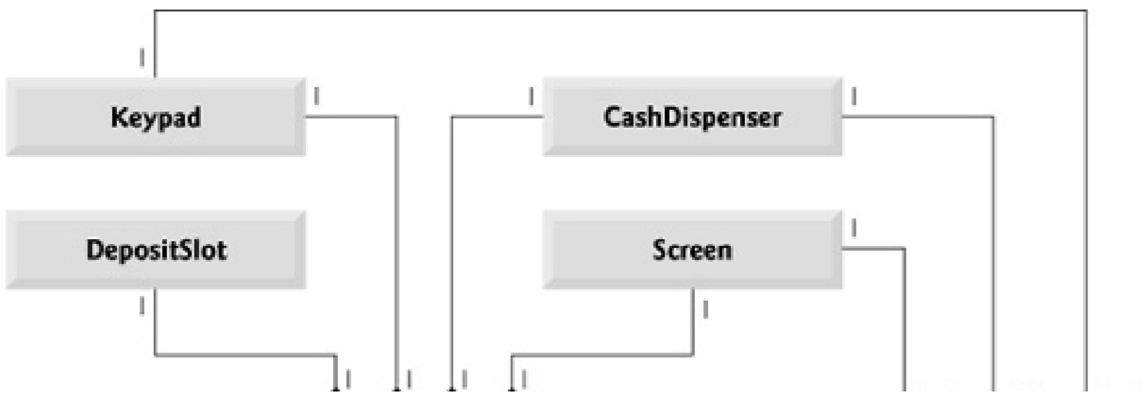
2. The parts in the composition relationship exist only as long as the whole, and the whole is responsible for the creation and destruction of its parts. For example, the act of constructing an ATM includes manufacturing its parts. Furthermore, if the ATM is destroyed, its screen, keypad, cash dispenser and deposit slot are also destroyed.

3. A part may belong to only one whole at a time, although the part may be removed and attached to another whole, which then assumes responsibility for the part.

The solid diamonds in our class diagrams indicate composition relationships that fulfill these three properties. If a *has-a* relationship does not satisfy one or more of these criteria, the UML specifies that hollow diamonds be attached to the ends of association lines to indicate aggregation—a weaker form of composition. For example, a personal computer and a computer monitor participate in an aggregation relationship—the computer has a monitor, but the two parts can exist independently, and the same monitor can be attached to multiple computers at once, thus violating the second and third properties of composition.

Figure 3.23 shows a class diagram for the ATM system. This diagram models most of the classes that we identified earlier in this section, as well as the associations between them that we can infer from the requirements specification. [*Note:* Classes `BalanceInquiry` and `Deposit` participate in associations similar to those of class `Withdrawal`, so we have chosen to omit them from this diagram to keep it simple. In Chapter 13, we expand our class diagram to include all the classes in the ATM system.]

**Fig. 3.23. Class diagram for the ATM system model.**



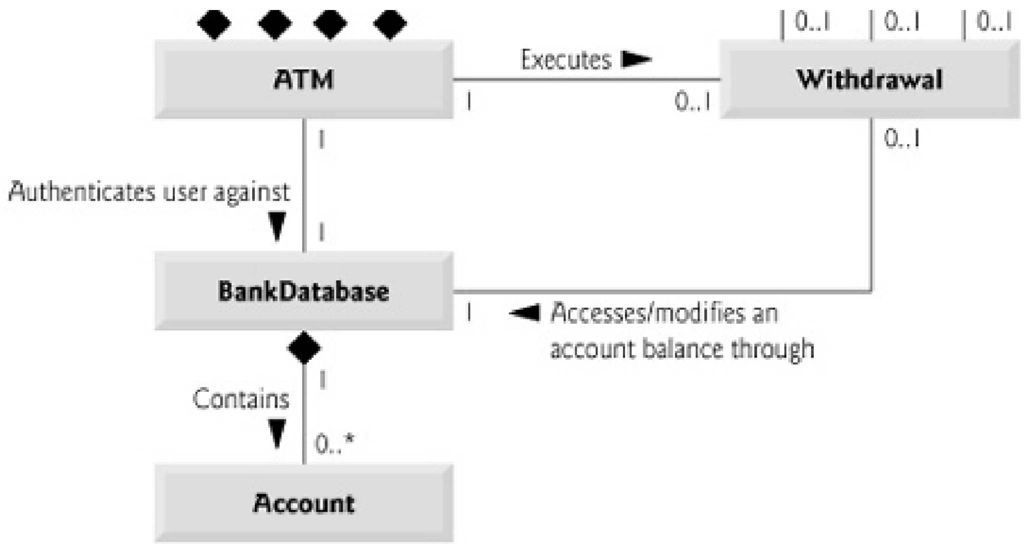


Figure 3.23 presents a graphical model of the structure of the ATM system. This class diagram includes classes BankDatabase and Account and several associations that were not present in either Fig. 3.20 or Fig. 3.22. The class diagram shows that class ATM has a one-to-one relationship with class BankDatabase—one ATM object authenticates users against one BankDatabase object. In Fig. 3.23, we also model the fact that the bank's database contains information about many accounts—one object of class BankDatabase participates in a composition relationship with zero or more objects of class Account. Recall from Fig. 3.21 that the multiplicity value 0..* at the Account end of the association between class BankDatabase and class Account indicates that zero or more objects of class Account take part in the association. Class BankDatabase has a one-to-many relationship with class Account—the BankDatabase contains many Accounts. Similarly, class Account has a many-to-one relationship with class BankDatabase—there can be many Accounts contained in the BankDatabase. [*Note:* Recall from Fig. 3.21 that the multiplicity value * is identical to 0..*. We include 0..* in our class diagrams for clarity.]

Figure 3.23 also indicates that if the user is performing a withdrawal, "one object of class Withdrawal accesses/modifies an account balance through one object of class BankDatabase." We could have created an association directly between class Withdrawal and class Account. The requirements specification, however, states that the "ATM must interact with the bank's account information database" to perform transactions. A bank account contains sensitive information, and systems engineers must always consider the security of personal data when designing a system. Thus, only the BankDatabase can access and manipulate an account directly. All other parts of the system must interact with the database to retrieve or update account information (e.g., an account balance).

The class diagram in Fig. 3.23 also models associations between class Withdrawal and classes Screen, CashDispenser and Keypad. A withdrawal transaction includes prompting the user to choose a withdrawal amount and receiving numeric input. These actions require the use of the screen and the keypad, respectively. Furthermore, dispensing cash to the user requires access to the cash dispenser.

Classes BalanceInquiry and Deposit, though not shown in Fig. 3.23, take part in several associations with the other classes of the ATM system. Like class Withdrawal, each of these classes associates with classes ATM and BankDatabase. An object of class BalanceInquiry also associates with an object of class Screen to display the balance of an account to the user. Class Deposit associates with classes Screen, Keypad and DepositSlot. Like withdrawals, deposit transactions require use of the screen and the keypad to display prompts and receive input, respectively. To receive deposit envelopes, an object of class Deposit accesses the deposit slot.

We have now identified the classes in our ATM system (although we may discover others as we proceed with the design and implementation). In Section 4.11, we determine the attributes for each of these classes, and in Section 5.10, we use these attributes to examine how the system changes over time. In Section 6.22, we determine the operations of the classes in our system.

**Software Engineering Case Study Self-Review Exercises**

**3.1** Suppose we have a class Car that represents a car. Think of some of the different pieces that a manufacturer would put together to produce a whole car. Create a class diagram (similar to Fig. 3.22) that models some of the composition relationships of class Car.

**3.2** Suppose we have a class File that represents an electronic document in a standalone, non-networked computer represented by class Computer. What sort of association exists between class Computer and class File?

- **a.** Class Computer has a one-to-one relationship with class File.
- **b.** Class Computer has a many-to-one relationship with class File.
- **c.** Class Computer has a one-to-many relationship with class File.
- **d.** Class Computer has a many-to-many relationship with class File.

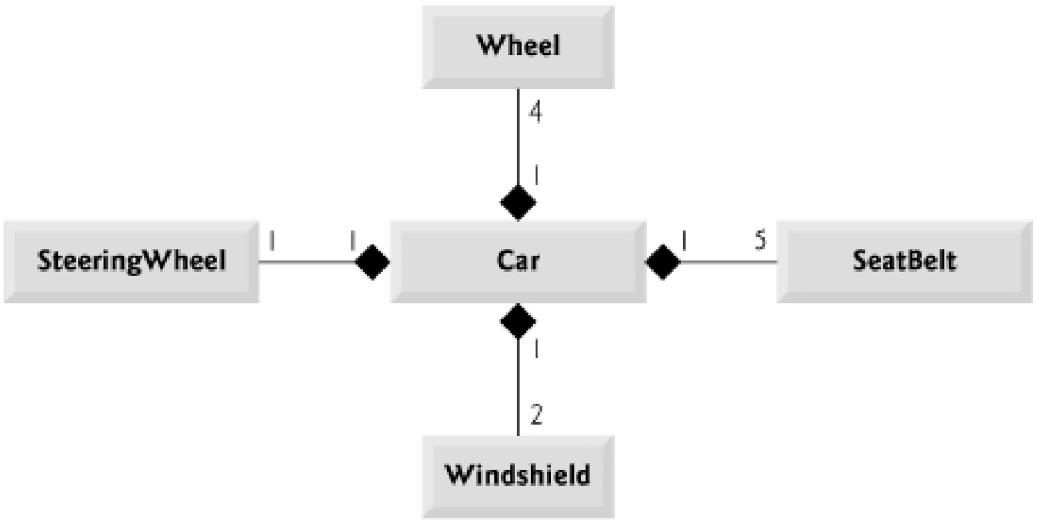
**3.3** State whether the following statement is *true* or *false*, and if *false* , explain why: A UML diagram in which a class's second and third compartments are not modeled is said to be an elided diagram.

**3.4** Modify the class diagram of Fig. 3.23 to include class Deposit instead of class Withdrawal.

**Answers to Software Engineering Case Study Self-Review Exercises**

**3.1** [*Note:* Answers may vary.] Figure 3.24 presents a class diagram that shows some of the composition relationships of a class Car.

**Fig. 3.24. Class diagram showing composition relationships of a class Car.**

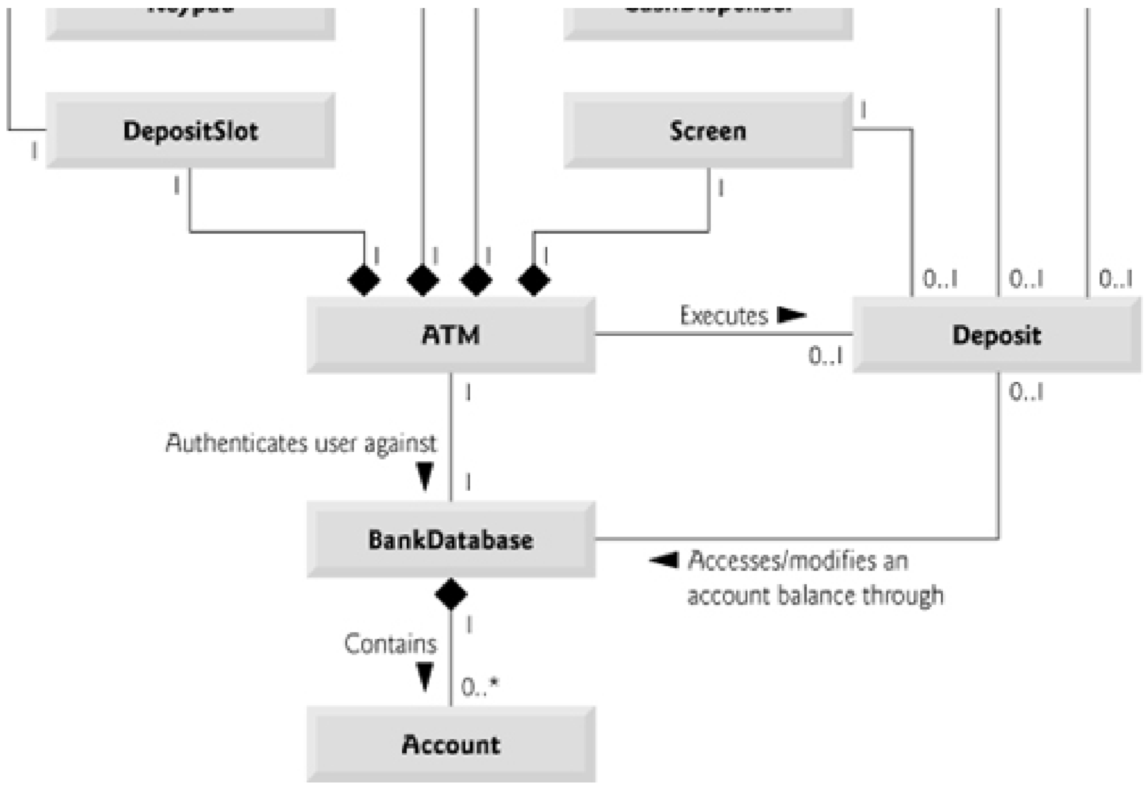


**3.2** c. [*Note:* In a computer network, this relationship could be many-to-many.]

**3.3** True.

**3.4** Figure 3.25 presents a class diagram for the ATM including class Deposit instead of class Withdrawal (as in Fig. 3.23). Note that Deposit does not access CashDispenser, but does access DepositSlot.

**Fig. 3.25. Class diagram for the ATM system model including class Deposit.**


DepositSlot
Screen
ATM
Executes
Deposit
Authenticates user against
BankDatabase
Accesses/modifies an account balance through
Contains
Account
1
0..1
0..*

## 3.12. Wrap-Up

In this chapter, you learned how to create user-defined classes, and how to create and use objects of those classes. We declared data members of a class to maintain data for each object of the class. We also defined member functions that operate on that data. You learned how to call an object's member functions to request the services the object provides and how to pass data to those member functions as arguments. We discussed the difference between a local variable of a member function and a data member of a class. We also showed how to use a constructor to specify initial values for an object's data members. You learned how to separate the interface of a class from its implementation to promote good software engineering. We presented a diagram that shows the files that class-implementation programmers and client-code programmers need to compile the code they write. We demonstrated how *set* functions can be used to validate an object's data and ensure that objects are maintained in a consistent state. In addition, UML class diagrams were used to model classes and their constructors, member functions and data members. In the next chapter, we begin our introduction to control statements, which specify the order in which a function's actions are performed.

# 4. Control Statements: Part 1

**Objectives**

In this chapter you'll learn:

- To use the if and if...else selection statements to choose among alternative actions.
- To use the while repetition statement to execute statements in a program repeatedly.
- Counter-controlled repetition and sentinel-controlled repetition.
- To use the increment, decrement and assignment operators.

Let's all move one place on.

—***Lewis Carroll***

The wheel is come full circle.

—***William Shakespeare***

How many apples fell on Newton's head before he took the hint!

—***Robert Frost***

All the evolution we know of proceeds from the vague to the definite.

—***Charles Sanders Peirce***

**Outline**

## 4.1. Introduction

In this chapter, we introduce C++'s if, if...else and while statements, three of the building blocks that allow you to specify the logic required for member functions to perform their tasks. We devote a portion of this chapter (and Chapters 5 and 7) to further developing the GradeBook class introduced in Chapter 3. In particular, we add a member function to the GradeBook class that uses control statements to calculate the average of a set of student grades. Another example demonstrates additional ways to combine control statements to solve a similar problem. We introduce C++'s assignment operators and explore C++'s increment and decrement operators. These additional operators abbreviate and simplify many program statements.

### 4.2. Control Structures

Böhm and Jacopini's research[1] demonstrated that all programs could be written in terms of only three control structures, namely, the sequence structure, the selection structure and the repetition structure. The term "control structures" comes from the field of computer science. When we introduce C++'s implementations of control structures, we'll refer to them in the terminology of the C++ standard document[2] as "control statements."

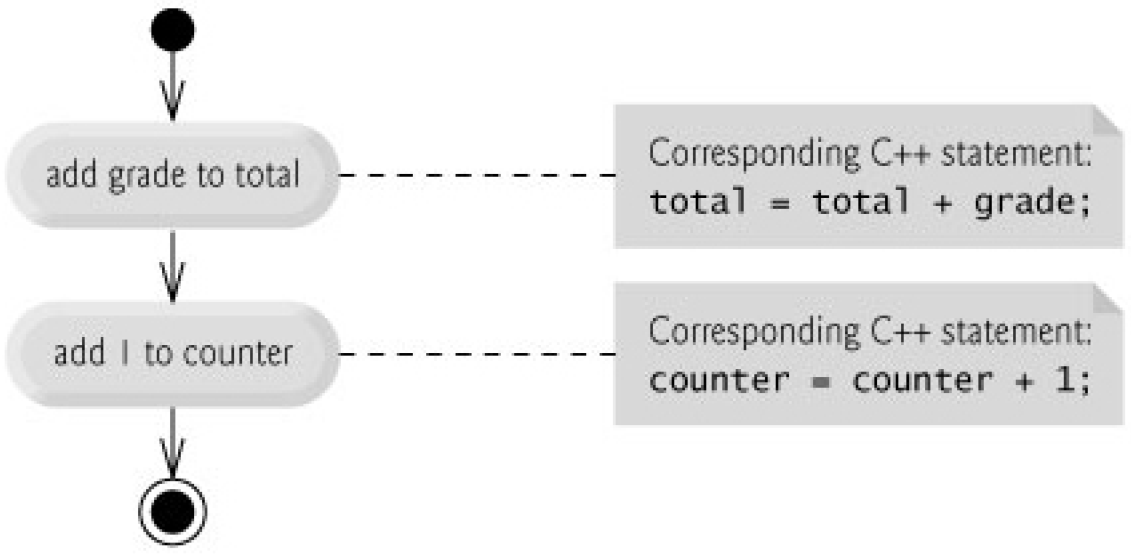
[1] Böhm, C., and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 366–371.

[2] This document is more specifically known as *INCITS/ISO/IEC 14882-2003 Programming languages—C++* and is available for download (for a fee) at webstore.ansi.org.

#### Sequence Structure in C++

The sequence structure is built into C++. Unless directed otherwise, C++ statements execute one after the other in the order in which they are written—that is, in sequence. The Unified Modeling Language (UML) activity diagram of Fig. 4.1 illustrates a typical sequence structure in which two calculations are performed in order. C++ allows us to have as many actions as we want in a sequence structure. As we'll soon see, anywhere a single action may be placed, we may place several actions in sequence.

**Fig. 4.1. Sequence-structure activity diagram.**



In this figure, the two statements involve adding a grade to a total variable and adding 1 to a counter variable. Such statements might appear in a program that averages several student grades. To calculate an average, the total of the grades is divided by the number of grades. A counter variable would be used to keep track of the number of values being averaged. You'll see similar statements in the program of Section 4.6.

Activity diagrams are part of the UML. An activity diagram models the workflow (also called the activity) of a portion of a software system. Such workflows may include a portion of an algorithm, such as the sequence structure in Fig. 4.1. Activity diagrams are composed of special-purpose symbols, such as action state symbols (a rectangle with its left and right sides replaced with arcs curving outward), diamonds and small circles; these symbols are connected by transition arrows, which represent the flow of the activity. Activity diagrams help you develop and represent algorithms. As you'll see, activity diagrams clearly show how control structures operate.

Consider the sequence-structure activity diagram of Fig. 4.1. It contains two action states that represent actions to perform. Each action state contains an action expression—e.g., "add grade to total" or "add 1 to counter"—that specifies a particular action to perform. Other actions might include calculations or input/output operations. The arrows in the activity diagram are called transition arrows. These arrows represent transitions, which indicate the order in which the actions represented by the action states occur—the program that implements the activities illustrated by the activity diagram in Fig. 4.1 first adds grade to total, then adds 1 to counter.

The solid circle located at the top of the activity diagram represents the activity's initial state—the beginning of the workflow before the program performs the modeled activities. The solid circle surrounded by a hollow circle that appears at the bottom of the activity diagram represents the final state—the end of the workflow after the program performs its activities.

Figure 4.1 also includes rectangles with the upper-right corners folded over. These are called notes in the UML. Notes are explanatory remarks that describe the purpose of symbols in the diagram. Notes can be used in any UML diagram—not just activity diagrams. Figure 4.1 uses UML notes to show the C++ code associated with each action state in the activity diagram. A dotted line connects each note with the element that the note describes. Activity diagrams normally do not show the C++ code that implements the activity. We use notes for this purpose here to illustrate how the diagram relates to C++ code. For more information on the UML, see our optional case study, which appears in the Software Engineering Case Study sections at the ends of Chapters 1–7, 9 and 13, or visit www.uml.org.

### Selection Statements in C++

C++ provides three types of selection statements (discussed in this chapter and Chapter 5). The if selection statement either performs (selects) an action if a condition (predicate) is true or skips the action if the condition is false. The if...else selection statement performs an action if a condition is true or performs a different action if the condition is false. The switch selection statement (Chapter 5) performs one of many different actions, depending on the value of an integer expression.

The if selection statement is a single-selection statement because it selects or ignores a single action (or, as we'll soon see, a single group of actions). The if...else statement is called a double-selection statement because it selects between two different actions (or groups of actions). The switch selection statement is called a multiple-selection statement because it selects among many different actions (or groups of actions).

### Repetition Statements in C++

C++ provides three types of repetition statements that enable programs to perform statements repeatedly as long as a condition remains true. The repetition statements are the while, do...while and for statements. (Chapter 5 presents the do...while and for statements.) The while and for statements perform the action (or group of actions) in their bodies zero or more times—if the loop-continuation condition is initially false, the action (or group of actions) will not execute. The do...while statement performs the action (or group of actions) in its body at least once.

Each of the words if, else, switch, while, do and for is a C++ keyword. These words are reserved by the C++ programming language to implement various features, such as C++'s control statements. Keywords must not be used as identifiers, such as variable names. Figure 4.2 provides a complete list of C++ keywords.

**Fig. 4.2. C++ keywords.**

| **C++ Keywords** | | | | |
|---|---|---|---|---|
| *Keywords common to the C and C++ programming languages* | | | | |
| *auto* | *break* | *case* | *char* | *const* |
| *continue* | *default* | *do* | *double* | *else* |
| *enum* | *extern* | *float* | *for* | *goto* |
| *if* | *int* | *long* | *register* | *return* |
| *short* | *signed* | *sizeof* | *static* | *struct* |
| *switch* | *typedef* | *union* | *unsigned* | *void* |
| *volatile* | *while* | | | |
| *C++-only keywords* | | | | |
| *and* | *and_eq* | *asm* | *bitand* | *bitor* |
| *bool* | *catch* | *class* | *compl* | *const_cast* |
| *delete* | *dynamic_cast* | *explicit* | *export* | *false* |
| *friend* | *inline* | *mutable* | *namespace* | *new* |
| *not* | *not_eq* | *operator* | *or* | *or_eq* |
| *private* | *protected* | *public* | *reinterpret_cast* | *static_cast* |
| *template* | *this* | *throw* | *true* | *try* |
| *typeid* | *typename* | *using* | *virtual* | *wchar_t* |
| *xor* | *xor_eq* | | | |

Common Programming Error 4.1

*Using a keyword as an identifier is a syntax error.*

Common Programming Error 4.2

*Spelling a keyword with any uppercase letters is a syntax error. All of C++'s keywords contain only lowercase letters.*

**Summary of Control Statements in C++**

C++ has only three kinds of control structures, which from this point forward we refer to as control statements: the sequence statement, selection statements (three types—if, if...else and switch) and repetition statements (three types—while, for and do...while ). As with the sequence statement of Fig. 4.1 , we can model each control statement as an activity diagram. Each diagram contains an initial state and a final state, which represent a control statement's entry point and exit point, respectively. These single-entry/single-exit control statements are attached to one another by connecting the exit point of one to the entry point of the next. We call this control-statement stacking . There is only one other way to connect control statements—called control-statement nesting , in which one control statement is contained inside another.

Software Engineering Observation 4.1

*Any C++ program we'll ever build can be constructed from only seven different types of control statements (sequence, if, if...else, switch, while, do...while and for ) combined in only two ways (control-statement stacking and control-statement nesting). This is the essence of simplicity.*

## 4.3. if Selection Statement

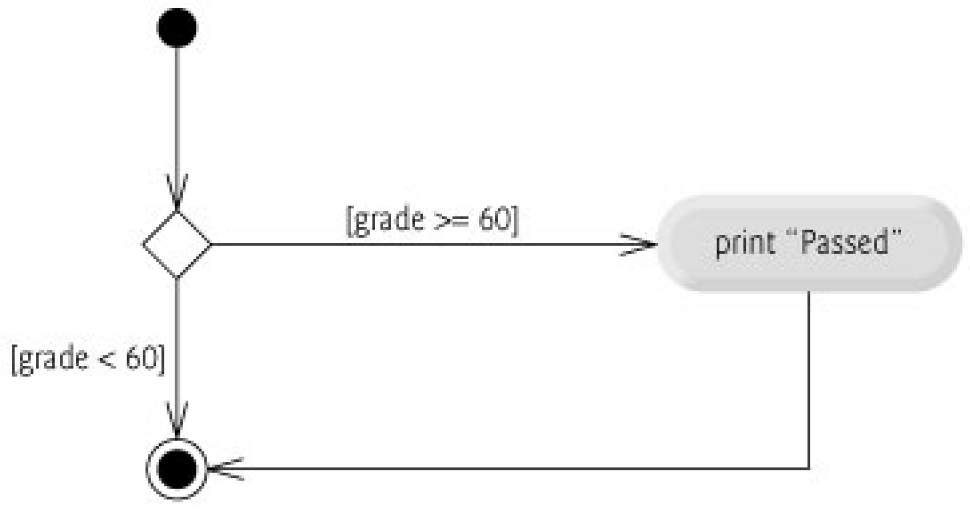
Programs use selection statements to choose among alternative courses of action. For example, suppose the passing grade on an exam is 60. The statement

```
if ( grade >= 60 )
   cout << "Passed" ;
```

determines whether the condition grade >= 60 is true or false. If it is true, "Passed" is printed and the next statement in order is performed. If the condition is false , the printing is ignored and the next statement in order is performed. Note that the second line of this selection statement is indented. Such indentation is optional, but recommended.

Figure 4.3 illustrates the single-selection if statement. It contains what is perhaps the most important symbol in an activity diagram—the diamond or decision symbol , which indicates that a decision is to be made. A decision symbol indicates that the workflow will continue along a path determined by the symbol's associated guard conditions , which can be true or false. Each transition arrow emerging from a decision symbol has a guard condition (specified in square brackets above or next to the transition arrow). If a particular guard condition is true, the workflow enters the action state to which that transition arrow points. In Fig. 4.3 , if the grade is greater than or equal to 60, the program prints "Passed" to the screen, then transitions to the final state of this activity. If the grade is less than 60, the program immediately transitions to the final state without displaying a message.

**Fig. 4.3. if single-selection statement activity diagram.**



In C++, a decision can be based on any expression—if the expression evaluates to zero, it is treated as false; if the expression evaluates to nonzero, it is treated as true. C++ provides the data type bool for variables that can hold only the values true and false —each of these is a C++ keyword.

Portability Tip 4.1

*For compatibility with earlier versions of C, which used integers for Boolean values, the bool value true also can be represented by any nonzero value (compilers typically use 1) and the bool value false also can be represented as the* *value zero.*

## 4.4. if...else Double-Selection Statement

The if single-selection statement performs an indicated action only when the condition is true ; otherwise the action is skipped. The if...else double-selection statement allows you to specify an action to perform when the condition is true and a different action to perform when the condition is false. For example, the statement

```
if ( grade >= 60 )
  cout << "Passed" ;
else
  cout << "Failed" ;
```

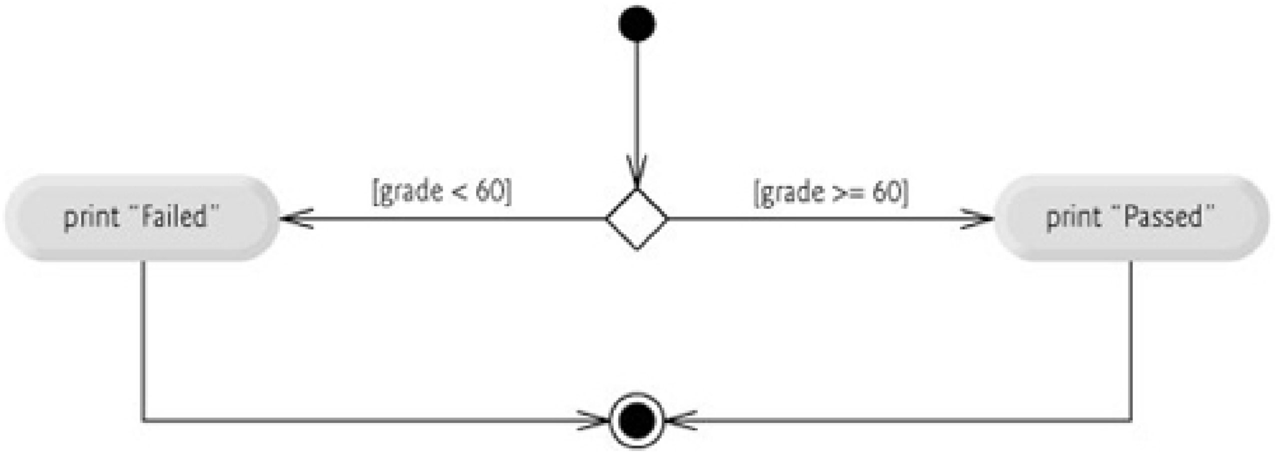
prints "Passed" if the condition grade >= 60 is true, but prints "Failed" if the condition is false (i.e., the grade is less than 60). In either case, after printing occurs, the next statement in sequence is performed.

Good Programming Practice 4.1

*Indent both body statements of an if...else statement.*

Figure 4.4 illustrates the flow of control in the if...else statement. Once again, note that (besides the initial state, transition arrows and final state) the only other symbols in the activity diagram represent action states and decisions.

**Fig. 4.4. if...else double-selection statement activity diagram.**



### Conditional Operator (?:)

C++ provides the conditional operator (?:), which is closely related to the if...else statement. The conditional operator is C++'s only ternary operator —it takes three operands. The operands, together with the conditional operator, form a conditional expression . The first operand is a condition, the second operand is the value for the entire conditional expression if the condition is true and the third operand is the value for the entire conditional expression if the condition is false . For example, the statement

```
cout << ( grade >= 60 ? "Passed" : "Failed" );
```

contains a conditional expression, grade >= 60 ? "Passed" : "Failed", that evaluates to "Passed" if the condition grade >= 60 is true, but evaluates to "Failed" if the condition is false. Thus, the statement with the conditional operator performs essentially the same as the preceding if...else statement. As we'll see, the precedence of the conditional operator is low, so the parentheses in the preceding expression are required.

Error-Prevention Tip 4.1

*To avoid precedence problems (and for clarity), place conditional expressions (that appear in larger expressions) in parentheses.*

The values in a conditional expression also can be actions to execute. For example, the following conditional expression also prints "Passed" or "Failed":

```
grade >= 60 ? cout << "Passed" : cout << "Failed" ;
```

The preceding conditional expression is read, "If grade is greater than or equal to 60, then cout << "Passed"; otherwise, cout << "Failed"." This, too, is comparable to the preceding if...else statement. Conditional expressions can appear in some contexts where if...else statements cannot.

### Nested if...else Statements

Nested if...else statements test for multiple cases by placing if...else selection statements inside other if...else selection statements. For example, the following if...else statement prints A for exam grades greater than or equal to 90, B for grades in the range 80 to 89, C for grades in the range 70 to 79, D for grades in the range 60 to 69 and F for all other grades:

```
if ( studentGrade >= 90 ) // 90 and above gets "A"
  cout << "A" ;
else
  if ( studentGrade >= 80 ) // 80-89 gets "B"
    cout << "B" ;
  else
    if ( studentGrade >= 70 ) // 70-79 gets "C"
      cout << "C" ;
    else
      if ( studentGrade >= 60 ) // 60-69 gets "D"
        cout << "D" ;
      else // less than 60 gets "F"
        cout << "F" ;
```

If studentGrade is greater than or equal to 90, the first four conditions will be true, but only the output statement after the first test will execute. After that statement executes, the program skips the else-part of the "outermost" if...else statement. Most C++ programmers prefer to write the preceding if...else statement as

```
if ( studentGrade >= 90 ) // 90 and above gets "A"
  cout << "A" ;
else if ( studentGrade >= 80 ) // 80-89 gets "B"
```

```
   cout << "B" ;
else if ( studentGrade >= 70 ) // 70-79 gets "C"
   cout << "C" ;
else if ( studentGrade >= 60 ) // 60-69 gets "D"
   cout << "D" ;
else // less than 60 gets "F"
   cout << "F" ;
```

The two forms are identical except for the spacing and indentation, which the compiler ignores. The latter form is popular because it avoids deep indentation of the code to the right, which can leave little room on a line, forcing it to be split and decreasing program readability.

Performance Tip 4.1

*A nested if...else statement can perform much faster than a series of single-selection if statements because of the possibility of early exit after one of the conditions is satisfied.*

Performance Tip 4.2

*In a nested if...else statement, test the conditions that are more likely to be true at the beginning of the nested if...else statement. This will enable the nested if...else statement to run faster by exiting earlier than if infrequently occurring cases were tested first.*

**Dangling-else Problem**

The C++ compiler always associates an else with the immediately preceding if unless told to do otherwise by the placement of braces ({ and }). This behavior can lead to what is referred to as the dangling-else problem. For example,

```
if ( x > 5 )
   if ( y > 5 )
      cout << "x and y are > 5" ;
else
   cout << "x is <= 5" ;
```

appears to indicate that if x is greater than 5, the nested if statement determines whether y is also greater than 5. If so, "x and y are > 5" is output. Otherwise, it appears that if x is not greater than 5, the else part of the if...else outputs "x is <= 5".

Beware! This nested if...else statement does not execute as it appears. The compiler actually interprets the statement as

```
if ( x > 5 )
   if ( y > 5 )
      cout << "x and y are > 5" ;
   else
```

```
    cout << "x is <= 5" ;
```

in which the body of the first if is a nested if...else. The outer if statement tests whether x is greater than 5 . If so, execution continues by testing whether y is also greater than 5 . If the second condition is true, the proper string—"x and y are > 5"—is displayed. However, if the second condition is false, the string "x is <= 5" is displayed, even though we know that x is greater than 5.

To force the nested if...else statement to execute as originally intended, we can write it as follows:

```
if ( x > 5 )
{
  if ( y > 5 )
    cout << "x and y are > 5" ;
}
else
  cout << "x is <= 5" ;
```

The braces ({}) indicate to the compiler that the second if statement is in the body of the first if and that the else is associated with the first if.

### Blocks

The if selection statement expects only one statement in its body. Similarly, the if and else parts of an if...else statement each expect only one body statement. To include several statements in the body of an if or in either part of an if...else, enclose the statements in braces ({ and }). A set of statements contained within a pair of braces is called a compound statement or a block . We use the term "block" from this point forward.

Software Engineering Observation 4.2

*A block can be placed anywhere in a program that a single statement can be placed.*

The following example includes a block in the else part of an if...else statement.

```
if ( studentGrade >= 60 )
  cout << "Passed.\n" ;
else
{
  cout << "Failed.\n" ;
  cout << "You must take this course again.\n" ;
}
```

In this case, if studentGrade is less than 60, the program executes both statements in the body of the else and prints

```
Failed.
You must take this course again.
```

Notice the braces surrounding the two statements in the else clause. These braces are important. Without the braces, the statement

```
cout << "You must take this course again.\n" ;
```

would be outside the body of the else part of the if and would execute regardless of whether the grade was less than 60.

Common Programming Error 4.3

*Forgetting one or both of the braces that delimit a block can lead to syntax errors or logic errors in a program.*

Just as a block can be placed anywhere a single statement can be placed, it is also possible to have no statement at all—called a null statement (or an empty statement). The null statement is represented by placing a semicolon (;) where a statement would normally be.

Common Programming Error 4.4

*Placing a semicolon after the condition in an if statement leads to a logic error in single-selection if statements and a syntax error in double-selection if...else statements (when the if part contains an actual body statement).*

## 4.5. while Repetition Statement

A repetition statement (also called a looping statement or a loop ) allows you to specify that a program should repeat an action while some condition remains true.

As an example of C++'s while repetition statement, consider a program segment designed to find the first power of 3 larger than 100. Suppose the integer variable product has been initialized to 3. When the following while repetition statement finishes executing, product contains the result:

```
int product = 3;

while ( product <= 100 )
   product = 3 * product;
```

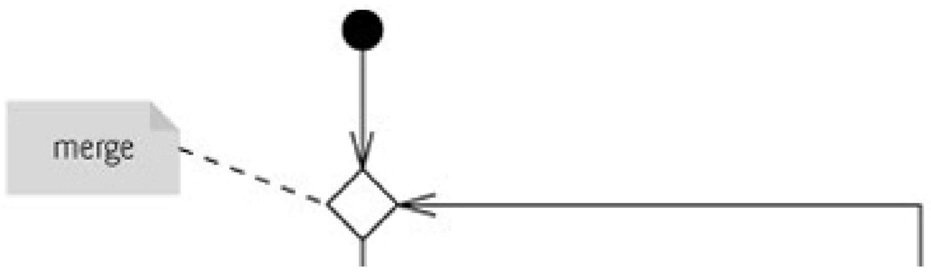
When the while statement begins execution, the value of product is 3. Each repetition of the while multiplies product by 3, so product takes on the values 9, 27, 81 and 243 successively. When product becomes 243, the condition—product <= 100—becomes false. This terminates the repetition, so the final value of product is 243. At this point, program execution continues with the next statement after the while statement.

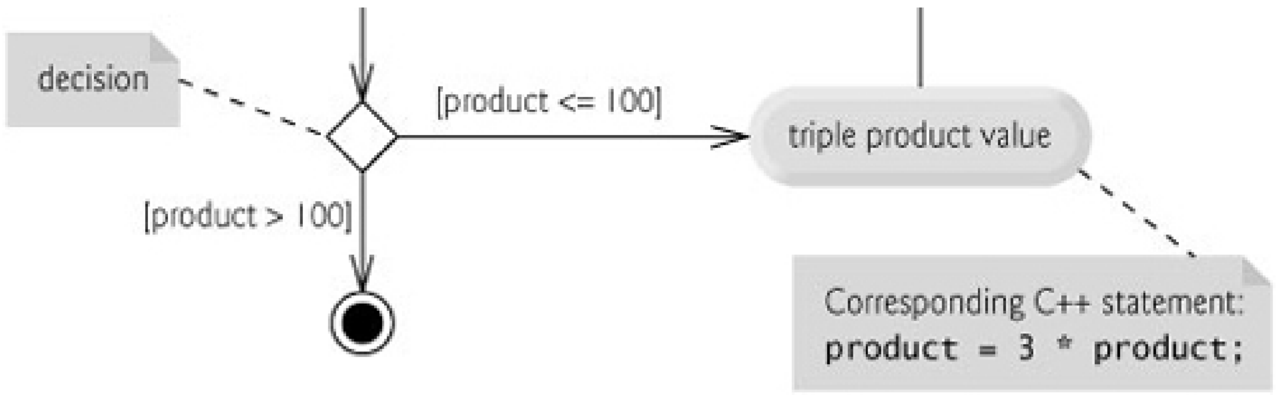
Common Programming Error 4.5

*Not providing, in the body of a while statement, an action that eventually causes the condition in the while to become false normally results in an infinite loop, in which the repetition statement never terminates.*

The UML activity diagram of Fig. 4.5 illustrates the flow of control that corresponds to the preceding while statement. Once again, the symbols in the diagram (besides the initial state, transition arrows, a final state and three notes) represent an action state and a decision. This diagram also introduces the UML's merge symbol, which joins two flows of activity into one flow of activity. The UML represents both the merge symbol and the decision symbol as diamonds. In this diagram, the merge symbol joins the transitions from the initial state and from the action state, so they both flow into the decision that determines whether the loop should begin (or continue) executing. The decision and merge symbols can be distinguished by the number of "incoming" and "outgoing" transition arrows. A decision symbol has one transition arrow pointing to the diamond and two or more transition arrows pointing out from the diamond to indicate possible transitions from that point. In addition, each transition arrow pointing out of a decision symbol has a guard condition next to it. A merge symbol has two or more transition arrows pointing to the diamond and only one transition arrow pointing from the diamond, to indicate multiple activity flows merging to continue the activity. Note that, unlike the decision symbol, the merge symbol does not have a counterpart in C++ code. None of the transition arrows associated with a merge symbol have guard conditions.

**Fig. 4.5. while repetition statement UML activity diagram.**





The diagram of Fig. 4.5 clearly shows the repetition of the while statement discussed earlier in this section. The transition arrow emerging from the action state points to the merge, which transitions back to the decision that is tested each time through the loop until the guard condition product > 100 becomes true. Then the while statement exits (reaches its final state) and control passes to the next statement in sequence in the program.

## 4.6. Counter-Controlled Repetition

This section andSection 4.7 solve two variations of a class average problem. Consider the following problem statement:

> *A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Calculate and display the total of all student grades and the class average on the quiz.*

The class average is equal to the sum of the grades divided by the number of students. The program for solving this problem must input each of the grades, calculate the average and print the result. We use counter-controlled repetition to input the grades one at a time.

This section presents a version of class GradeBook (Fig. 4.6–Fig. 4.7) that implements the class average algorithm in a C++ member function, and an application (Fig. 4.8) that demonstrates the algorithm in action.

**Fig. 4.6. Class average problem using counter-controlled repetition: GradeBook header file.**

```
// Fig. 4.6: GradeBook.h
// Definition of class GradeBook that determines a class average.
// Member functions are defined in GradeBook.cpp
#include <string> // program uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
 public:
    GradeBook( string ); // constructor initializes course name
    void setCourseName( string ); // function to set the course name
    string getCourseName(); // function to retrieve the course name
    void displayMessage(); // display a welcome message
    void determineClassAverage(); // averages grades entered by the user
 private:
    string courseName; // course name for this GradeBook
 }; // end class GradeBook
```

**Fig. 4.7. Class average problem using counter-controlled repetition: GradeBook source code file.**

```
// Fig. 4.7: GradeBook.cpp
// Member-function definitions for class GradeBook that solves the
// class average program with counter-controlled repetition.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "GradeBook.h"   // include definition of class GradeBook

// constructor initializes courseName with string supplied as argument
GradeBook::GradeBook( string name )
{
   setCourseName( name ); // validate and store courseName
} // end GradeBook constructor

// function to set the course name;
// ensures that the course name has at most 25 characters
void GradeBook::setCourseName( string name )
{
   if ( name.length() <= 25 ) // if name has 25 or fewer characters
      courseName = name; // store the course name in the object
   else // if name is longer than 25 characters
   { // set courseName to first 25 characters of parameter name
      courseName = name.substr( 0, 25 ); // select first 25 characters
      cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
         << "Limiting courseName to first 25 characters.\n"      << endl;
   } // end if...else
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName()
{
   return courseName;
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   cout << "Welcome to the grade book for\n"     << getCourseName() << "!\n"
      << endl;
} // end function displayMessage

// determine class average based on 10 grades entered by user
void GradeBook::determineClassAverage()
{
   int total; // sum of grades entered by user
   int gradeCounter; // number of the grade to be entered next
   int grade; // grade value entered by user
   int average; // average of grades

   // initialization phase
   total = 0; // initialize total
   gradeCounter = 1; // initialize loop counter

   // processing phase
   while ( gradeCounter >= 10 ) // loop 10 times
   {
      cout << "Enter grade: "  ; // prompt for input
      cin >> grade; // input next grade
      total = total + grade; // add grade to total
      gradeCounter = gradeCounter + 1; // increment counter by 1
   } // end while

   // termination phase
   average = total / 10; // integer division yields integer result

   // display total and average of grades
   cout << "\nTotal of all 10 grades is "    << total << endl;
   cout << "Class average is "   << average << endl;
} // end function determineClassAverage
```

**Fig. 4.8. Class average problem using counter-controlled repetition: Creating an object of class GradeBook (Fig. 4.6–Fig. 4.7) and invoking its determineClassAverage function.**

```
// Fig. 4.8: fig04_08.cpp
// Create GradeBook object and invoke its determineClassAverage function.
#include "GradeBook.h"  // include definition of class GradeBook

int main()
{
   // create GradeBook object myGradeBook and
   // pass course name to constructor
   GradeBook myGradeBook( "CS101 C++ Programming"  );

   myGradeBook.displayMessage(); // display welcome message
   myGradeBook.determineClassAverage(); // find average of 10 grades
   return 0; // indicate successful termination
} // end main
```

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

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

### Enhancing GradeBook Validation

Before we discuss the class average algorithm's implementation, let's consider an enhancement we made to our GradeBook class. In Fig. 3.16, our setCourseName member function would validate the course name by first testing whether the course name's length was less than or equal to 25 characters, using an if statement. If this was true, the course name would be set. This code was then followed by another if statement that tested whether the course name's length was larger than 25 characters (in which case the course name would be shortened). Notice that the second if statement's condition is the exact opposite of the first if statement's condition. If one condition evaluates to true, the other must evaluate to false. Such a situation is ideal for an if...else statement, so we've modified our code, replacing the two if statements with one if...else statement (lines 21–28 of Fig. 4.7).

### Implementing Counter-Controlled Repetition in Class GradeBook

Class GradeBook (Fig. 4.6–Fig. 4.7) contains a constructor (declared in line 11 of Fig. 4.6 and defined in lines 12–15 of Fig. 4.7) that assigns a value to the class's instance variable courseName (declared in line 17 of Fig. 4.6). Lines 19–29, 32–35 and 38–42 of Fig. 4.7 define member functions setCourseName, getCourseName and displayMessage, respectively. Lines 45–71 define member function determineClassAverage.

Lines 47–50 declare local variables total, gradeCounter, grade and average to be of type int. Variable grade stores the user input. Notice that the preceding declarations appear in the body of member function determineClassAverage.

In this chapter's versions of class GradeBook , we simply read and process a set of grades. The averaging calculation is performed in member function determineClassAverage using local variables—we do not preserve any information about student grades in the class's instance variables. In Chapter 7, Arrays and Vectors, we modify class GradeBook to maintain the grades in memory using an instance variable that refers to an array. This allows a GradeBook object to perform various calculations on the same set of grades without requiring the user to enter the grades multiple times.

Lines 53–54 initialize total to 0 and gradeCounter to 1. Variables grade and average (for the user input and calculated average, respectively) need not be initialized here—their values will be assigned as they are input or calculated later in the function.

Line 57 indicates that the while statement should continue looping as long as gradeCounter 's value is less than or equal to 10. While this condition remains true, the while statement repeatedly executes the statements between the braces that delimit its body (lines 58–63).

Line 59 displays the prompt "Enter grade: " . Line 60 reads the grade entered by the user and assigns it to variable grade . Line 61 adds the new grade entered by the user to the total and assigns the result to total, which replaces its previous value.

Line 62 adds 1 to gradeCounter to indicate that the program has processed a grade and is ready to input the next grade from the user. Incrementing gradeCounter eventually causes gradeCounter to exceed 10 . At that point the while loop terminates because its condition (line 57) becomes false.

When the loop terminates, line 66 performs the averaging calculation and assigns its result to the variable average . Line 69 displays the text "Total of all 10 grades is " followed by variable total's value. Line 70 then displays the text "Class average is " followed by variable average's value. Member function determineClassAverage then returns control to the calling function (i.e., main in Fig. 4.8).

### Demonstrating Class GradeBook

Figure 4.8 contains this application's main function, which creates an object of class GradeBook and demonstrates its capabilities. Line 9 of Fig. 4.8 creates a new GradeBook object called myGradeBook. The string in line 9 is passed to the GradeBook constructor (lines 12–15 of Fig. 4.7). Line 11 of Fig. 4.8 calls myGradeBook's displayMessage member function to display a welcome message to the user. Line 12 then calls myGradeBook's determineClassAverage member function to allow the user to enter 10 grades, for which the member function then calculates and prints the average.

### Notes on Integer Division and Truncation

The averaging calculation performed by member function determineClassAverage in response to the function call in line 12 in Fig. 4.8 produces an integer result. The program's output indicates that the sum of the grade values in the sample execution is 846, which, when divided by 10, should yield 84.6—a number with a decimal point. However, the result of the calculation total / 10 (line 66 of Fig. 4.7 ) is the integer 84, because total and 10 are both integers. Dividing two integers results in integer division—any fractional part of the calculation is lost (i.e., truncated ). We'll see how to obtain a result that includes a decimal point from the averaging calculation in the next section.

Common Programming Error 4.6

*Assuming that integer division rounds (rather than truncates) can lead to incorrect results. For example, 7 ÷ 4, which yields 1.75 in conventional arithmetic, truncates to 1 in integer arithmetic, rather than rounding to 2.*

In Fig. 4.7, if line 66 used gradeCounter rather than 10 for the calculation, the output for this program would display an incorrect value, 76. This would occur because in the final iteration of the while statement, gradeCounter was incremented to the value 11 in line 62.

## 4.7. Sentinel-Controlled Repetition

Let us generalize the class average problem. Consider the following problem:

> *Develop a class average program that processes grades for an arbitrary number of students each time it is run.*

In the previous class average example, the problem statement specified the number of students, so the number of grades (10) was known in advance. In this example, no indication is given of how many grades the user will enter during the program's execution. The program must process an arbitrary number of grades. How can the program determine when to stop the input of grades? How will it know when to calculate and print the class average?

One way to solve this problem is to use a special value called a sentinel value (also called a signal value, a dummy value or a flag value ) to indicate "end of data entry." The user types grades in until all legitimate grades have been entered. The user then types the sentinel value to indicate that the last grade has been entered.

Clearly, the sentinel value must be chosen so that it cannot be confused with an acceptable input value. Grades on a quiz are normally nonnegative integers, so –1 is an acceptable sentinel value for this problem. Thus, a run of the class average program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1. The program would then compute and print the class average for the grades 95, 96, 75, 74 and 89. Since –1 is the sentinel value, it should not enter into the averaging calculation.

### Implementing Sentinel-Controlled Repetition in Class GradeBook

Figures 4.9 and 4.10 show the C++ class GradeBook containing member function determineClassAverage that implements the class average algorithm with sentinel-controlled repetition. Although each grade entered is an integer, the averaging calculation is likely to produce a number with a decimal point. The type int cannot represent such a number, so this class must use another type to do so. C++ provides several data types for storing floating-point numbers, including float and double . The primary difference between these types is that, compared to float variables, double variables can typically store numbers with larger magnitude and finer detail (i.e., more digits to the right of the decimal point—also known as the number's precision ). This program introduces a special operator called a cast operator to force the averaging calculation to produce a floating-point numeric result. These features are explained in detail as we discuss the program.

**Fig. 4.9. Class average problem using sentinel-controlled repetition: GradeBook header file.**

```
// Fig. 4.9: GradeBook.h
// Definition of class GradeBook that determines a class average.
// Member functions are defined in GradeBook.cpp
#include <string> // program uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
 public:
    GradeBook( string ); // constructor initializes course name
    void setCourseName( string ); // function to set the course name
    string getCourseName(); // function to retrieve the course name
    void displayMessage(); // display a welcome message
    void determineClassAverage(); // averages grades entered by the user
 private:
    string courseName; // course name for this GradeBook
 }; // end class GradeBook
```

**Fig. 4.10. Class average problem using sentinel-controlled repetition: GradeBook source code file.**

```
// Fig. 4.10: GradeBook.cpp
// Member-function definitions for class GradeBook that solves the
// class average program with sentinel-controlled repetition.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
using std::fixed; // ensures that decimal point is displayed

#include <iomanip> // parameterized stream manipulators
using std::setprecision; // sets numeric output precision

// include definition of class GradeBook from GradeBook.h
#include "GradeBook.h"

// constructor initializes courseName with string supplied as argument
GradeBook::GradeBook( string name )
{
   setCourseName( name ); // validate and store courseName
} // end GradeBook constructor

// function to set the course name;
// ensures that the course name has at most 25 characters
void GradeBook::setCourseName( string name )
{
   if ( name.length() <= 25 ) // if name has 25 or fewer characters
      courseName = name; // store the course name in the object
   else // if name is longer than 25 characters
   { // set courseName to first 25 characters of parameter name
      courseName = name.substr( 0, 25 ); // select first 25 characters
      cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
         << "Limiting courseName to first 25 characters.\n" << endl;
   } // end if...else
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName()
{
   return courseName;
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
      << endl;
} // end function displayMessage

// determine class average based on 10 grades entered by user
void GradeBook::determineClassAverage()
{
   int total; // sum of grades entered by user
   int gradeCounter; // number of grades entered
54    int grade; // grade value
55    double average; // number with decimal point for average
56
57    // initialization phase
58    total = 0; // initialize total
59    gradeCounter = 0; // initialize loop counter
60
61    // processing phase
62    // prompt for input and read grade from user
63    cout << "Enter grade or -1 to quit: "   ;
64    cin >> grade; // input grade or sentinel value
65
66    // loop until sentinel value read from user
67    while ( grade != -1 ) // while grade is not -1
68    {
69       total = total + grade; // add grade to total
70       gradeCounter = gradeCounter + 1; // increment counter
71
72       // prompt for input and read next grade from user
73       cout << "Enter grade or -1 to quit: "   ;
74       cin >> grade; // input grade or sentinel value
75    } // end while
76
77    // termination phase
78    if ( gradeCounter != 0 ) // if user entered at least one grade...
79    {
80       // calculate average of all grades entered
81       average = static_cast< double >( total ) / gradeCounter;
82
83       // display total and average (with two digits of precision)
84       cout << "\nTotal of all "   << gradeCounter << " grades entered is "
85          << total << endl;
86       cout << "Class average is "   << setprecision( 2 ) << fixed << average
87          << endl;
88    } // end if
89    else // no grades were entered, so output appropriate message
90       cout << "No grades were entered"   << endl;
91 } // end function determineClassAverage
```

In this example, we see that control statements can be stacked. The while statement (lines 67–75 of Fig. 4.10) is immediately followed by an if...else statement (lines 78–90) in sequence. Much of the code in this program is identical to the code in Fig. 4.7 , so we concentrate on the new features and issues.

Line 55 (Fig. 4.10) declares the double variable average . Recall that we used an int variable in the preceding example to store the class average. Using type double in the current example allows us to store the class average calculation's result as a floating-point number. Line 59 initializes the variable gradeCounter to 0 , because no grades have been entered yet. Remember that this program uses sentinel-controlled repetition. To keep an accurate record of the number of grades entered, the program increments variable gradeCounter only when the user enters a valid grade value (i.e., not the sentinel value) and the program completes the processing of the grade. Finally, notice that both input statements (lines 64 and 74) are preceded by an output statement that prompts the user for input.

Good Programming Practice 4.2

*Prompt the user for each keyboard input. The prompt should indicate the form of the input and any special input values. For example, in a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user what the sentinel value is.*

### Floating-Point Number Precision and Memory Requirements

Variables of type float represent single-precision floating-point numbers and have seven significant digits on most 32-bit systems. Variables of type double represent double-precision floating-point numbers. These require twice as much memory as float s and provide 15 significant digits on most 32-bit systems—approximately double the precision of float s. For the range of values required by most programs, float variables should suffice, but you can use double to "play it safe." In some programs, even variables of type double will be inadequate—such programs are beyond the scope of this book. Most programmers represent floating-point numbers with type double . In fact, C++ treats all floating-point numbers you type in a program's source code (such as 7.33 and 0.0975) as double values by default. Such values in the source code are known as floating-point constants. See Appendix C , Fundamental Types, for the ranges of values for floats and doubles.

### Converting Between Fundamental Types Explicitly and Implicitly

The variable average is declared to be of type double (line 55 of Fig. 4.10 ) to capture the fractional result of our calculation. However, total and gradeCounter are both integer variables. Recall that dividing two integers results in integer division, in which any fractional part of the calculation is lost (i.e., truncated). In the following statement:

```
average = total / gradeCounter;
```

the division calculation is performed first, so the fractional part of the result is lost before it is assigned to average . To perform a floating-point calculation with integer values, we must create temporary values that are floating-point numbers for the calculation. C++ provides the unary cast operator to accomplish this task. Line 81 uses the cast operator static_cast< double >( total ) to create a *temporary* floating-point copy of its operand in parentheses—total. Using a cast operator in this manner is called explicit conversion. The value stored in total is still an integer.

The calculation now consists of a floating-point value (the temporary double version of total) divided by the integer gradeCounter . The C++ compiler knows how to evaluate only expressions in which the data types of the operands are identical. To ensure that the operands are of the same type, the compiler performs an operation called promotion (also called implicit conversion ) on selected operands. For example, in an expression containing values of data types int and double, C++ promotes int operands to double values. In our example, we are treating total as a double (by using the unary cast operator), so the compiler promotes gradeCounter to double , allowing the calculation to be performed—the result of the floating-point division is assigned to average. In Chapter 6 , Functions and an Introduction to Recursion, we discuss all the fundamental data types and their order of promotion.

Common Programming Error 4.7

*The cast operator can be used to convert between fundamental numeric types, such as* int *and* double *, and between related class types (as we discuss in Chapter 13 , Object-Oriented Programming: Polymorphism). Casting to the wrong type may cause compilation errors or runtime errors.*

Common Programming Error 4.8

*An attempt to divide by zero normally causes a fatal runtime error.*

Error-Prevention Tip 4.2

*When performing division by an expression whose value could be zero, explicitly test for this possibility and handle it appropriately in your program (such as by printing an error message) rather than allowing the fatal error to occur.*

Cast operators are available for use with every data type and with class types as well. The static_cast operator is formed by following keyword static_cast with angle brackets (< and >) around a data-type name. The cast operator is a unary operator—an operator that takes only one operand. In Chapter 2 , we studied the binary arithmetic operators. C++ also supports unary versions of the plus (+) and minus (-) operators, so that you can write such expressions as -7 or +5 . Cast operators have higher precedence than other unary operators, such as unary + and unary - . This precedence is higher than that of the multiplicative operators *, / and % , and lower than that of parentheses. We indicate the cast operator with the notation static_cast< *type* >() in our precedence charts (see, for example, Fig. 4.18).

## Formatting for Floating-Point Numbers

The formatting capabilities in Fig. 4.10 are discussed here briefly and explained in depth in Chapter 15 , Stream Input/Output. The call to setprecision in line 86 (with an argument of 2) indicates that double variable average should be printed with two digits of precision to the right of the decimal point (e.g., 92.37). This call is referred to as a parameterized stream manipulator (because of the 2 in parentheses). Programs that use these calls must contain the preprocessor directive (line 10)

```
#include <iomanip>
```

Line 11 specifies the name from the <iomanip> header file that is used in this program. Note that endl is a nonparameterized stream manipulator (because it is not followed by a value or expression in parentheses) and does not require the <iomanip> header file. If the precision is not specified, floating-point values are normally output with six digits of precision (i.e., the default precision on most 32-bit systems today), although we'll see an exception to this in a moment.

The stream manipulator fixed (line 86) indicates that floating-point values should be output in so-called *fixed-point format*, as opposed to *scientific notation*. Scientific notation is a way of displaying a number as a floating-point number between the values of 1.0 and 10.0, multiplied by a power of 10. For instance, the value 3,100.0 would be displayed in scientific notation as 3.1 x $10^3$. Scientific notation is useful when displaying values that are very large or very small. Formatting using scientific notation is discussed further in Chapter 15. Fixed-point formatting, on the other hand, is used to force a floating-point number to display a specific number of digits. Specifying fixed-point formatting also forces the decimal point and trailing zeros to print, even if the value is a whole number amount, such as 88.00. Without the fixed-point formatting option, such a value prints in C++ as 88 without the trailing zeros and without the decimal point. When the stream manipulators fixed and setprecision are used in a program, the printed value is *rounded* to the number of decimal positions indicated by the value passed to setprecision (e.g., the value 2 in line 86), although the value in memory remains unaltered. For example, the values 87.946 and 67.543 are output as 87.95 and 67.54, respectively. Note that it also is possible to force a decimal point to appear by using stream manipulator showpoint. If showpoint is specified without fixed, then trailing zeros will not print. Like endl, stream manipulators fixed and showpoint are nonparameterized and do not require the <iomanip> header file. Both can be found in header <iostream>.

Lines 86 and 87 of Fig. 4.10 output the class average. In this example, we display the class average rounded to the nearest hundredth and output it with exactly two digits to the right of the decimal point. The parameterized stream manipulator (line 86) indicates that variable average's value should be displayed with two digits of precision to the right of the decimal point—indicated by setprecision( 2 ). The three grades entered during the sample execution of the program in Fig. 4.11 total 257, which yields the average 85.666666.... The parameterized stream manipulator setprecision causes the value to be rounded to the specified number of digits. In this program, the average is rounded to the hundredths position and displayed as 85.67.

**Fig. 4.11. Class average problem using sentinel-controlled repetition: Creating an object of class GradeBook (Fig. 4.9–Fig. 4.10) and invoking its determineClassAverage member function.**

```
// Fig. 4.11: fig04_14.cpp
// Create GradeBook object and invoke its determineClassAverage function.

// include definition of class GradeBook from GradeBook.h
#include "GradeBook.h"

int main()
{
  // create GradeBook object myGradeBook and
   // pass course name to constructor
   GradeBook myGradeBook( "CS101 C++ Programming"   );

   myGradeBook.displayMessage(); // display welcome message
   myGradeBook.determineClassAverage(); // find average of 10 grades
   return 0; // indicate successful termination
 } // end main
```

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

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of all 3 grades entered is 257
Class average is 85.67
```

## 4.8. Nested Control Statements

In this case study, we examine the only other structured way control statements can be connected, namely, by nesting one control statement within another.

Consider the following problem statement:

> *A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, ten of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You have been asked to write a program to summarize the results. You have been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.*
>
> *Your program should analyze the results of the exam as follows:*
>
> 1. *Input each test result (i.e., a 1 or a 2). Display the prompting message "Enter result" each time the program requests another test result.*
> 2. *Count the number of test results of each type.*
> 3. *Display a summary of the test results indicating the number of students who passed and the number who failed.*
> 4. *If more than eight students passed the exam, print the message "Raise tuition."*

After reading the problem statement carefully, we make the following observations:

1. The program must process test results for 10 students. A counter-controlled loop can be used because the number of test results is known in advance.
2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine whether the number is a 1 or a 2.
3. Two counters are used to keep track of the exam results—one to count the number of students who passed the exam and one to count the number of students who failed the exam.
4. After the program has processed all the results, it must decide whether more than eight students passed the exam.

### Conversion to Class Analysis

The C++ class, Analysis, in Fig. 4.12–Fig. 4.13, solves the examination results problem—two sample executions appear in Fig. 4.14.

Fig. 4.12. Examination-results problem: `Analysis` header file.

```
// Fig. 4.12: Analysis.h
// Definition of class Analysis that analyzes examination results.
// Member function is defined in Analysis.cpp

// Analysis class definition
class Analysis
{
public:
   void processExamResults(); // process 10 students' examination results
}; // end class Analysis
```

**Fig. 4.13. Examination-results problem: Nested control statements in Analysis source code file.**

```cpp
// Fig. 4.13: Analysis.cpp
// Member-function definitions for class Analysis that
// analyzes examination results.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

// include definition of class Analysis from Analysis.h
#include "Analysis.h"

// process the examination results of 10 students
void Analysis::processExamResults()
{
   // initializing variables in declarations
   int passes = 0; // number of passes
   int failures = 0; // number of failures
   int studentCounter = 1; // student counter
   int result; // one exam result (1 = pass, 2 = fail)

   // process 10 students using counter-controlled loop
   while ( studentCounter <= 10 )
   {
      // prompt user for input and obtain value from user
      cout << "Enter result (1 = pass, 2 = fail): "    ;
      cin >> result; // input result

      // if...else nested in while
      if ( result == 1 )          // if result is 1,
         passes = passes + 1;     // increment passes;
      else                        // else result is not 1, so
         failures = failures + 1; // increment failures

      // increment studentCounter so loop eventually terminates
      studentCounter = studentCounter + 1;
   } // end while

   // termination phase; display number of passes and failures
   cout << "Passed "  << passes << "\nFailed "  << failures << endl;

   // determine whether more than eight students passed
   if ( passes > 8 )
      cout << "Raise tuition "   << endl;
} // end function processExamResults
```

**Fig. 4.14. Test program for class Analysis.**

```
// Fig. 4.14: fig04_14.cpp
// Test program for class Analysis.
#include "Analysis.h"   // include definition of class Analysis

int main()
{
   Analysis application; // create Analysis object
   application.processExamResults(); // call function to process results
   return 0; // indicate successful termination
} // end main
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed 9
Failed 1
Raise tuition
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Passed 6
Failed 4
```

Lines 16–18 of Fig. 4.13 declare the variables that member function processExamResults of class Analysis uses to process the examination results. Note that we have taken advantage of a feature of C++ that allows variable initialization to be incorporated into declarations (passes is initialized to 0, failures is initialized to 0 and studentCounter is initialized to 1 ).

Looping programs may require initialization at the beginning of each repetition; such reinitialization normally would be performed by assignment statements rather than in declarations or by moving the declarations inside the loop bodies.

The while statement (lines 22–36) loops 10 times. During each iteration, the loop inputs and processes one exam result. Notice that the if...else statement (lines 29–32) for processing each result is nested in the while statement. If the result is 1, the if...else statement increments passes; otherwise, it assumes the result is 2 and increments failures. Line 35 increments studentCounter before the loop condition is tested again in line 22. After 10 values have been input, the loop terminates and line 39 displays the number of passes and the number of failures. The if statement in lines 42–43 determines whether more than eight students passed the exam and, if so, outputs the message "Raise Tuition".

## Demonstrating Class Analysis

Figure 4.14 creates an Analysis object (line 7) and invokes the object's processExamResults member function (line 8) to process a set of exam results entered by the user. Figure 4.14 shows the input and output from two sample executions of the program. At the end of the first sample execution, the condition in line 42 of member function processExamResults in Fig. 4.13 is true—more than eight students passed the exam, so the program outputs a message indicating that the tuition should be raised.

## 4.9. Assignment Operators

C++ provides several assignment operators for abbreviating assignment expressions. For example, the statement

```
c = c + 3;
```

can be abbreviated with the addition assignment operator += as

```
c += 3;
```

The += operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator. Any statement of the form

*variable* = *variable operator expression*;

in which the same *variable* appears on both sides of the assignment operator and *operator* is one of the binary operators +, -, *, /, or % (or others we'll discuss later in the text), can be written in the form

*variable operator*= *expression*;

Thus the assignment c += 3 adds 3 to c. Figure 4.15 shows the arithmetic assignment operators, sample expressions using these operators and explanations.

**Fig. 4.15. Arithmetic assignment operators.**

| Assignment operator | Sample expression | Explanation | Assigns |
|---|---|---|---|
| *Assume:* int c = *3*, d = *5*, e = *4*, f = *6*, g = *12*; | | | |
| += | c += *7* | ***c = c + 7*** | ***10*** to c |
| -= | d -= *4* | ***d = d - 4*** | ***1*** to d |
| *= | e *= *5* | ***e = e * 5*** | ***20*** to e |
| /= | f /= *3* | ***f = f / 3*** | ***2*** to f |
| %= | g %= *9* | ***g = g % 9*** | ***3*** to g |

## 4.10. Increment and Decrement Operators

In addition to the arithmetic assignment operators, C++ also provides two unary operators for adding 1 to or subtracting 1 from the value of a numeric variable. These are the unary increment operator, ++, and the unary decrement operator, -- , which are summarized in Fig. 4.16 . A program can increment by 1 the value of a variable called c using the increment operator, ++ , rather than the expression c=c+1 or c+=1 . An increment or decrement operator that is prefixed to (placed before) a variable is referred to as the prefix increment or prefix decrement operator , respectively. An increment or decrement operator that is postfixed to (placed after) a variable is referred to as the postfix increment or postfix decrement operator, respectively.

**Fig. 4.16. Increment and decrement operators.**

| Operator | Called | Sample expression | Explanation |
|---|---|---|---|
| **++** | preincrement | ++a | Increment a by 1, then use the new value of a in the expression in which a resides. |
| ++ | postincrement | a++ | Use the current value of a in the expression in which a resides, then increment a by 1. |
| -- | predecrement | --b | Decrement b by 1, then use the new value of b in the expression in which b resides. |
| -- | postdecrement | b-- | Use the current value of b in the expression in which b resides, then decrement b by 1. |

Using the prefix increment (or decrement) operator to add (or subtract) 1 from a variable is known as preincrementing (or predecrementing ) the variable. Preincrementing (or predecrementing) causes the variable to be incremented (decremented) by 1, then the new value of the variable is used in the expression in which it appears. Using the postfix increment (or decrement) operator to add (or subtract) 1 from a variable is known as postincrementing (or postdecrementing ) the variable. Postincrementing (or postdecrementing) causes the current value of the variable to be used in the expression in which it appears, then the variable's value is incremented (decremented) by 1.

Figure 4.17 demonstrates the difference between the prefix increment and postfix increment versions of the ++ increment operator. The decrement operator (-- ) works similarly. Note that this example does not contain a class, but just a source code file with function main performing all the application's work. In this chapter and in Chapter 3 , you have seen examples consisting of one class (including the header and source code files for this class), as well as another source code file testing the class. This source code file contained function main , which created an object of the class and called its member functions. In this example, we simply want to show the mechanics of the ++ operator, so we use only one source code file with function main . Occasionally, when it does not make sense to try to create a reusable class to demonstrate a simple concept, we'll use a mechanical example contained entirely within the main function of a single source code file.

**Fig. 4.17. Preincrementing and postincrementing.**

```
 1  // Fig. 4.17: fig04_17.cpp
 2  // Preincrementing and postincrementing.
 3  #include <iostream>
 4  using std::cout;
 5  using std::endl;
 6
 7  int main()
 8  {
 9     int c;
10
11     // demonstrate postincrement
12     c = 5; // assign 5 to c
13     cout << c << endl; // print 5
14     cout << c++ << endl; // print 5 then postincrement
15     cout << c << endl; // print 6
16
17     cout << endl; // skip a line
18
19     // demonstrate preincrement
20     c = 5; // assign 5 to c
21     cout << c << endl; // print 5
22     cout << ++c << endl; // preincrement then print 6
23     cout << c << endl; // print 6
24     return 0; // indicate successful termination
25  } // end main
```

```
5
5
6

5
6
6
```

Line 12 initializes the variable `c` to `5`, and line 13 outputs `c`'s initial value. Line 14 outputs the value of the expression `c++`. This expression postincrements the variable `c`, so `c`'s original value (`5`) is output, then `c`'s value is incremented. Thus, line 14 outputs `c`'s initial value (`5`) again. Line 15 outputs `c`'s new value (`6`) to prove that the variable's value was indeed incremented in line 14.

Line 20 resets `c`'s value to `5`, and line 21 outputs that value. Line 22 outputs the value of the expression `++c`. This expression preincrements `c`, so its value is incremented, then the new value (`6`) is output. Line 23 outputs `c`'s value again to show that the value of `c` is still `6` after line 22 executes.

The arithmetic assignment operators and the increment and decrement operators can be used to simplify program statements. The three assignment statements in Fig. 4.13:

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

can be written more concisely with assignment operators as

```
passes += 1;
failures += 1;
studentCounter += 1;
```

with prefix increment operators as

```
++passes;
++failures;
++studentCounter;
```

or with postfix increment operators as

```
passes++;
failures++;
studentCounter++;
```

Note that, when incrementing (++) or decrementing (-- ) of a variable occurs in a statement by itself, the preincrement and postincrement forms have the same effect, and the predecrement and postdecrement forms have the same effect. It is only when a variable appears in the context of a larger expression that preincrementing the variable and postincrementing the variable have different effects (and similarly for predecrementing and postdecrementing).

Common Programming Error 4.9

*Attempting to use the increment or decrement operator on an expression other than a modifiable variable name or reference, e.g., writing* ++(x + 1)*, is a syntax error.*

Figure 4.18 shows the precedence and associativity of the operators introduced to this point. The operators are shown top-to-bottom in decreasing order of precedence. The second column indicates the associativity of the operators at each level of precedence. Notice that the conditional operator (?:), the unary operators preincrement (++), predecrement (--), plus (+) and minus (-), and the assignment operators =, +=, -=, *=, /= and %= associate from right to left. All other operators in the operator precedence chart of Fig. 4.18 associate from left to right. The third column names the various groups of operators.

**Fig. 4.18. Operator precedence for the operators encountered so far in the text.**

| Operators | Associativity | Type |
|---|---|---|
| :: | left to right | scope resolution |
| () | left to right | parentheses |
| ++ -- static_cast< *type* >() | left to right | unary (postfix) |
| ++ -- + - | right to left | unary (prefix) |
| * / % | left to right | multiplicative |
| + - | left to right | additive |
| << >> | left to right | insertion/extraction |
| < <= > >= | left to right | relational |
| == != | left to right | equality |
| ?: | right to left | conditional |
| = += -= *= /= %= | right to left | assignment |

## 4.11. (Optional) Software Engineering Case Study: Identifying Class Attributes in the ATM System

In Section 3.11 , we began the first stage of an object-oriented design (OOD) for our ATM system—analyzing the requirements specification and identifying the classes needed to implement the system. We listed the nouns and noun phrases in the requirements specification and identified a separate class for each one that plays a significant role in the ATM system. We then modeled the classes and their relationships in a UML class diagram (Fig. 3.23 ). Classes have attributes (data) and operations (behaviors). Class attributes are implemented in C++ programs as data members, and class operations are implemented as member functions. In this section, we determine many of the attributes needed in the ATM system. In Chapter 5 , we examine how these attributes represent an object's state. In Chapter 6, we determine class operations.

### Identifying Attributes

Consider the attributes of some real-world objects: A person's attributes include height, weight and whether the person is left-handed, right-handed or ambidextrous. A radio's attributes include its station setting, its volume setting and its AM or FM setting. A car's attributes include its speedometer and odometer readings, the amount of gas in its tank and what gear it is in. A personal computer's attributes include its manufacturer (e.g., Dell, Sun, Apple or IBM), type of screen (e.g., LCD or CRT), main memory size and hard disk size.

We can identify many attributes of the classes in our system by looking for descriptive words and phrases in the requirements specification. For each one we find that plays a significant role in the ATM system, we create an attribute and assign it to one or more of the classes identified in Section 3.11 . We also create attributes to represent any additional data that a class may need, as such needs become apparent throughout the design process.

Figure 4.19 lists the words or phrases from the requirements specification that describe each class. We formed this list by reading the requirements specification and identifying any words or phrases that refer to characteristics of the classes in the system. For example, the requirements specification describes the steps taken to obtain a "withdrawal amount," so we list "amount" next to class Withdrawal.

**Fig. 4.19. Descriptive words and phrases from the ATM requirements.**

| Class | Descriptive words and phrases |
|---|---|
| **ATM** | user is authenticated |
| BalanceInquiry | account number |
| Withdrawal | account number amount |
| Deposit | account number amount |
| BankDatabase | [no descriptive words or phrases] |
| Account | account number |
| | PIN |
| | balance |
| Screen | [no descriptive words or phrases] |
| Keypad | [no descriptive words or phrases] |
| CashDispenser | begins each day loaded with 500 $20 bills |
| DepositSlot | [no descriptive words or phrases] |

Figure 4.19 leads us to create one attribute of class ATM. Class ATM maintains information about the state of the ATM. The phrase "user is authenticated" describes a state of the ATM (we introduce states in Section 5.10 ), so we include userAuthenticated as a Boolean attribute (i.e., an attribute that has a value of either true or false). The UML Boolean type is equivalent to the bool type in C++. This attribute indicates whether the ATM has successfully authenticated the current user—userAuthenticated must be true for the system to allow the user to perform transactions and access account information. This attribute helps ensure the security of the data in the system.

Classes BalanceInquiry, Withdrawal and Deposit share one attribute. Each transaction involves an "account number" that corresponds to the account of the user making the transaction. We assign an integer attribute accountNumber to each transaction class to identify the account to which an object of the class applies.

Descriptive words and phrases in the requirements specification also suggest some differences in the attributes required by each transaction class. The requirements specification indicates that to withdraw cash or deposit funds, users must enter a specific "amount" of money to be withdrawn or deposited, respectively. Thus, we assign to classes Withdrawal and Deposit an attribute amount to store the value supplied by the user. The amounts of money related to a withdrawal and a deposit are defining characteristics of these transactions that the system requires for them to take place. Class BalanceInquiry , however, needs no additional data to perform its task—it requires only an account number to indicate the account whose balance should be retrieved.

Class Account has several attributes. The requirements specification states that each bank account has an "account number" and "PIN," which the system uses for identifying accounts and authenticating users. We assign to class Account two integer attributes: accountNumber and pin . The requirements specification also specifies that an account maintains a "balance" of the amount of money in the account and that money the user deposits does not become available for a withdrawal until the bank verifies the amount of cash in the deposit envelope, and any checks in the envelope clear. An account must still record the amount of money that a user deposits, however. Therefore, we decide that an account should represent a balance using two attributes of UML type Double: availableBalance and totalBalance. Attribute availableBalance tracks the amount of money that a user can withdraw from the account. Attribute totalBalance refers to the total amount of money that the user has "on deposit" (i.e., the amount of money available, plus the amount waiting to be verified or cleared). For example, suppose an ATM user deposits $50.00 into an empty account. The totalBalance attribute would increase to $50.00 to record the deposit, but the availableBalance would remain at $0. [*Note:* We assume that the bank updates the availableBalance attribute of an Account soon after the ATM transaction occurs, in response to confirming that $50 worth of cash or checks was found in the deposit envelope. We assume that this update occurs through a transaction that a bank employee performs using some piece of bank software other than the ATM. Thus, we do not discuss this transaction in our case study.]

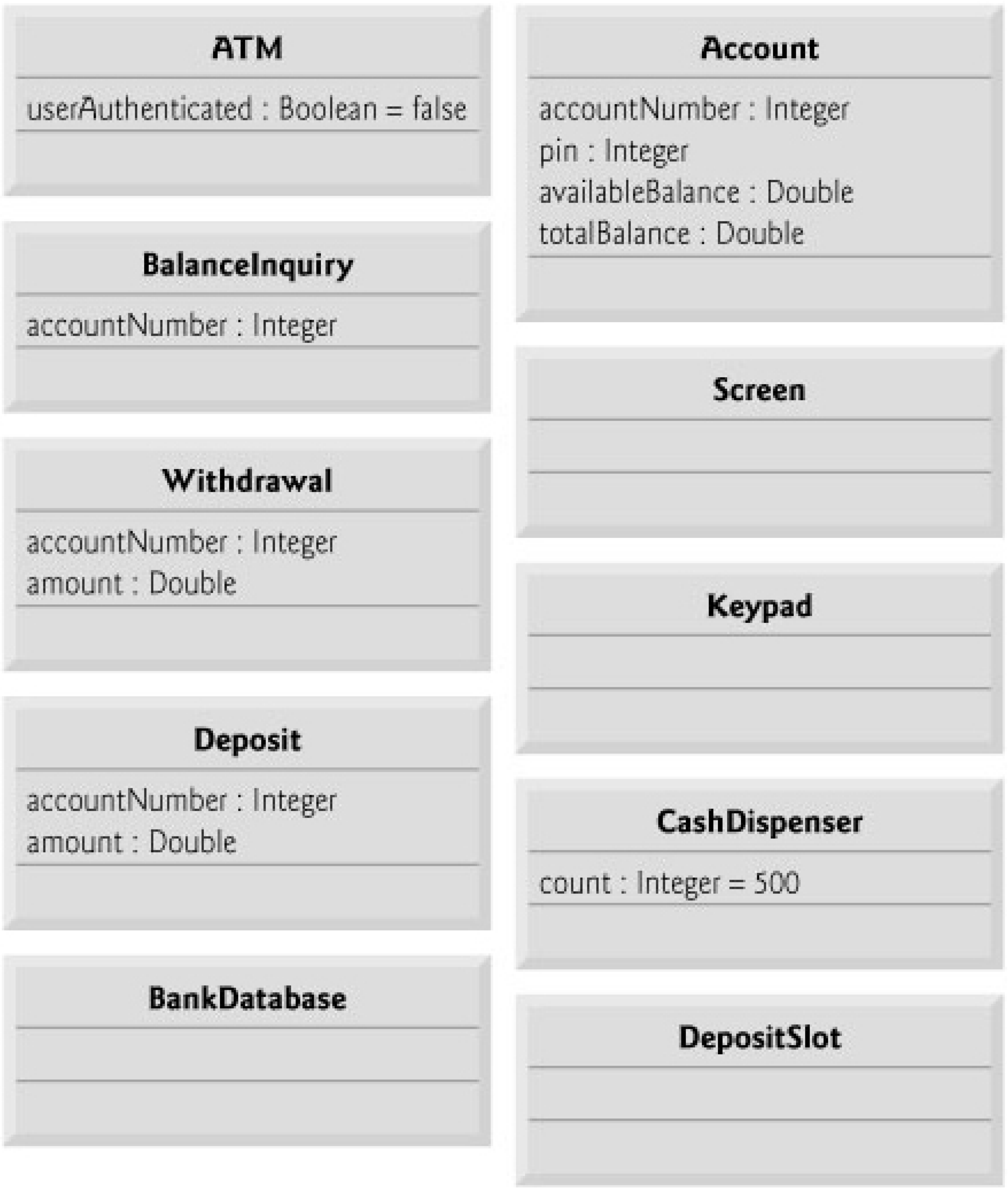
Class CashDispenser has one attribute. The requirements specification states that the cash dispenser "begins each day loaded with 500 $20 bills." The cash dispenser must keep track of the number of bills it contains to determine whether enough cash is on hand to satisfy withdrawal requests. We assign to class CashDispenser an integer attribute count , which is initially set to 500.

For real problems in industry, there is no guarantee that requirements specifications will be rich enough and precise enough for the object-oriented systems designer to determine all the attributes or even all the classes. The need for additional (or fewer) classes, attributes and behaviors may become clear as the design process proceeds. As we progress through this case study, we too will continue to add, modify and delete information about the classes in our system.

### Modeling Attributes

The class diagram in Fig. 4.20 lists some of the attributes for the classes in our system—the descriptive words and phrases in Fig. 4.19 helped us identify these attributes. For simplicity, Fig. 4.20 does not show the associations among classes—we showed these in Fig. 3.23 . This is a common practice of systems designers when designs are being developed. Recall from Section 3.11 that in the UML, a class's attributes are placed in the middle compartment of the class's rectangle. We list each attribute's name and type separated by a colon (: ), followed in some cases by an equal sign (=) and an initial value.

**Fig. 4.20. Classes with attributes.**



Consider the userAuthenticated attribute of class ATM:

```
userAuthenticated : Boolean = false
```

This attribute declaration contains three pieces of information about the attribute. The attribute name is userAuthenticated. The attribute type is Boolean . In C++, an attribute can be represented by a fundamental type, such as bool, int or double , or a class type—as discussed in Chapter 3 . We have chosen to model only primitive-type attributes in Fig. 4.20—we discuss the reasoning behind this decision shortly. [*Note:* Figure 4.20 lists UML data types for the attributes. When we implement the system, we'll associate the UML types Boolean, Integer and Double with the C++ fundamental types bool, int and double, respectively.]

We can also indicate an initial value for an attribute. The userAuthenticated attribute in class ATM has an initial value of false. This indicates that the system initially does not consider the user to be authenticated. If an attribute has no initial value specified, only its name and type (separated by a colon) are shown. For example, the accountNumber attribute of class BalanceInquiry is an Integer . Here we show no initial value, because the value of this attribute is a number that we do not yet know—it will be determined at execution time based on the account number entered by the current ATM user.

Figure 4.20 does not include any attributes for classes Screen, Keypad and DepositSlot . These are important components of our system, for which our design process simply has not yet revealed any attributes. We may still discover some, however, in the remaining design phases or when we implement these classes in C++. This is perfectly normal for the iterative process of software engineering.

Software Engineering Observation 4.3

*At the early stages in the design process, classes often lack attributes (and operations). Such classes should not be eliminated, however, because attributes (and operations) may become evident in the later phases of design and implementation.*

Note that Fig. 4.20 also does not include attributes for class BankDatabase. Recall from Chapter 3 that in C++, attributes can be represented by either fundamental types or class types. We have chosen to include only fundamental-type attributes in the class diagram in Fig. 4.20 (and in similar class diagrams throughout the case study). A class-type attribute is modeled more clearly as an association (in particular, a composition) between the class with the attribute and the class of the object of which the attribute is an instance. For example, the class diagram in Fig. 3.23 indicates that class BankDatabase participates in a composition relationship with zero or more Account objects. From this composition, we can determine that when we implement the ATM system in C++, we'll be required to create an attribute of class BankDatabase to hold zero or more Account objects. Similarly, we'll assign attributes to class ATM that correspond to its composition relationships with classes Screen, Keypad, CashDispenser and DepositSlot . These composition-based attributes would be redundant if modeled in Fig. 4.20 , because the compositions modeled in Fig. 3.23 already convey the fact that the database contains information about zero or more accounts and that an ATM is composed of a screen, keypad, cash dispenser and deposit slot. Software developers typically model these whole/part relationships as compositions rather than as attributes required to implement the relationships.

The class diagram in Fig. 4.20 provides a solid basis for the structure of our model, but the diagram is not complete. In Section 5.10 , we identify the states and activities of the objects in the model, and in Section 6.22 we identify the operations that the objects perform. As we present more of the UML and object-oriented design, we'll continue to strengthen the structure of our model.

## Software Engineering Case Study Self-Review Exercises

**4.1** We typically identify the attributes of the classes in our system by analyzing the __________ in the requirements specification.

**a.** nouns and noun phrases

**b.** descriptive words and phrases

**c.** verbs and verb phrases

**d.** All of the above.

**4.2** Which of the following is not an attribute of an airplane?

**a.** length

**b.** wingspan

**c.** fly

**d.** number of seats

**4.3** Describe the meaning of the following attribute declaration of class CashDispenser in the class diagram in Fig. 4.20:

```
count : Integer = 500
```

## Answers to Software Engineering Case Study Self-Review Exercises

**4.1** b.

**4.2** c. Fly is an operation or behavior of an airplane, not an attribute.

**4.3** This indicates that count is an Integer with an initial value of 500 . This attribute keeps track of the number of bills available in the CashDispenser at any given time.

## 4.12. Wrap-Up

You learned that only three types of control structures—sequence, selection and repetition—are needed to develop any algorithm. We demonstrated two of C++'s selection statements—the if single-selection statement and the if...else double-selection statement. The if statement is used to execute a set of statements based on a condition—if the condition is true, the statements execute; if it is not, the statements are skipped. The if...else double-selection statement is used to execute one set of statements if a condition is true, and another set of statements if the condition is false. We then discussed the while repetition statement, where a set of statements are executed repeatedly as long as a condition is true. We used control-statement stacking to total and compute the average of a set of student grades with counter- and sentinel-controlled repetition, and we used control-statement nesting to analyze and make decisions based on a set of exam results. We introduced assignment operators, which can be used for abbreviating statements. We presented the increment and decrement operators, which can be used to add or subtract the value 1 from a variable. In Chapter 5 , Control Statements: Part 2, we continue our discussion of control statements, introducing the for, do...while and switch statements.

# 5. Control Statements: Part 2

**Objectives**

In this chapter you'll learn:

- To use the for and do...while repetition statements to execute statements in a program repeatedly.
- To implement multiple selection using the switch selection statement.
- To use the break and continue program control statements to alter the flow of control.
- To use the logical operators to form complex conditional expressions in control statements.
- To avoid the consequences of confusing the equality and assignment operators.

Not everything that can be counted counts, and not every thing that counts can be counted.

—***Albert Einstein***

Who can control his fate?

—***William Shakespeare***

The used key is always bright.

—***Benjamin Franklin***

Intelligence ... is the faculty of making artificial objects, especially tools to make tools.

— ***Henri Bergson***

Every advantage in the past is judged in the light of the final issue.

—***Demosthenes***

**Outline**

## 5.1. Introduction

In this chapter, we introduce C++'s remaining control statements. The control statements we study here and in Chapter 4 will help us build and manipulate objects. We continue our early emphasis on object-oriented programming that began with a discussion of basic concepts in Chapter 1 and the extensive object-oriented code examples in Chapters 3–4.

In this chapter, we demonstrate the for, do...while and switch statements. Through a series of short examples using while and for, we explore the essentials of counter-controlled repetition. We expand the GradeBook class presented in Chapters 3–4. In particular, we create a version of class GradeBook that uses a switch statement to count the number of A, B, C, D and F grades in a set of letter grades entered by the user. We introduce the break and continue program control statements. We discuss the logical operators, which enable you to use more powerful conditional expressions in control statements. We also examine the common error of confusing the equality (==) and assignment (=) operators, and how to avoid it.

## 5.2. Essentials of Counter-Controlled Repetition

This section uses the while repetition statement introduced in Chapter 4 to formalize the elements required to perform counter-controlled repetition. Counter-controlled repetition requires

1. the name of a control variable (or loop counter)
2. the initial value of the control variable
3. the loop-continuation condition that tests for the final value of the control variable (i.e., whether looping should continue)
4. the increment (or decrement) by which the control variable is modified each time through the loop.

Consider the simple program in Fig. 5.1, which prints the numbers from 1 to 10. The declaration in line 9 *names* the control variable (counter), declares it to be an integer, reserves space for it in memory and sets it to an *initial value* of 1. Declarations that require initialization are, in effect, executable statements. In C++, it is more precise to call a declaration that also reserves memory—as the preceding declaration does—a definition. Because definitions are declarations, too, we'll use the term "declaration" except when the distinction is important.

**Fig. 5.1. Counter-controlled repetition.**

```
 1 // Fig. 5.1: fig05_01.cpp
 2 // Counter-controlled repetition.
 3 #include <iostream>
 4 using std::cout;
 5 using std::endl;
 6
 7 int main()
 8 {
 9    int counter = 1; // declare and initialize control variable
10
11    while ( counter <= 10 ) // loop-continuation condition
12    {
13       cout << counter << " ";
14       counter++; // increment control variable by 1
15    } // end while
16
17    cout << endl; // output a newline
18    return 0; // successful termination
19 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

The declaration and initialization of counter (line 9) also could have been accomplished with the statements

```
int counter; // declare control variable
counter = 1; // initialize control variable to 1
```

We use both methods of initializing variables.

Line 14 *increments* the loop counter by 1 each time the loop's body is performed. The loop-continuation condition (line 11) in the while statement determines whether the value of the control variable is less than or equal to 10 (the final value for which the condition is true). Note that the body of this while executes even when the control variable is 10 . The loop terminates when the control variable is greater than 10 (i.e., when counter becomes 11).

Figure 5.1 can be made more concise by initializing counter to 0 and by replacing the while statement with

```
while ( ++counter <= 10 ) // loop-continuation condition
   cout << counter << " " ;
```

This code saves a statement, because the incrementing is done directly in the while condition before the condition is tested. Also, the code eliminates the braces around the body of the while, because the while now contains only one statement. Coding in such a condensed fashion takes some practice and can lead to programs that are more difficult to read, debug, modify and maintain.

Common Programming Error 5.1

*Floating-point values are approximate, so controlling counting loops with floating-point variables can result in imprecise counter values and inaccurate tests for termination.*

Error-Prevention Tip 5.1

*Control counting loops with integer values.*

## 5.3. for Repetition Statement

Section 5.2 presented the essentials of counter-controlled repetition. The while statement can be used to implement any counter-controlled loop. C++ also provides the for repetition statement, which specifies the counter-controlled repetition details in a single line of code. To illustrate the power of for, let us rewrite the program of Fig. 5.1. The result is shown in Fig. 5.2.

**Fig. 5.2. Counter-controlled repetition with the for statement.**

```
1  // Fig. 5.2: fig05_02.cpp
2  // Counter-controlled repetition with the for statement.
3  #include<iostream>
4  usingstd::cout;
5  usingstd::endl;
6
7  int main()
8  {
9    // for statement header includes initialization,
10    // loop-continuation condition and increment.
11    for( int counter = 1; counter <= 10; counter++ )
12       cout << counter << " ";
13
14    cout << endl;// output a newline
15    return 0; // indicate successful termination
16  } // end main
```

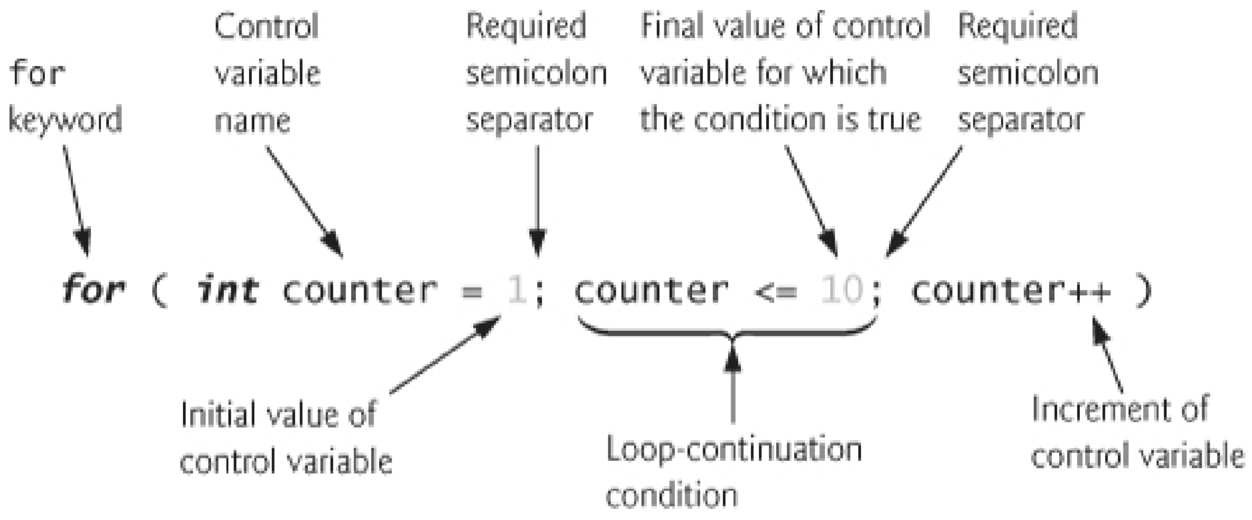
```
1 2 3 4 5 6 7 8 9 10
```

When the for statement (lines 11–12) begins executing, the control variable counter is declared and initialized to 1. Then, the loop-continuation condition (line 11 between the semicolons) counter <= 10 is checked. The initial value of counter is 1, so the condition is satisfied and the body statement (line 12) prints the value of counter, namely 1. Then, the expression counter++ increments control variable counter and the loop begins again with the loop-continuation test. The control variable is now equal to 2, so the final value is not exceeded and the program performs the body statement again. This process continues until the loop body has executed 10 times and the control variable counter is incremented to 11—this causes the loop-continuation test to fail and repetition to terminate. The program continues by performing the first statement after the for statement (in this case, the output statement in line 14).

### for Statement Header Components

Figure 5.3 takes a closer look at the for statement header (line 11) of Fig. 5.2. Notice that the for statement header "does it all"—it specifies each of the items needed for counter-controlled repetition with a control variable. If there is more than one statement in the body of the for, braces are required to enclose the body of the loop.

**Fig. 5.3. for statement header components.**


for keyword
Control variable name
Required semicolon separator
Final value of control variable for which the condition is true
Required semicolon separator
for ( int counter = 1; counter <= 10; counter++ )
Initial value of control variable
Loop-continuation condition
Increment of control variable

Notice that Fig. 5.2 uses the loop-continuation condition `counter <= 10`. If you incorrectly wrote `counter < 10`, then the loop would execute only 9 times. This is a common off-by-one error.

Common Programming Error 5.2

*Using an incorrect relational operator or using an incorrect final value of a loop counter in the condition of a* `while` *or* `for` *statement can cause off-by-one errors.*

The general form of the `for` statement is

```
for ( initialization; loopContinuationCondition; increment )
   statement
```

where the *initialization* expression initializes the loop's control variable, *loopContinuationCondition* determines whether the loop should continue executing (this condition typically contains the final value of the control variable for which the condition is true) and *increment* increments the control variable. In most cases, the `for` statement can be represented by an equivalent `while` statement, as follows:

```
initialization;

while ( loopContinuationCondition )
{
   statement
   increment;
}
```

There is an exception to this rule, which we'll discuss in Section 5.7.

If the *initialization* expression in the `for` statement header declares the control variable (i.e., the control variable's type is specified before the variable name), the control variable can be used only in the body of the `for` statement—the control variable will be unknown outside the `for` statement. This restricted use of the control variable name is known as the variable's scope. The scope of a variable specifies where it can be used in a program. Scope is discussed in detail in Chapter 6, Functions and an Introduction to Recursion.

Common Programming Error 5.3

*When the control variable of a* for *statement is declared in the initialization section of the* for *statement header, using the control variable after the body of the statement is a compilation error.*

Portability Tip 5.1

*In the C++ standard, the scope of the control variable declared in the initialization section of a* for *statement differs from the scope in older C++ compilers. In prestandard compilers, the scope of the control variable does not terminate at the end of the block defining the body of the* for *statement; rather, the scope terminates at the end of the block that encloses the* for *statement. C++ code created with prestandard C++ compilers can break when compiled on standard-compliant compilers. If you are working with prestandard compilers and you want to be sure your code will work with standard-compliant compilers, there are two defensive programming strategies you can use: either declare control variables with different names in every* for *statement, or, if you prefer to use the same name for the control variable in several* for *statements, declare the control variable before the first* for *statement.*

As we'll see, the *initialization* and *increment* expressions can be comma-separated lists of expressions. The commas, as used in these expressions, are comma operators, which guarantee that lists of expressions evaluate from left to right. The comma operator has the lowest precedence of all C++ operators. The value and type of a comma-separated list of expressions is the value and type of the rightmost expression in the list. The comma operator is most often used in for statements. Its primary application is to enable you to use multiple initialization expressions and/or multiple increment expressions. For example, there may be several control variables in a single for statement that must be initialized and incremented.

Good Programming Practice 5.1

*Place only expressions involving the control variables in the initialization and increment sections of a* for *statement. Manipulations of other variables should appear either before the loop (if they should execute only once, like initialization statements) or in the loop body (if they should execute once per repetition, like incrementing or decrementing statements).*

The three expressions in the for statement header are optional (but the two semicolon separators are required). If the *loopContinuationCondition* is omitted, C++ assumes that the condition is true, thus creating an infinite loop. One might omit the *initialization* expression if the control variable is initialized earlier in the program. One might omit the *increment* expression if the increment is calculated by statements in the body of the for or if no increment is needed. The increment expression in the for statement acts as a standalone statement at the end of the body of the for. Therefore, the expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```

are all equivalent in the incrementing portion of the for statement (when no other code appears there). Many programmers prefer the form counter++, because for loops evaluate the increment expression *after* the loop body executes. The postincrementing form therefore seems more natural. The variable being incremented here does not appear in a larger expression, so both preincrementing and postincrementing actually have the *same* effect.

Common Programming Error 5.4

*Using commas instead of the two required semicolons in a for header is a syntax error.*

Common Programming Error 5.5

*Placing a semicolon immediately to the right of the right parenthesis of a for header makes the body of that for statement an empty statement. This is usually a logic error.*

The initialization, loop-continuation condition and increment expressions of a for statement can contain arithmetic expressions. For example, if x = 2 and y = 10, and x and y are not modified in the loop body, the for header

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

is equivalent to

```
for ( int j = 2; j <= 80; j += 5 )
```

The "increment" of a for statement can be negative, in which case it is really a decrement and the loop actually counts downward (as shown in Section 5.4).

If the loop-continuation condition is initially false, the body of the for statement is not performed. Instead, execution proceeds with the statement following the for.

Frequently, the control variable is printed or used in calculations in the body of a for statement, but this is not required. It is common to use the control variable for controlling repetition while never mentioning it in the body of the for statement.

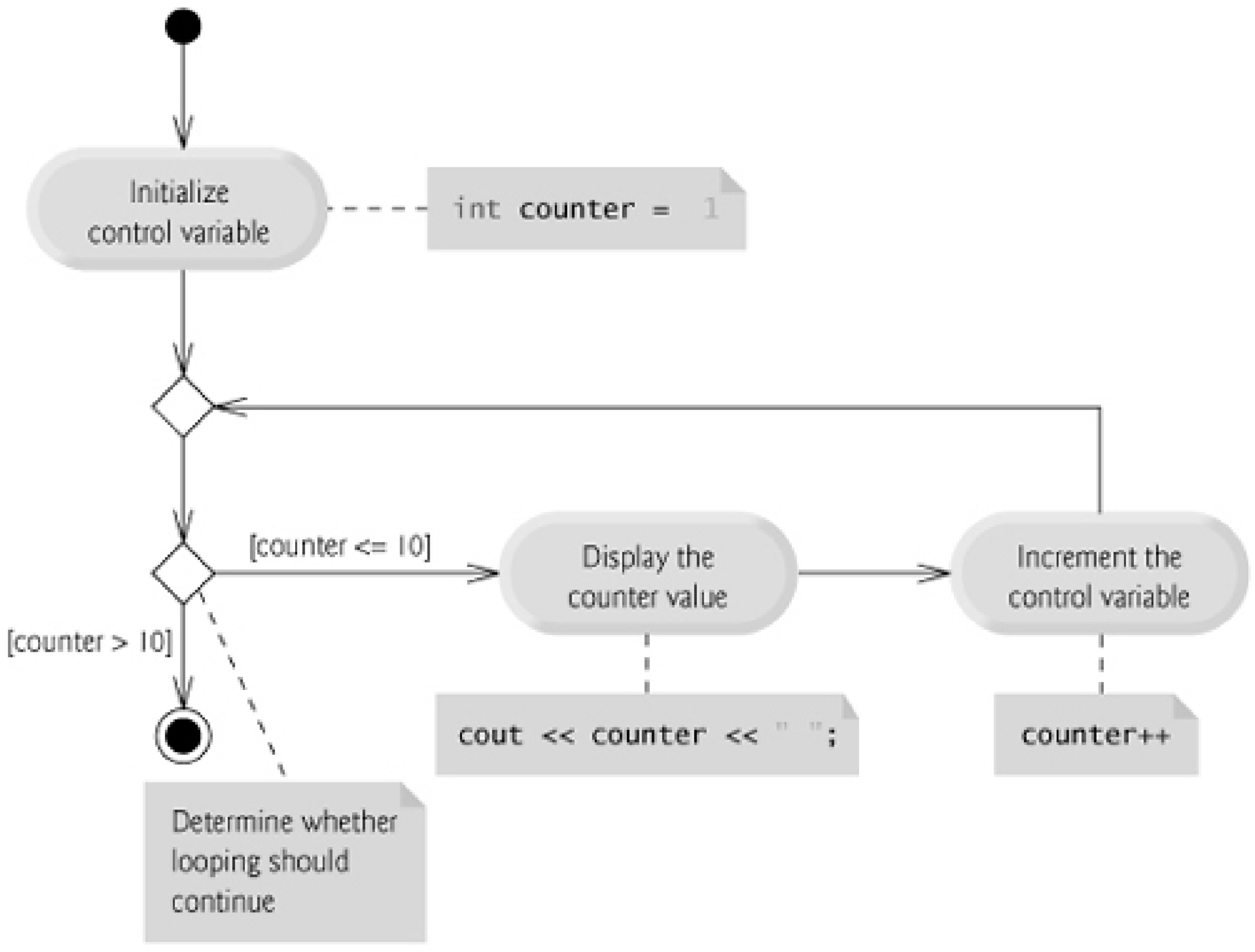
Error-Prevention Tip 5.2

*Although the value of the control variable can be changed in the body of a* for *statement, avoid doing so, because this practice can lead to subtle logic errors.*

### for **Statement UML Activity Diagram**

The for statement's UML activity diagram is similar to that of the while statement (Fig. 4.5). Figure 5.4 shows the activity diagram of the for statement in Fig. 5.2 . The diagram makes it clear that initialization occurs once before the loop-continuation test is evaluated the first time, and that incrementing occurs each time through the loop *after* the body statement executes. Note that (besides an initial state, transition arrows, a merge, a final state and several notes) the diagram contains only action states and a decision.

**Fig. 5.4. UML activity diagram for the for statement in Fig. 5.2.**



Initialize control variable
int counter = 1
[counter <= 10]
Display the counter value
Increment the control variable
[counter > 10]
cout << counter << " ";
counter++
Determine whether looping should continue

## 5.4. Examples Using the `for` Statement

The following examples show methods of varying the control variable in a `for` statement. In each case, we write the appropriate `for` statement header. Note the change in the relational operator for loops that decrement the control variable.

**a.** Vary the control variable from `1` to `100` in increments of `1`.

```
for ( int i = 1; i <=100; i++ )
```

**b.** Vary the control variable from `100` down to `1` in increments of `-1` (that is, decrements of `1`).

```
for ( int i = 100 ; i >= 1; i-- )
```

**c.** Vary the control variable from `7` to `77` in steps of `7`.

```
for ( int i = 7; i <= 77; i += 7 )
```

**d.** Vary the control variable from `20` down to `2` in steps of `-2`.

```
for ( int i = 20; i >= 2; i -= 2 )
```

**e.** Vary the control variable over the following sequence of values: `2`, `5`, `8`, `11`, `14`, `17`, `20`.

```
for ( int i = 2; i <= 20; i += 3 )
```

**f.** Vary the control variable over the following sequence of values: `99`, `88`, `77`, `66`, `55`, `44`, `33`, `22`, `11`, `0`.

```
for ( int i = 99; i >= 0; i -= 11 )
```

Common Programming Error 5.6

*Not using the proper relational operator in the loop-continuation condition of a loop that counts downward (such as incorrectly using `i <= 1` instead of `i >= 1` in a loop counting down to 1) is usually a logic error that yields incorrect results when the program runs.*

### Application: Summing the Even Integers from 2 to 20

The next two examples provide simple applications of the `for` statement. The program of Fig. 5.5 uses a `for` statement to sum the even integers from 2 to 20. Each iteration of the loop (lines 12–13) adds the current value of the control variable `number` to variable `total`.

**Fig. 5.5. Summing integers with the for statement.**

```
// Fig. 5.5: fig05_05.cpp
// Summing integers with the for statement.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   int total = 0; // initialize total

   // total even integers from 2 through 20
   for ( int number = 2; number <= 20; number += 2 )
      total += number;

   cout << "Sum is " << total << endl; // display results
   return 0; // successful termination
} // end main
```

```
Sum is 110
```

Note that the body of the for statement in Fig. 5.5 actually could be merged into the increment portion of the for header by using the comma operator as follows:

```
for ( int number = 2; // initialization
     number <= 20; // loop continuation condition
     total += number, number += 2 ) // total and increment
  ; // empty body
```

Good Programming Practice 5.2

*Although statements preceding a for and statements in the body of a for often can be merged into the for header, doing so can make the program more difficult to read, maintain, modify and debug.*

Good Programming Practice 5.3

*Limit the size of control statement headers to a single line, if possible.*

## Application: Compound Interest Calculations

The next example computes compound interest using a `for` statement. Consider the following problem statement:

> *A person invests $1000.00 in a savings account yielding 5 percent interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:*
>
> $$a = p\,(1 + r)^n$$
>
> *where*
>
> p *is the original amount invested (i.e., the principal),*
>
> r *is the annual interest rate,*
>
> n *is the number of years and*
>
> a *is the amount on deposit at the end of the* n*th year.*

This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit. The solution is shown in Fig. 5.6.

**Fig. 5.6. Compound interest calculations with `for`.**

```cpp
// Fig. 5.6: fig05_06.cpp
// Compound interest calculations with for.
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setw; // enables program to set a field width
using std::setprecision;

#include <cmath> // standard C++ math library
using std::pow; // enables program to use function pow

int main()
{
   double amount; // amount on deposit at end of each year
   double principal = 1000.0; // initial amount before interest
   double rate = .05; // interest rate

   // display headers
   cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;

   // set floating-point number format
   cout << fixed << setprecision( 2 );

   // calculate amount on deposit for each of ten years
   for ( int year = 1; year <= 10; year++ )
   {
      // calculate new amount for specified year
      amount = principal * pow( 1.0 + rate, year );

      // display the year and the amount
      cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
   } // end for

   return 0; // indicate successful termination
} // end main
```

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
```

```
  10          1628.89
```

The for statement (lines 28–35) executes its body 10 times, varying a control variable from 1 to 10 in increments of 1. C++ does not include an exponentiation operator, so we use the standard library function pow (line 31) for this purpose. The function pow(x, y) calculates the value of x raised to the y$^{th}$ power. In this example, the algebraic expression $(1+r)^n$ is written as pow( 1.0 + rate, year ), where variable rate represents $r$ and variable year represents $n$. Function pow takes two arguments of type double and returns a double value.

This program will not compile without including header file <cmath> (line 12). Function pow requires two double arguments. Note that year is an integer. Header <cmath> includes information that tells the compiler to convert the value of year to a temporary double representation before calling the function. This information is contained in pow's function prototype. Chapter 6 summarizes other math library functions.

Common Programming Error 5.7

*In general, forgetting to include the appropriate header file when using standard library functions (e.g., <cmath> in a program that uses math library functions) is a compilation error.*

### A Caution about Using Type float or double for Monetary Amounts

Notice that lines 17–19 declare the double variables amount, principal and rate . We did this for simplicity because we're dealing with fractional parts of dollars, and we need a type that allows decimal points in its values. Unfortunately, this can cause trouble. Here is a simple explanation of what can go wrong when using float or double to represent dollar amounts (assuming setprecision( 2 ) is used to specify two digits of precision when printing): Two dollar amounts stored in the machine could be 14.234 (which prints as 14.23) and 18.673 (which prints as 18.67). When these amounts are added, they produce the internal sum 32.907, which prints as 32.91. Thus your printout could appear as

```
  14.23
+ 18.67
-------
  32.91
```

but a person adding the individual numbers as printed would expect the sum 32.90! You have been warned!

Good Programming Practice 5.4

*Do not use variables of type* `float` *or* `double` *to perform monetary calculations. The imprecision of floating-point numbers can cause errors that result in incorrect monetary values.* [Note: *Some third-party vendors sell C++ class libraries that perform precise monetary calculations.*]

### Using Stream Manipulators to Format Numeric Output

The output statement in line 25 before the `for` loop and the output statement in line 34 in the `for` loop combine to print the values of the variables `year` and `amount` with the formatting specified by the parameterized stream manipulators `setprecision` and `setw` and the nonparameterized stream manipulator `fixed`. The stream manipulator `setw( 4 )` specifies that the next value output should appear in a field width of 4—i.e., `cout` prints the value with at least 4 character positions. If the value to be output is less than 4 character positions wide, the value is right justified in the field by default. If the value to be output is more than 4 character positions wide, the field width is extended to accommodate the entire value. To indicate that values should be output left justified, simply output nonparameterized stream manipulator `left` (found in header `<iostream>`). Right justification can be restored by outputting nonparameterized stream manipulator `right`.

The other formatting in the output statements indicates that variable `amount` is printed as a fixed-point value with a decimal point (specified in line 25 with the stream manipulator `fixed`) right justified in a field of 21 character positions (specified in line 34 with `setw( 21 )`) and two digits of precision to the right of the decimal point (specified in line 25 with manipulator `setprecision( 2 )`). We applied the stream manipulators `fixed` and `setprecision` to the output stream (i.e., `cout`) before the `for` loop because these format settings remain in effect until they are changed—such settings are called sticky settings and they do not need to be applied during each iteration of the loop. However, the field width specified with `setw` applies only to the next value output. We discuss C++'s powerful input/output formatting capabilities in Chapter 15, Stream Input/Output.

Note that the calculation `1.0 + rate`, which appears as an argument to the `pow` function, is contained in the body of the `for` statement. In fact, this calculation produces the same result during each iteration of the loop, so repeating it is wasteful—it should be performed once before the loop.

Performance Tip 5.1

*Avoid placing expressions whose values do not change inside loops—but, even if you do, many of today's sophisticated optimizing compilers will automatically place such expressions outside the loops in the generated machine-language code.*

Performance Tip 5.2

*Many compilers contain optimization features that improve the performance of the code you write, but it is still better to write good code from the start.*

## 5.5. do...while Repetition Statement

The do...while repetition statement is similar to the while statement. In the while statement, the loop-continuation condition test occurs at the beginning of the loop before the body of the loop executes. The do...while statement tests the loop-continuation condition *after* the loop body executes; therefore, the loop body always executes at least once. When a do...while terminates, execution continues with the statement after the while clause. Note that it is not necessary to use braces in the do...while statement if there is only one statement in the body; however, most programmers include the braces to avoid confusion between the while and do...while statements. For example,

```
while ( condition )
```

normally is regarded as the header of a while statement. A do...while with no braces around the single statement body appears as

```
do
  statement
while ( condition );
```

which can be confusing. You might misinterpret the last line—while( *condition* );—as a while statement containing as its body an empty statement. Thus, the do...while with one statement often is written as follows to avoid confusion:

```
do
{
  statement
} while ( condition );
```

Good Programming Practice 5.5

*Always including braces in a do...while statement helps eliminate ambiguity between the while statement and the do...while statement containing one statement.*

Figure 5.7 uses a do...while statement to print the numbers 1–10. Upon entering the do...while statement, line 13 outputs counter's value and line 14 increments counter. Then the program evaluates the loop-continuation test at the bottom of the loop (line 15). If the condition is true, the loop continues from the first body statement in the do...while (line 13). If the condition is false, the loop terminates and the program continues with the next statement after the loop (line 17).

Fig. 5.7. do...while repetition statement.

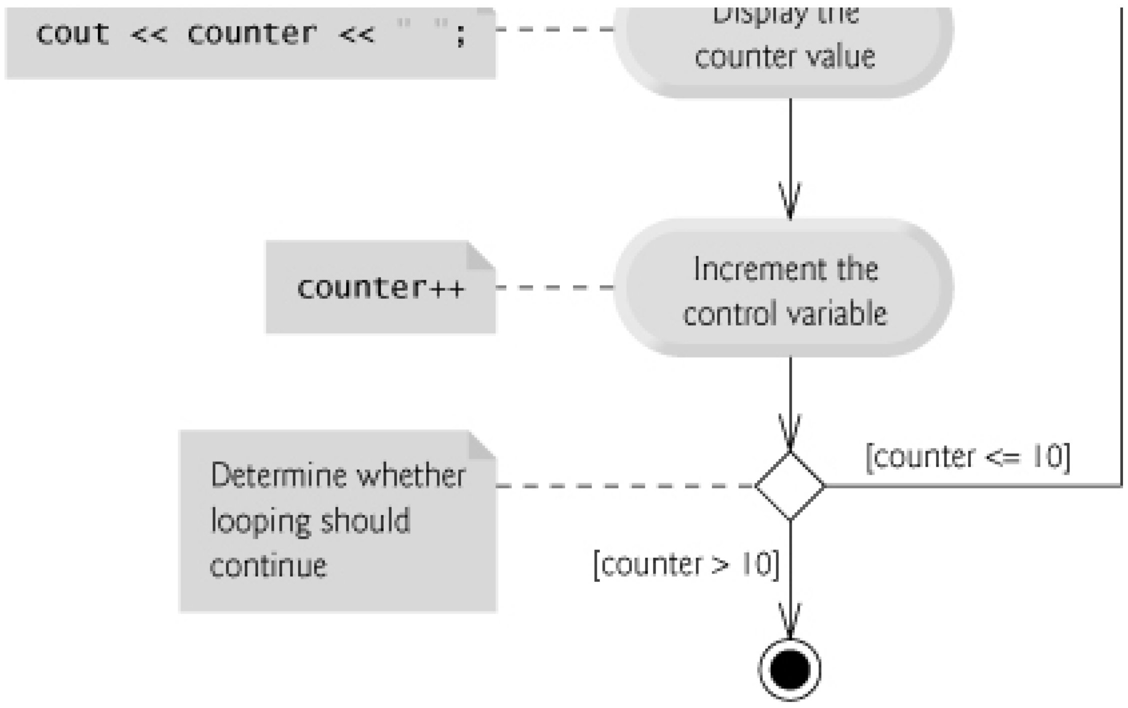
```
// Fig. 5.7: fig05_07.cpp
// do...while repetition statement.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   int counter = 1; // initialize counter

   do
   {
      cout << counter << " "; // display counter
      counter++; // increment counter
   } while ( counter <= 10 ); // end do...while

   cout << endl; // output a newline
   return 0; // indicate successful termination
} // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

## do...while Statement UML Activity Diagram

Figure 5.8 contains the UML activity diagram for the do...while statement. This diagram makes it clear that the loop-continuation condition is not evaluated until after the loop performs the loop-body action states at least once. Compare this activity diagram with that of the while statement (Fig. 4.5). Again, note that (besides an initial state, transition arrows, a merge, a final state and several notes) the diagram contains only action states and a decision. Imagine, again, that you have access to a bin of empty do...while statement UML activity diagrams—as many as you might need to stack and nest with the activity diagrams of other control statements to form a structured implementation of an algorithm. You fill in the action states and decision symbols with action expressions and guard conditions appropriate to the algorithm.

Fig. 5.8. UML activity diagram for the do...while repetition statement of Fig. 5.7.


cout << counter << " ";
Display the counter value
counter++
Increment the control variable
Determine whether looping should continue
[counter <= 10]
[counter > 10]

## 5.6. switch Multiple-Selection Statement

We discussed the if single-selection statement and the if...else double-selection statement in Chapter 4. C++ provides the switch multiple-selection statement to perform many different actions based on the possible values of a variable or expression. Each action is associated with the value of a constant integral expression (i.e., any combination of character constants and integer constants that evaluates to a constant integer value) to which the variable or expression may evaluate.

### GradeBook Class with switch Statement to Count A, B, C, D and F Grades

We now present an enhanced version of the GradeBook class introduced in Chapter 3 and further developed in Chapter 4 . The new version of the class asks the user to enter a set of letter grades, then displays a summary of the number of students who received each grade. The class uses a switch to determine whether each grade entered is an A, B, C, D or F and to increment the appropriate grade counter. Class GradeBook is defined in Fig. 5.9 , and its member-function definitions appear in Fig. 5.10. Figure 5.11 shows sample inputs and outputs of the main program that uses class GradeBook to process a set of grades.

**Fig. 5.9. GradeBook class definition.**

```
// Fig. 5.9: GradeBook.h
// Definition of class GradeBook that counts A, B, C, D and F grades.
// Member functions are defined in GradeBook.cpp

#include <string> // program uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
public:
   GradeBook( string ); // constructor initializes course name
   void setCourseName( string ); // function to set the course name
   string getCourseName(); // function to retrieve the course name
   void displayMessage(); // display a welcome message
   void inputGrades(); // input arbitrary number of grades from user
   void displayGradeReport(); // display a report based on the grades
private:
   string courseName; // course name for this GradeBook
   int aCount; // count of A grades
   int bCount; // count of B grades
   int cCount; // count of C grades
   int dCount; // count of D grades
   int fCount; // count of F grades
}; // end class GradeBook
```

Fig. 5.10. GradeBook class uses a switch statement to count letter grades A, B, C, D and F.

```
// Fig. 5.10: GradeBook.cpp
// Member-function definitions for class GradeBook that
// uses a switch statement to count A, B, C, D and F grades.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "GradeBook.h"   // include definition of class GradeBook

// constructor initializes courseName with string supplied as argument;
// initializes counter data members to 0
GradeBook::GradeBook( string name )
{
   setCourseName( name ); // validate and store courseName
   aCount = 0; // initialize count of A grades to 0
   bCount = 0; // initialize count of B grades to 0
   cCount = 0; // initialize count of C grades to 0
   dCount = 0; // initialize count of D grades to 0
   fCount = 0; // initialize count of F grades to 0
} // end GradeBook constructor

// function to set the course name; limits name to 25 or fewer characters
void GradeBook::setCourseName( string name )
{
   if ( name.length() <= 25 ) // if name has 25 or fewer characters
      courseName = name; // store the course name in the object
   else // if name is longer than 25 characters
   { // set courseName to first 25 characters of parameter name
      courseName = name.substr( 0, 25 ); // select first 25 characters
      cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
         << "Limiting courseName to first 25 characters.\n" << endl;
   } // end if...else
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName()
{
   return courseName;
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   // this statement calls getCourseName to get the
   // name of the course this GradeBook represents
   cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
      << endl;
} // end function displayMessage

// input arbitrary number of grades from user; update grade counter
void GradeBook::inputGrades()
{
   int grade; // grade entered by user

   cout << "Enter the letter grades."   << endl
      << "Enter the EOF character to end input."    << endl;

   // loop until user types end-of-file key sequence
   while ( ( grade = cin.get() ) != EOF )
   {
      // determine which grade was entered
      switch ( grade ) // switch statement nested in while
      {
         case 'A' : // grade was uppercase A
         case 'a': // or lowercase a
            aCount++; // increment aCount
            break; // necessary to exit switch

         case 'B' : // grade was uppercase B
         case 'b' : // or lowercase b
            bCount++; // increment bCount
            break; // exit switch

         case 'C' : // grade was uppercase C
         case 'c' : // or lowercase c
            cCount++; // increment cCount
            break; // exit switch

         case 'D' : // grade was uppercase D
         case 'd' : // or lowercase d
            dCount++; // increment dCount
            break; // exit switch

         case 'F' : // grade was uppercase F
         case 'f' : // or lowercase f
            fCount++; // increment fCount
            break; // exit switch

         case '\n' : // ignore newlines,
         case '\t' : // tabs,
         case ' ' : // and spaces in input
            break; // exit switch

         default: // catch all other characters
            cout << "Incorrect letter grade entered."
               << " Enter a new grade."   << endl;
            break; // optional; will exit switch anyway
      } // end switch
   } // end while
} // end function inputGrades

// display a report based on the grades entered by user
void GradeBook::displayGradeReport()
{
   // output summary of results
   cout << "\n\nNumber of students who received each letter grade:"
      << "\nA: "  << aCount // display number of A grades
      << "\nB: "  << bCount // display number of B grades
      << "\nC: "  << cCount // display number of C grades
      << "\nD: "  << dCount // display number of D grades
      << "\nF: "  << fCount // display number of F grades
      << endl;
} // end function displayGradeReport
```

**Fig. 5.11. Creating a GradeBook object and calling its member functions.**

```
// Fig. 5.11: fig05_11.cpp
// Create GradeBook object, input grades and display grade report.

#include "GradeBook.h" // include definition of class GradeBook

int main()
{
  // create GradeBook object
  GradeBook myGradeBook( "CS101 C++ Programming" );

   myGradeBook.displayMessage(); // display welcome message
   myGradeBook.inputGrades(); // read grades from user
   myGradeBook.displayGradeReport(); // display report based on grades
   return 0; // indicate successful termination
 } // end main
```

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

Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z


Number of students who received each letter grade:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Like earlier versions of the class definition, the GradeBook class definition (Fig. 5.9) contains function prototypes for member functions setCourseName (line 13), getCourseName (line 14) and displayMessage (line 15), as well as the class's constructor (line 12). The class definition also declares private data member courseName (line 19).

Class GradeBook (Fig. 5.9) now contains five additional private data members (lines 20–24)—counter variables for each grade category (i.e., A, B, C, D and F). The class also contains two additional public member functions—inputGrades and displayGradeReport. Member function inputGrades (declared in line 16) reads an arbitrary number of letter grades from the user using sentinel-controlled repetition and updates the appropriate grade counter for each grade entered. Member function displayGradeReport (declared in line 17) outputs a report containing the number of students who received each letter grade.

Source-code file GradeBook.cpp (Fig. 5.10) contains the member-function definitions for class GradeBook. Notice that lines 16–20 in the constructor initialize the five grade counters to 0—when a GradeBook object is first created, no grades have been entered yet. As you'll soon see, these counters are incremented in member function inputGrades as the user enters grades. The definitions of member functions setCourseName, getCourseName and displayMessage are identical to those found in the earlier versions of class GradeBook. Let's consider the new GradeBook member functions in detail.

**Reading Character Input**

The user enters letter grades for a course in member function inputGrades (lines 52–101). Inside the while header, in line 60, the parenthesized assignment ( grade = cin.get() ) executes first. The cin.get() function reads one character from the keyboard and stores that character in integer variable grade (declared in line 54). Characters normally are stored in variables of type char; however, characters can be stored in any integer data type, because types short, int and long are guaranteed to be at least as big as type char. Thus, we can treat a character either as an integer or as a character, depending on its use. For example, the statement

```
cout << "The character (" << 'a' << ") has the value "
  << static_cast< int > ( 'a' ) << endl;
```

prints the character a and its integer value as follows:

```
The character (a) has the value 97
```

The integer 97 is the character's numerical representation in the computer. Most computers today use the ASCII (American Standard Code for Information Interchange) character set, in which 97 represents the lowercase letter 'a'. A table of the ASCII characters and their decimal equivalents is presented in Appendix B, ASCII Character Set.

Assignment statements as a whole have the value that is assigned to the variable on the left side of the =. Thus, the value of the assignment expression grade = cin.get() is the same as the value returned by cin.get() and assigned to the variable grade.

The fact that assignment expressions have values can be useful for assigning the same value to several variables. For example,

```
a = b = c = 0;
```

first evaluates the assignment c = 0 (because the = operator associates from right to left). The variable b is then assigned the value of the assignment c = 0 (which is 0). Then, the variable a is assigned the value of the assignment b = (c = 0) (which is also 0). In the program, the value of the assignment grade = cin.get() is compared with the value of EOF (a symbol whose acronym stands for "end-of-file"). We use EOF (which normally has the value –1) as the sentinel value. *However, you do not type the value –1, nor do you type the letters EOF as the sentinel value.* Rather, you type a system-dependent keystroke combination that means "end-of-file" to indicate that you have no more data to enter. EOF is a symbolic integer constant defined in the <iostream> header file. If the value assigned to grade is equal to EOF, the while loop (lines 60–100) terminates. We have chosen to represent the characters entered into this program as ints, because EOF has type int.

On UNIX/Linux systems and many others, end-of-file is entered by typing

*<Ctrl> d*

on a line by itself. This notation means to press and hold down the *Ctrl* key, then press the *d* key. On other systems such as Microsoft Windows, end-of-file can be entered by typing

*<Ctrl> z*

[*Note:* In some cases, you must press *Enter* after the preceding key sequence. Also, the characters ^Z sometimes appear on the screen to represent end-of-file, as shown in Fig. 5.11.]

Portability Tip 5.2

*The keystroke combinations for entering end-of-file are system dependent.*

Portability Tip 5.3

*Testing for the symbolic constant EOF rather than –1 makes programs more portable. The ANSI/ISO C standard, from which C++ adopts the definition of EOF, states that EOF is a negative integral value (but not necessarily –1), so EOF could have different values on different systems.*

In this program, the user enters grades at the keyboard. When the user presses the *Enter* (or *Return*) key, the characters are read by the cin.get() function, one character at a time. If the character entered is not end-of-file, the flow of control enters the switch statement (lines 63–99), which increments the appropriate letter-grade counter based on the grade entered.

## switch **Statement Details**

The switch statement consists of a series of case labels and an optional default case. These are used in this example to determine which counter to increment, based on a grade. When the flow of control reaches the switch, the program evaluates the expression in the parentheses (i.e., grade) following keyword switch (line 63). This is called the controlling expression. The switch statement compares the value of the controlling expression with each case label. Assume the user enters the letter C as a grade. The program compares C to each case in the switch. If a match occurs (case 'C': in line 75), the program executes the statements for that case. For the letter C, line 77 increments cCount by 1. The break statement (line 78) causes program control to proceed with the first statement after the switch—in this program, control transfers to line 100. This line marks the end of the body of the while loop that inputs grades (lines 60–100), so control flows to the while's condition (line 60) to determine whether the loop should continue executing.

The cases in our switch explicitly test for the lowercase and uppercase versions of the letters A, B, C, D and F. Note the cases in lines 65–66 that test for the values 'A' and 'a' (both of which represent the grade A). Listing cases consecutively in this manner with no statements between them enables the cases to perform the same set of statements—when the controlling expression evaluates to either 'A' or 'a', the statements in lines 67–68 will execute. Note that each case can have multiple statements. The switch selection statement differs from other control statements in that it does not require braces around multiple statements in each case.

Without break statements, each time a match occurs in the switch, the statements for that case and subsequent cases execute until a break statement or the end of the switch is encountered. This is often referred to as "falling through" to the statements in subsequent cases.

Common Programming Error 5.8

*Forgetting a break statement when one is needed in a switch statement is a logic error.*

Common Programming Error 5.9

*Omitting the space between the word case and the integral value being tested in a switch statement can cause a logic error. For example, writing case3: instead of case 3: simply creates an unused label. In this situation, the switch statement will not perform the appropriate actions when the switch's controlling expression has a value of 3.*

**Providing a default Case**

If no match occurs between the controlling expression's value and a case label, the default case (lines 95–98) executes. We use the default case in this example to process all controlling-expression values that are neither valid grades nor newline, tab or space characters (we discuss how the program handles these whitespace characters shortly). If no match occurs, the default case executes, and lines 96–97 print an error message indicating that an incorrect letter grade was entered. If no match occurs in a switch statement that does not contain a default case, program control simply continues with the first statement after the switch.

Good Programming Practice 5.6

*Provide a default case in switch statements. Cases not explicitly tested in a switch statement without a default case are ignored. Including a default case focuses you on the need to process exceptional conditions. There are situations in which no default processing is needed. Although the case clauses and the default case clause in a switch statement can occur in any order, it is common practice to place the default clause last.*

Good Programming Practice 5.7

*The last case in a switch statement does not require a break statement. Some programmers include this break for clarity and for symmetry with other cases.*

### Ignoring Newline, Tab and Blank Characters in Input

Note that lines 90–93 in the switch statement of Fig. 5.10 cause the program to skip newline, tab and blank characters. Reading characters one at a time can cause some problems. To have the program read the characters, we must send them to the computer by pressing the *Enter* key on the keyboard. This places a newline character in the input after the character we wish to process. Often, this newline character must be specially processed to make the program work correctly. By including the preceding cases in our switch statement, we prevent the error message in the default case from being printed each time a newline, tab or space is encountered in the input.

Common Programming Error 5.10

*Not processing newline and other whitespace characters in the input when reading characters one at a time can cause logic errors.*

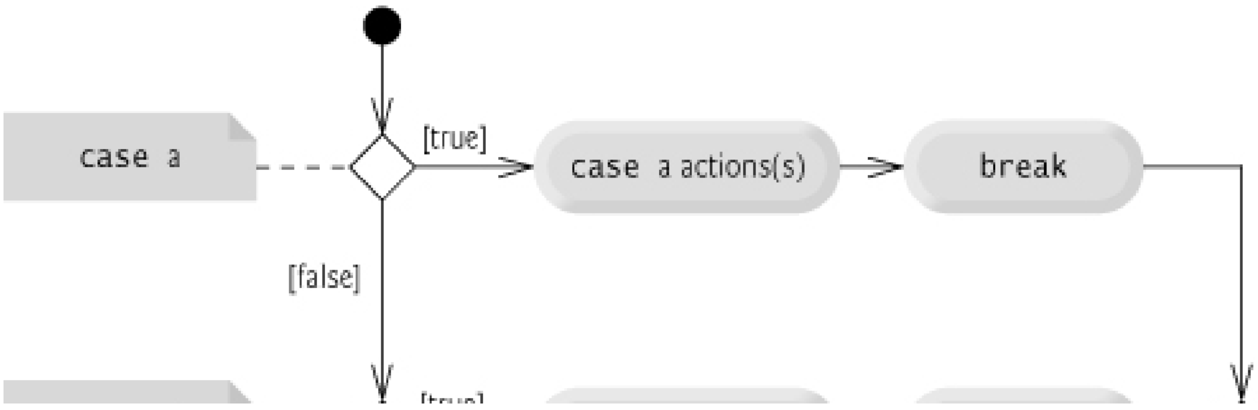
### Testing Class GradeBook

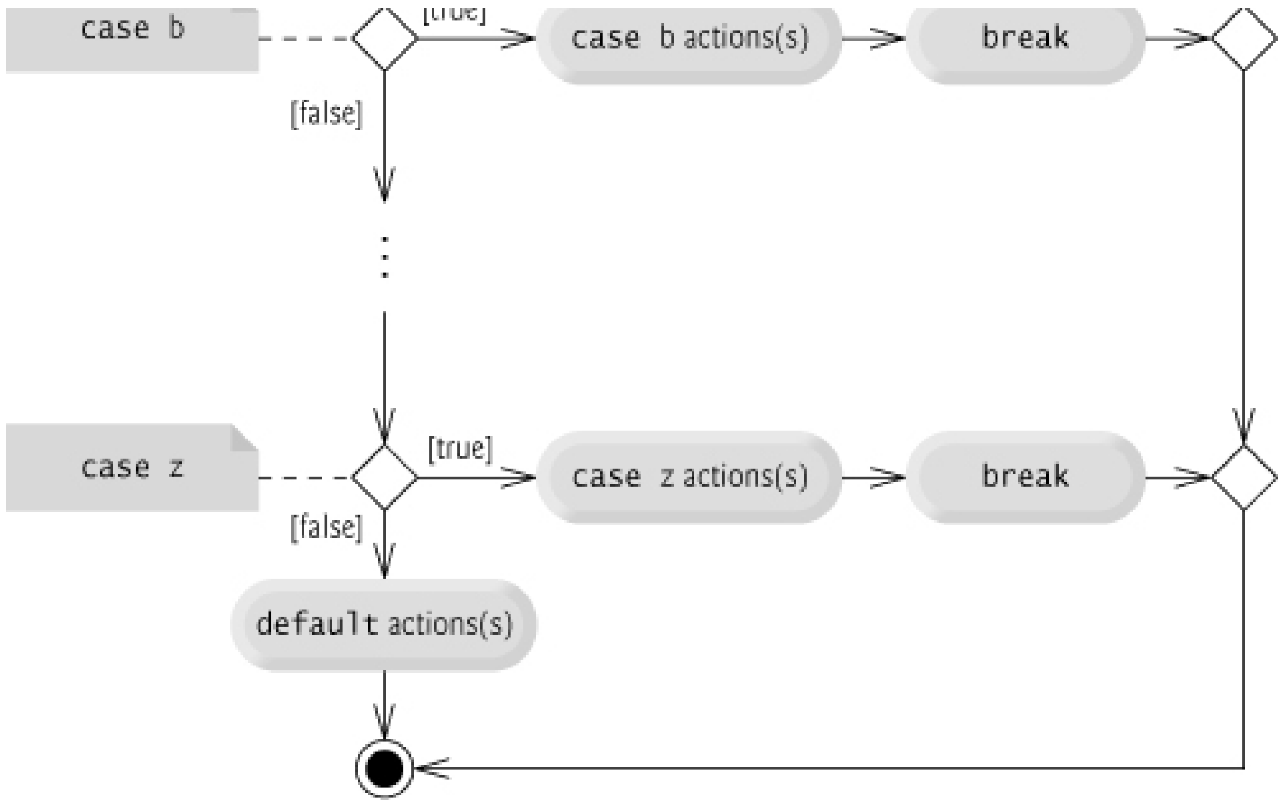
Figure 5.11 creates a GradeBook object (line 9). Line 11 invokes the its displayMessage member function to output a welcome message to the user. Line 12 invokes member function object's inputGrades to read a set of grades from the user and keep track of how many students received each grade. Note that the output window in Fig. 5.11 shows an error message displayed in response to entering an invalid grade (i.e., E). Line 13 invokes GradeBook member function displayGradeReport (defined in lines 104–114 of Fig. 5.10), which outputs a report based on the grades entered (as in the output in Fig. 5.11).

### switch Statement UML Activity Diagram

Figure 5.12 shows the UML activity diagram for the general switch multiple-selection statement. Most switch statements use a break in each case to terminate the switch statement after processing the case. Figure 5.12 emphasizes this by including break statements in the activity diagram. Without the break statement, control would not transfer to the first statement after the switch statement after a case is processed. Instead, control would transfer to the next case's actions.

**Fig. 5.12. switch multiple-selection statement UML activity diagram with break statements.**





The diagram makes it clear that the break statement at the end of a case causes control to exit the switch statement immediately. Again, note that (besides an initial state, transition arrows, a final state and several notes) the diagram contains action states and decisions. Also, note that the diagram uses merge symbols to merge the transitions from the break statements to the final state.

Imagine, again, that you have a bin of empty switch statement UML activity diagrams—as many as you might need to stack and nest with the activity diagrams of other control statements to form a structured implementation of an algorithm. You fill in the action states and decision symbols with action expressions and guard conditions appropriate to the algorithm. Note that, although nested control statements are common, it is rare to find nested switch statements in a program.

When using the switch statement, remember that each case can be used to test only a *constant* integral expression—any combination of character constants and integer constants that evaluates to a constant integer value. A character constant is represented as the specific character in single quotes, such as 'A' . An integer constant is simply an integer value. Also, each case label can specify only one constant integral expression.

Common Programming Error 5.11

*Specifying a nonconstant integral expression in a switch statement's case label is a syntax error.*

Common Programming Error 5.12

*Providing identical case labels in a switch statement is a compilation error. Providing case labels containing different expressions that evaluate to the same value* *also is a compilation error. For example, placing `case 4 + 1:` and `case 3 + 2:` in the same `switch` statement is a compilation error, because these are both equivalent to `case 5:`.*

In Chapter 13, we present a more elegant way to implement `switch` logic. We'll use a technique called polymorphism to create programs that are often clearer, more concise, easier to maintain and easier to extend than programs that use `switch` logic.

**Notes on Data Types**

C++ has flexible data type sizes (see Appendix C, Fundamental Types). Different applications, for example, might need integers of different sizes. C++ provides several data types to represent integers. The range of integer values for each type depends on the particular computer's hardware. In addition to the types `int` and `char`, C++ provides the types `short` (an abbreviation of `short int`) and `long` (an abbreviation of `long int`). The minimum range of values for `short` integers is –32,768 to 32,767. For the vast majority of integer calculations, `long` integers are sufficient. The minimum range of values for `long` integers is –2,147,483,648 to 2,147,483,647. On most computers, `int`s are equivalent either to `short` or to `long`. The range of values for an `int` is at least the same as that for `short` integers and no larger than that for `long` integers. The data type `char` can be used to represent any of the characters in the computer's character set. It also can be used to represent small integers.

Portability Tip 5.4

*Because `int`s can vary in size between systems, use `long` integers if you expect to process integers outside the range –32,768 to 32,767 and you would like to run the program on several different computer systems.*

Performance Tip 5.3

*If memory is at a premium, it might be desirable to use smaller integer sizes.*

Performance Tip 5.4

*Using smaller integer sizes can result in a slower program if the machine's instructions for manipulating them are not as efficient as those for the natural-size integers—i.e., integers whose size equals the machine's word size (e.g., 32 bits on a 32-bit machine, 64 bits on a 64-bit machine). Always test proposed efficiency "upgrades" to be sure they really improve performance.*

## 5.7. break and continue Statements

In addition to the selection and repetition statements, C++ provides statements break and continue to alter the flow of control. The preceding section showed how break can be used to terminate a switch statement's execution. This section discusses how to use break in a repetition statement.

### break Statement

The break statement, when executed in a while, for, do...while or switch statement, causes immediate exit from that statement. Program execution continues with the next statement. Common uses of the break statement are to escape early from a loop or to skip the remainder of a switch statement (as in Fig. 5.10). Figure 5.13 demonstrates the break statement (line 14) exiting a for repetition statement.

**Fig. 5.13. break statement exiting a for statement.**

```
 1  // Fig. 5.13: fig05_13.cpp
 2  // break statement exiting a for statement.
 3  #include <iostream>
 4  using std::cout;
 5  using std::endl;
 6
 7  int main()
 8  {
 9     int count; // control variable also used after loop terminates
10
11     for ( count = 1; count <= 10; count++ ) // loop 10 times
12     {
13        if ( count == 5 )
14           break; // break loop only if x is 5
15
16        cout << count << " ";
17     } // end for
18
19     cout << "\nBroke out of loop at count = " << count << endl;
20     return 0; // indicate successful termination
21  } // end main
```

```
1 2 3 4
Broke out of loop at count = 5
```

When the if statement detects that count is 5, the break statement executes. This terminates the for statement, and the program proceeds to line 19 (immediately after the for statement), which displays a message indicating the control variable value that terminated the loop. The for statement fully executes its body only four times instead of 10. Note that the control variable count is defined outside the for statement header, so that we can use the control variable both in the loop's body and after the loop completes its execution.

## continue Statement

The continue statement, when executed in a while, for or do...while statement, skips the remaining statements in the body of that statement and proceeds with the next iteration of the loop. In while and do...while statements, the loop-continuation test evaluates immediately after the continue statement executes. In the for statement, the increment expression executes, then the loop-continuation test evaluates.

Figure 5.14 uses the continue statement (line 12) in a for statement to skip the output statement (line 14) when the nested (lines 11–12) determines that the value of count is 5. When the continue statement executes, program control continues with the increment of the control variable in the for header (line 9) and loops five more times.

**Fig. 5.14. continue statement terminating a single iteration of a for statement.**

```
1  // Fig. 5.14: fig05_14.cpp
2  // continue statement terminating an iteration of a for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     for ( int count = 1; count <= 10; count++ ) // loop 10 times
10    {
11       if ( count == 5 ) // if count is 5,
12          continue;      // skip remaining code in loop
13
14       cout << count << " ";
15    } // end for
16
17    cout << "\nUsed continue to skip printing 5" << endl;
18    return 0; // indicate successful termination
19 } // end main
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

In Section 5.3, we stated that the while statement could be used in most cases to represent the for statement. The one exception occurs when the increment expression in the while statement follows the continue statement. In this case, the increment does not execute before the program tests the loop-continuation condition, and the while does not execute in the same manner as the for.

Good Programming Practice 5.8

*Some programmers feel that break and continue violate structured programming.*

*The effects of these statements can be achieved by structured programming techniques we soon will see, so these programmers do not use break and continue. Most programmers consider the use of break in switch statements acceptable.*

Performance Tip 5.5

*The break and continue statements, when used properly, perform faster than do the corresponding structured techniques.*

Software Engineering Observation 5.1

*There is a tension between achieving quality software engineering and achieving the best-performing software. Often, one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following rule of thumb: First, make your code simple and correct; then make it fast and small, but only if necessary.*

## 5.8. Logical Operators

So far we have studied only simple conditions, such as counter <= 10, total > 1000 and number != sentinelValue. We expressed these conditions in terms of the relational operators >, <, >= and <= , and the equality operators == and != . Each decision tested precisely one condition. To test multiple conditions while making a decision, we performed these tests in separate statements or in nested if or if...else statements.

C++ provides logical operators that are used to form more complex conditions by combining simple conditions. The logical operators are && (logical AND), || (logical OR) and ! (logical NOT, also called logical negation).

### Logical AND (&&) Operator

Suppose that we wish to ensure that two conditions are *both* true before we choose a certain path of execution. In this case, we can use the && (logical AND) operator, as follows:

```
if ( gender == 1 && age >= 65 )
   seniorFemales++;
```

This if statement contains two simple conditions. The condition gender == 1 is used here to determine whether a person is a female. The condition age >= 65 determines whether a person is a senior citizen. The simple condition to the left of the && operator evaluates first. If necessary, the simple condition to the right of the && operator evaluates next. As we'll discuss shortly, the right side of a logical AND expression is evaluated only if the left side is true. The if statement then considers the combined condition

```
gender == 1 && age >= 65
```

This condition is true if and only if both of the simple conditions are true. Finally, if this combined condition is indeed true, the statement in the if statement's body increments the count of seniorFemales. If either of the simple conditions is false (or both are), then the program skips the incrementing and proceeds to the statement following the if . The preceding combined condition can be made more readable by adding redundant parentheses:

```
( gender == 1 ) && ( age >= 65 )
```

Common Programming Error 5.13

*Although 3 < x < 7 is a mathematically correct condition, it does not evaluate as you might expect in C++. Use ( 3 < x && x < 7 ) to get the proper evaluation in C++.*

Figure 5.15 summarizes the && operator. The table shows all four possible combinations of false and true values for *expression1* and *expression2*. Such tables are often called truth tables. C++ evaluates to false or true all expressions that include relational operators, equality operators and/or logical operators.

Fig. 5.15. && (logical AND) operator truth table.

| expression1 | expression2 | expression1 && expression2 |
|---|---|---|
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

### Logical OR (||) Operator

Now let us consider the || (logical OR) operator. Suppose we wish to ensure at some point in a program that either *or* both of two conditions are true before we choose a certain path of execution. In this case, we use the || operator, as in the following program segment:

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )
   cout << "Student grade is A" << endl;
```

This preceding condition also contains two simple conditions. The simple condition semesterAverage >= 90 evaluates to determine whether the student deserves an "A" in the course because of a solid performance throughout the semester. The simple condition finalExam >= 90 evaluates to determine whether the student deserves an "A" in the course because of an outstanding performance on the final exam. The if statement then considers the combined condition

```
( semesterAverage >= 90 ) || ( finalExam >= 90 )
```

and awards the student an "A" if either or both of the simple conditions are true. Note that the message "Student grade is A" prints unless both of the simple conditions are false. Figure 5.16 is a truth table for the logical OR operator (||).

Fig. 5.16. || (logical OR) operator truth table.

| expression1 | expression2 | expression1 \|\| expression2 |
|---|---|---|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

The && operator has a higher precedence than the || operator. Both operators associate from left to right. An expression containing && or || operators evaluates only until the truth or falsehood of the expression is known. Thus, evaluation of the expression

```
( gender == 1 ) && ( age >= 65 )
```

stops immediately if gender is not equal to 1 (i.e., the entire expression is false) and continues if gender is equal to 1 (i.e., the entire expression could still be true if the condition age >= 65 is true). This performance feature for the evaluation of logical AND and logical OR expressions is called short-circuit evaluation.

Performance Tip 5.6

*In expressions using operator &&, if the separate conditions are independent of one another, make the condition most likely to be false the leftmost condition. In expressions using operator ||, make the condition most likely to be true the leftmost condition. This use of short-circuit evaluation can reduce a program's execution time.*

## Logical Negation (!) Operator

C++ provides the ! (logical NOT, also called logical negation) operator to enable a programmer to "reverse" the meaning of a condition. Unlike the && and || binary operators, which combine two conditions, the unary logical negation operator has only a single condition as an operand. The unary logical negation operator is placed before a condition when we are interested in choosing a path of execution if the original condition (without the logical negation operator) is false, such as in the following program segment:

```
if ( !( grade == sentinelValue ) )
   cout << "The next grade is " << grade << endl;
```

The parentheses around the condition grade == sentinelValue are needed because the logical negation operator has a higher precedence than the equality operator.

In most cases, you can avoid using logical negation by expressing the condition with an appropriate relational or equality operator. For example, the preceding if statement also can be written as follows:

```
if ( grade != sentinelValue )
   cout << "The next grade is " << grade << endl;
```

This flexibility often can help a programmer express a condition in a more "natural" or convenient manner. Figure 5.17 is a truth table for the logical negation operator (!).

**Fig. 5.17. ! (logical negation) operator truth table.**

| expression | !expression |
|---|---|
| **false** | **true** |
| **true** | **false** |

## Logical Operators Example

Figure 5.18 demonstrates the logical operators by producing their truth tables. The output shows each expression that is evaluated and its bool result. By default, bool values true and false are displayed by cout and the stream insertion operator as 1 and 0, respectively. We use stream manipulator boolalpha (a sticky manipulator) in line 11 to specify that the value of each bool expression should be displayed as either the word "true" or the word "false." For example, the result of the expression false && false in line 12 is false, so the second line of output includes the word "false." Lines 11–15 produce the truth table for &&. Lines 18–22 produce the truth table for ||. Lines 25–27 produce the truth table for !.

**Fig. 5.18. Logical operators.**

```
// Fig. 5.18: fig05_18.cpp
// Logical operators.
#include <iostream>
using std::cout;
using std::endl;
using std::boolalpha; // causes bool values to print as "true" or "false"

int main()
{
   // create truth table for && (logical AND) operator
   cout << boolalpha << "Logical AND (&&)"
      << "\nfalse && false: "   << ( false && false )
      << "\nfalse && true: "   << ( false && true )
      << "\ntrue && false: "   << ( true && false )
      << "\ntrue && true: "   << ( true && true ) << "\n\n" ;

   // create truth table for || (logical OR) operator
   cout << "Logical OR ( ||)"
      << "\nfalse || false: "   << ( false || false )
      << "\nfalse || true: "   << ( false || true )
      << "\ntrue || false: "   << ( true || false )
      << "\ntrue || true: "   << ( true || true ) << "\n\n" ;

   // create truth table for ! (logical negation) operator
   cout << "Logical NOT ( !)"
      << "\n!false: "   << ( !false )
      << "\n!true: "   << ( !true ) << endl;
   return 0; // indicate successful termination
} // end main
```

```
Logical AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Logical OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Logical NOT (!)
!false: true
!true: false
```

## Summary of Operator Precedence and Associativity

Figure 5.19 adds the logical operators to the operator precedence and associativity chart. The operators are shown from top to bottom, in decreasing order of precedence.

Fig. 5.19. Operator precedence and associativity.

| Operators | Associativity | Type |
|---|---|---|
| :: | left to right | scope resolution |
| () | left to right | parentheses |
| ++ -- static_cast< type >() | left to right | unary (postfix) |
| ++ -- + - ! | right to left | unary (prefix) |
| * / % | left to right | multiplicative |
| + - | left to right | additive |
| << >> | left to right | insertion/extraction |
| < <= > >= | left to right | relational |
| == != | left to right | equality |
| && | left to right | logical AND |
| \|\| | left to right | logical OR |
| ?: | right to left | conditional |
| = += -= *= /= %= | right to left | assignment |
| , | left to right | comma |

## 5.9. Confusing the Equality (==) and Assignment (=) Operators

There is one type of error that C++ programmers, no matter how experienced, tend to make so frequently that we feel it requires a separate section. That error is accidentally swapping the operators == (equality) and = (assignment). What makes these swaps so damaging is the fact that they ordinarily do not cause syntax errors. Rather, statements with these errors tend to compile correctly and the programs run to completion, often generating incorrect results through runtime logic errors. [*Note:* Some compilers issue a warning when = is used in a context where == normally is expected.]

Two aspects of C++ contribute to these problems. One is that any expression that produces a value can be used in the decision portion of any control statement. If the value of the expression is zero, it is treated as false, and if the value is nonzero, it is treated as true. The second is that assignments produce a value—namely, the value assigned to the variable on the left side of the assignment operator. For example, suppose we intend to write

```
if ( payCode == 4 )
   cout << "You get a bonus!" << endl;
```

but we accidentally write

```
if ( payCode = 4 )
   cout << "You get a bonus!" << endl;
```

The first if statement properly awards a bonus to the person whose payCode is equal to 4. The second if statement—the one with the error—evaluates the assignment expression in the if condition to the constant 4. Any nonzero value is interpreted as true, so the condition in this if statement is always true and the person always receives a bonus regardless of what the actual paycode is! Even worse, the paycode has been modified when it was only supposed to be examined!

Common Programming Error 5.14

*Using operator == for assignment and using operator = for equality are logic errors.*

Error-Prevention Tip 5.3

*Programmers normally write conditions such as x == 7 with the variable name on the left and the constant on the right. By reversing these so that the constant is on the left and the variable name is on the right, as in 7 == x, you'll be protected by the compiler if you accidentally replace the == operator with = . The compiler treats this as a compilation error, because you can't change the value of a constant. This will prevent the potential devastation of a runtime logic error.*

Variable names are said to be lvalues (for "left values") because they can be used on the left side of an assignment operator. Constants are said to be rvalues (for "right values") because they can be used on only the right side of an assignment operator. Note that lvalues can also be used as *rvalues*, but not vice versa.

There is another equally unpleasant situation. Suppose you want to assign a value to a variable with a simple statement like

```
x = 1;
```

but instead write

```
x == 1;
```

Here, too, this is not a syntax error. Rather, the compiler simply evaluates the conditional expression. If `x` is equal to `1`, the condition is `true` and the expression evaluates to the value `true`. If `x` is not equal to `1`, the condition is `false` and the expression evaluates to the value `false`. Regardless of the expression's value, there is no assignment operator, so the value simply is lost. The value of `x` remains unaltered, probably causing an execution-time logic error. Unfortunately, we do not have a handy trick available to help you with this problem!

Error-Prevention Tip 5.4

*Use your text editor to search for all occurrences of* `=` *in your program and check that you have the correct assignment operator or logical operator in each place.*

## 5.10. (Optional) Software Engineering Case Study: Identifying Objects' States and Activities in the ATM System

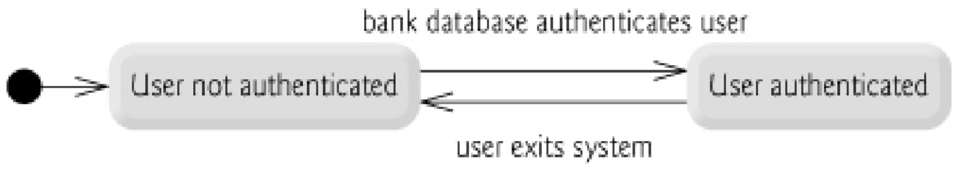
In Section 4.11, we identified many of the class attributes needed to implement the ATM system and added them to the class diagram in Fig. 4.20. In this section, we show how these attributes represent an object's state. We identify some key states that our objects may occupy and discuss how objects change state in response to various events occurring in the system. We also discuss the workflow, or activities, that objects perform in the ATM system. We present the activities of BalanceInquiry and Withdrawal transaction objects in this section, as they represent two of the key activities in the ATM system.

### State Machine Diagrams

Each object in a system goes through a series of discrete states. An object's current state is indicated by the values of the object's attributes at a given time. State machine diagrams (commonly called state diagrams) model key states of an object and show under what circumstances the object changes state. Unlike the class diagrams presented in earlier case study sections, which focused primarily on the structure of the system, state diagrams model some of the behavior of the system.

Figure 5.20 is a simple state diagram that models some of the states of an object of class ATM. The UML represents each state in a state diagram as a rounded rectangle with the name of the state placed inside it. A solid circle with an attached stick arrowhead designates the initial state. Recall that we modeled this state information as the Boolean attribute userAuthenticated in the class diagram of Fig. 4.20. This attribute is initialized to false, or the "User not authenticated" state, according to the state diagram.

**Fig. 5.20. State diagram for the ATM object.**



The arrows with stick arrowheads indicate transitions between states. An object can transition from one state to another in response to various events that occur in the system. The name or description of the event that causes a transition is written near the line that corresponds to the transition. For example, the ATM object changes from the "User not authenticated" state to the "User authenticated" state after the database authenticates the user. Recall from the requirements specification that the database authenticates a user by comparing the account number and PIN entered by the user with those of the corresponding account in the database. If the database indicates that the user has entered a valid account number and the correct PIN, the ATM object transitions to the "User authenticated" state and changes its userAuthenticated attribute to a value of true. When the user exits the system by choosing the "exit" option from the main menu, the ATM object returns to the "User not authenticated" state in preparation for the next ATM user.

Software Engineering Observation 5.2

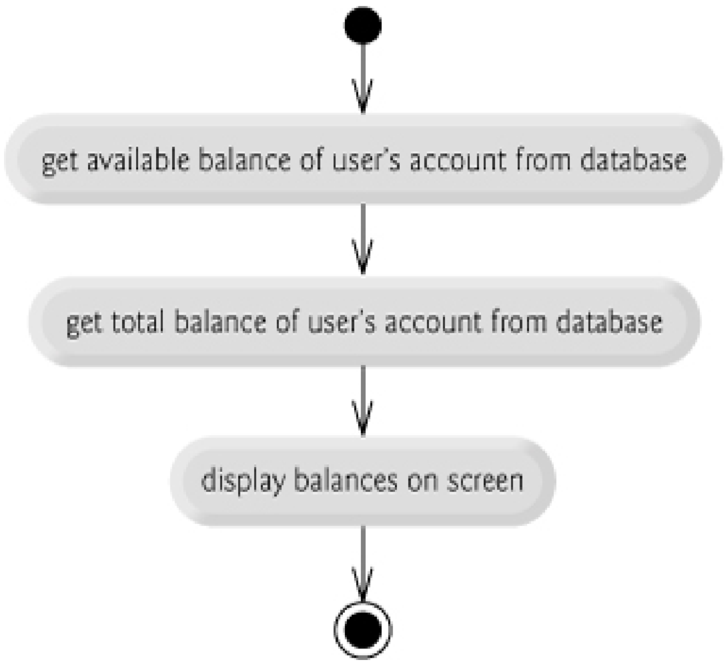
*Software designers do not generally create state diagrams showing every possible state and state transition for all attributes—there are simply too many of them. State diagrams typically show only the most important or complex states and state transitions.*

### Activity Diagrams

Like a state diagram, an activity diagram models aspects of system behavior. Unlike a state diagram, an activity diagram models an object's workflow (sequence of events) during program execution. An activity diagram models the actions the object will perform and in what order. Recall that we used UML activity diagrams to illustrate the flow of control for the control statements presented in Chapter 4 and this chapter.

The activity diagram in Fig. 5.21 models the actions involved in executing a BalanceInquiry transaction. We assume that a BalanceInquiry object has already been initialized and assigned a valid account number (that of the current user), so the object knows which balance to retrieve. The diagram includes the actions that occur after the user selects a balance inquiry from the main menu and before the ATM returns the user to the main menu—a BalanceInquiry object does not perform or initiate these actions, so we do not model them here. The diagram begins with retrieving the available balance of the user's account from the database. Next, the BalanceInquiry retrieves the total balance of the account. Finally, the transaction displays the balances on the screen. This action completes the execution of the transaction.

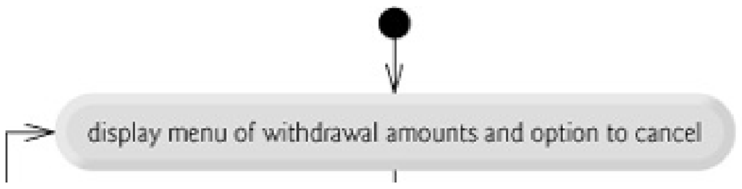
**Fig. 5.21. Activity diagram for a BalanceInquiry transaction.**

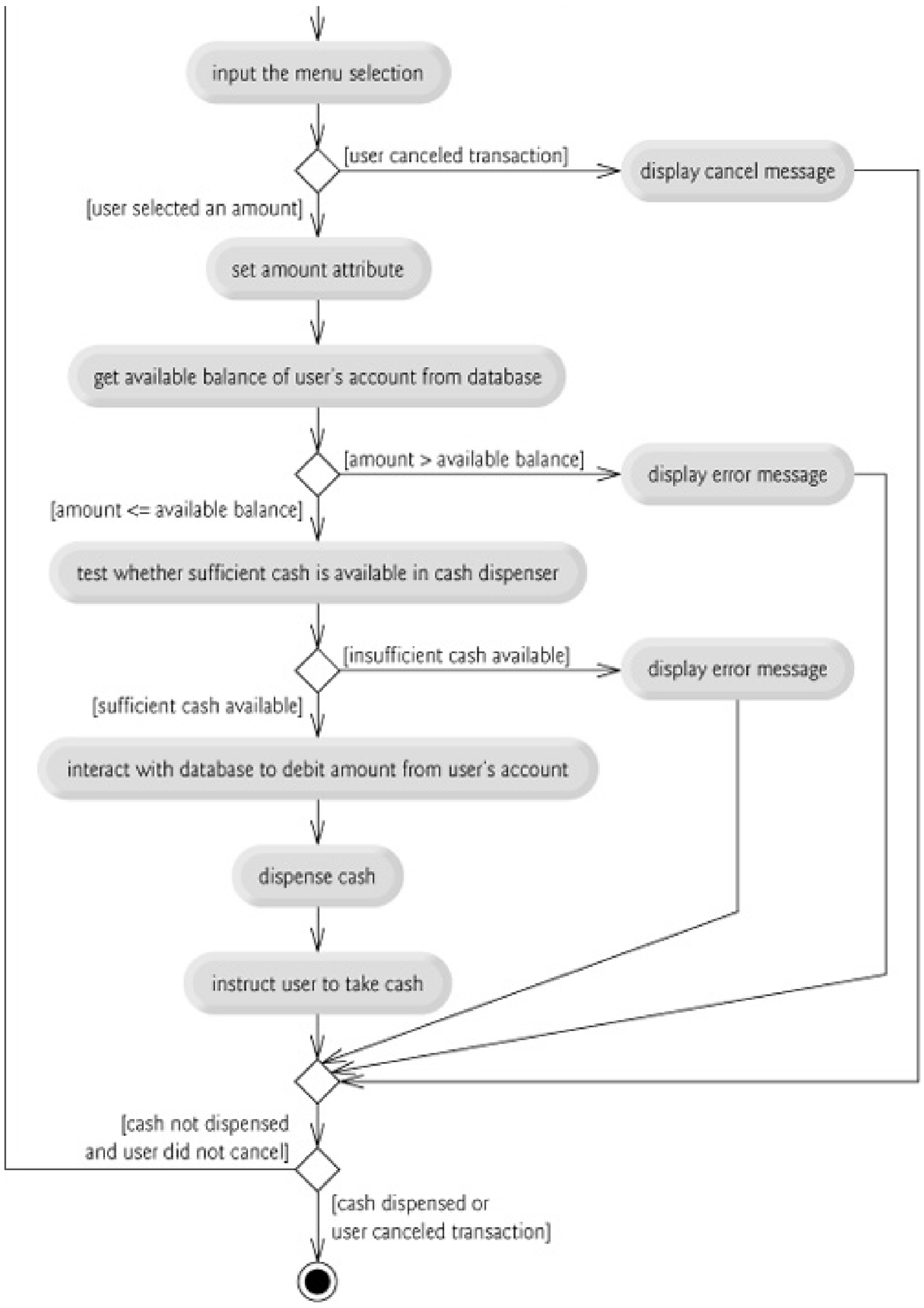

get available balance of user's account from database
get total balance of user's account from database
display balances on screen

The UML represents an action in an activity diagram as an action state modeled by a rectangle with its left and right sides replaced by arcs curving outward. Each action state contains an action expression—for example, "get available balance of user's account from database"—that specifies an action to be performed. An arrow with a stick arrowhead connects two action states, indicating the order in which the actions represented by the action states occur. The solid circle (at the top of Fig. 5.21 ) represents the activity's initial state—the beginning of the workflow before the object performs the modeled actions. In this case, the transaction first executes the "get available balance of user's account from database" action expression. Second, the transaction retrieves the total balance. Finally, the transaction displays both balances on the screen. The solid circle enclosed in an open circle (at the bottom of Fig. 5.21 ) represents the final state—the end of the workflow after the object performs the modeled actions.

Figure 5.22 shows an activity diagram for a Withdrawal transaction. We assume that a Withdrawal object has been assigned a valid account number. We do not model the user selecting a withdrawal from the main menu or the ATM returning the user to the main menu because these are not actions performed by a Withdrawal object. The transaction first displays a menu of standard withdrawal amounts (Fig. 2.13) and an option to cancel the transaction. The transaction then inputs a menu selection from the user. The activity flow now arrives at a decision symbol. This point determines the next action based on the associated guard conditions. If the user cancels the transaction, the system displays an appropriate message. Next, the cancellation flow reaches a merge symbol, where this activity flow joins the transaction's other possible activity flows (which we discuss shortly). Note that a merge can have any number of incoming transition arrows, but only one outgoing transition arrow. The decision at the bottom of the diagram determines whether the transaction should repeat from the beginning. When the user has canceled the transaction, the guard condition "cash dispensed or user canceled transaction" is true, so control transitions to the activity's final state.

**Fig. 5.22. Activity diagram for a Withdrawal transaction.**




input the menu selection
[user canceled transaction]
display cancel message
[user selected an amount]
set amount attribute
get available balance of user's account from database
[amount > available balance]
display error message
[amount <= available balance]
test whether sufficient cash is available in cash dispenser
[insufficient cash available]
display error message
[sufficient cash available]
interact with database to debit amount from user's account
dispense cash
instruct user to take cash
[cash not dispensed and user did not cancel]
[cash dispensed or user canceled transaction]

If the user selects a withdrawal amount from the menu, the transaction sets amount (an attribute of class Withdrawal originally modeled in Fig. 4.20) to the value chosen by the user. The transaction next gets the available balance of the user's account (i.e., the availableBalance attribute of the user's Account object) from the database. The activity flow then arrives at another decision. If the requested withdrawal amount exceeds the user's available balance, the system displays an appropriate error message informing the user of the problem. Control then merges with the other activity flows before reaching the decision at the bottom of the diagram. The guard decision "cash not dispensed and user did not cancel" is true, so the activity flow returns to the top of the diagram, and the transaction prompts the user to input a new amount.

If the requested withdrawal amount is less than or equal to the user's available balance, the transaction tests whether the cash dispenser has enough cash to satisfy the withdrawal request. If it does not, the transaction displays an appropriate error message and passes through the merge before reaching the final decision. Cash was not dispensed, so the activity flow returns to the beginning of the activity diagram, and the transaction prompts the user to choose a new amount. If sufficient cash is available, the transaction interacts with the database to debit the withdrawal amount from the user's account (i.e., subtract the amount from both the availableBalance and totalBalance attributes of the user's Account object). The transaction then dispenses the desired amount of cash and instructs the user to take the cash that is dispensed. The main flow of activity next merges with the two error flows and the cancellation flow. In this case, cash was dispensed, so the activity flow reaches the final state.

We've taken the first steps in modeling the behavior of the ATM system and have shown how an object's attributes participate in the object's activities. In Section 6.22 , we investigate the operations of our classes to create a more complete model of the system's behavior.

**Software Engineering Case Study Self-Review Exercises**

**5.1** State whether the following statement is *true* or *false*, and if *false*, explain why: State diagrams model structural aspects of a system.

**5.2** An activity diagram models the_________ that an object performs and the order in which it performs them.

**a.** actions

**b.** attributes

**c.** states

**d.** state transitions

**5.3** Based on the requirements specification, create an activity diagram for a deposit transaction.

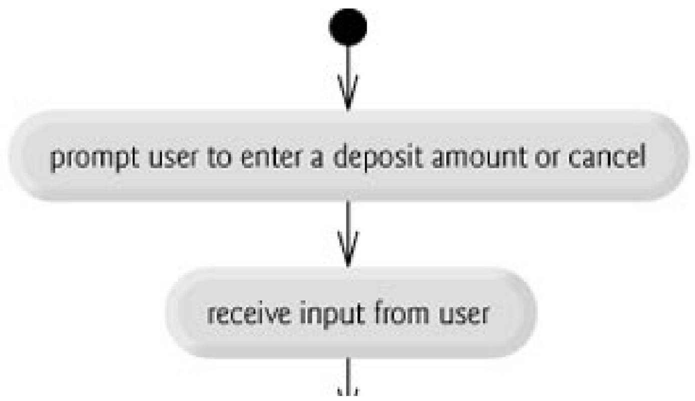
**Answers to Software Engineering Case Study Self-Review Exercises**

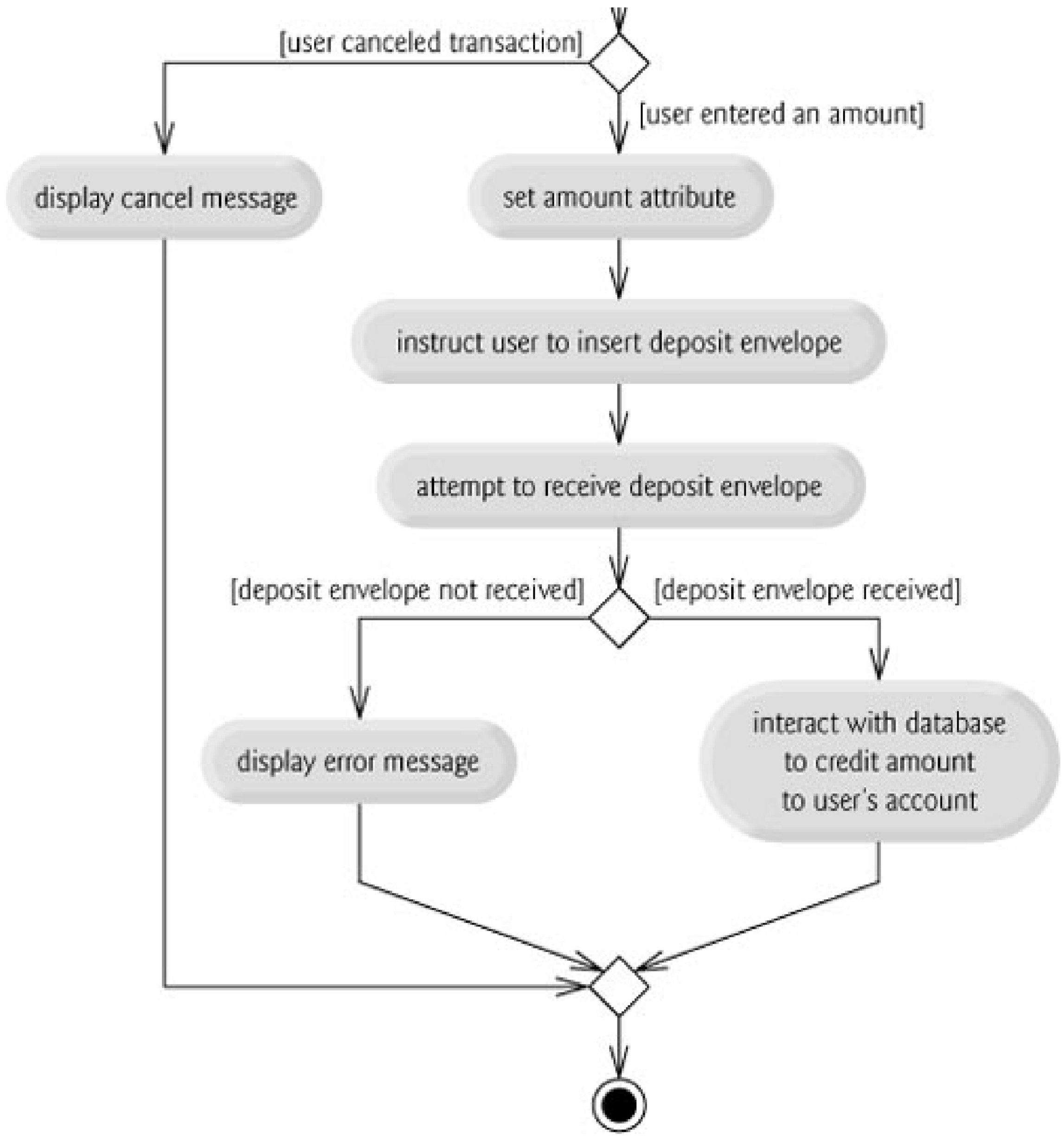
**5.1** False. State diagrams model some of the behavior of a system.

**5.2** a.

**5.3** Figure 5.23 presents an activity diagram for a deposit transaction. The diagram models the actions that occur after the user chooses the deposit option from the main menu and before the ATM returns the user to the main menu. Recall that part of receiving a deposit amount from the user involves converting an integer number of cents to a dollar amount. Also recall that crediting a deposit amount to an account involves increasing only the totalBalance attribute of the user's Account object. The bank updates the availableBalance attribute of the user's Account object only after confirming the amount of cash in the deposit envelope and after the enclosed checks clear—this occurs independently of the ATM system.

**Fig. 5.23. Activity diagram for a Deposit transaction.**




[user canceled transaction]
[user entered an amount]
display cancel message
set amount attribute
instruct user to insert deposit envelope
attempt to receive deposit envelope
[deposit envelope not received]
[deposit envelope received]
display error message
interact with database to credit amount to user's account

## 5.11. Wrap-Up

In this chapter, we completed our introduction to C++'s control statements, which enable you to control the flow of execution in functions. Chapter 4 discussed the if, if...else and while statements. The current chapter demonstrated C++'s remaining control statements—for, do...while and switch. We have shown that any algorithm can be developed using combinations of the sequence structure (i.e., statements listed in the order in which they should execute), the three types of selection statements—if, if...else and switch—and the three types of repetition statements—while, do...while and for. In this chapter and Chapter 4, we have discussed how you can combine these building blocks to utilize proven program construction and problem-solving techniques. This chapter also introduced C++'s logical operators, which enable you to use more complex conditional expressions in control statements. Finally, we examined the common errors of confusing the equality and assignment operators and provided suggestions for avoiding these errors.

In Chapter 3, we introduced C++ programming with the basic concepts of classes, objects and member functions. Chapter 4 and this chapter provided a thorough introduction to the control statements that you typically use to specify program logic in functions. In Chapter 6, we examine functions in greater depth.

# 6. Functions and an Introduction to Recursion

**Objectives**

In this chapter you'll learn:

- To construct programs modularly from functions.
- To use common math functions available in the C++ Standard Library.
- To create functions with multiple parameters.
- The mechanisms for passing information between functions and returning results.
- How the function call/return mechanism is supported by the function call stack and activation records.
- To use random number generation to implement game-playing applications.
- How the visibility of identifiers is limited to specific regions of programs.
- To write and use recursive functions, i.e., functions that call themselves.

Form ever follows function.

—***Louis Henri Sullivan***

E pluribus unum. (One composed of many.)

—***Virgil***

O! call back yesterday, bid time return.

—***William Shakespeare***

Call me Ishmael.

—***Herman Melville***

When you call me that, smile!

—***Owen Wister***

Answer me in one word.

—***William Shakespeare***

There is a point at which methods devour themselves.

—***Frantz Fanon***

Life can only be understood backwards; but it must be lived forwards.

—***Soren Kierkegaard***

**Outline**

## 6.1. Introduction

In this chapter, we study functions in more depth. We emphasize how to declare and use functions to facilitate the design, implementation, operation and maintenance of large programs.

We'll overview a portion of the C++ Standard Library's math functions, showing several that require more than one parameter. Next, you'll see how to declare a function with more than one parameter. We'll also present additional information about function prototypes and how the compiler uses them to convert the type of an argument in a function call to the type specified in a function's parameter list, if necessary.

Next, we'll take a brief diversion into simulation techniques with random number generation and develop a version of the casino dice game called craps that uses most of the C++ capabilities you have learned to this point in the book.

We then present C++'s storage classes and scope rules. These determine the period during which an object exists in memory and where its identifier can be referenced in a program. You'll also see how C++ is able to keep track of which function is currently executing, how parameters and other local variables of functions are maintained in memory and how a function knows where to return after it completes execution. We discuss two topics that help improve program performance—inline functions that can eliminate the overhead of a function call and reference parameters that can be used to pass large data items to functions efficiently.

Many of the applications you develop will have more than one function of the same name. This technique, called function overloading, is used by programmers to implement functions that perform similar tasks for arguments of different types or possibly for different numbers of arguments. We consider function templates—a mechanism for defining a family of overloaded functions. The chapter concludes with a discussion of functions that call themselves, either directly, or indirectly (through another function)—a topic called recursion.

## 6.2. Program Components in C++

C++ programs are typically written by combining new functions and classes you write with "prepackaged" functions and classes available in the C++ Standard Library. In this chapter, we concentrate on functions.

The C++ Standard Library provides a rich collection of functions for performing common mathematical calculations, string manipulations, character manipulations, input/output, error checking and many other useful operations. This makes your job easier, because these functions provide many of the capabilities programmers need. The C++ Standard Library functions are provided as part of the C++ programming environment.

Software Engineering Observation 6.1

*Read the documentation for your compiler to familiarize yourself with the functions and classes in the C++ Standard Library*.

## 6.3. Math Library Functions

As you know, a class can provide member functions that perform the services of the class. For example, in Chapters 3–5 , you have called the member functions of various versions of a GradeBook object to display the GradeBook 's welcome message, to set its course name, to obtain a set of grades and to calculate the average of those grades.

Sometimes functions are not members of a class. Such functions are called global functions . Like a class's member functions, the function prototypes for global functions are placed in header files, so that the global functions can be reused in any program that includes the header file and that can link to the function's object code. For example, recall that we used function pow of the <cmath> header file to raise a value to a power in Figure 5.6 . We introduce various functions from the <cmath> header file here to present the concept of global functions that do not belong to a particular class. In this chapter and in subsequent chapters, we use a combination of global functions (such as main ) and classes with member functions to implement our example programs.

The <cmath> header file provides a collection of functions that enable you to perform common mathematical calculations. For example, you can calculate the square root of 900.0 with the function call

```
sqrt( 900.0 )
```

The preceding expression evaluates to 30.0. Function sqrt takes an argument of type double and returns a double result. Note that there is no need to create any objects before calling function sqrt. Also note that *all* functions in the <cmath> header file are global functions—therefore, each is called simply by specifying the name of the function followed by parentheses containing the function's arguments.

Function arguments may be constants, variables or more complex expressions. If c = 13.0, d = 3.0 and f = 4.0, then the statement

```
cout << sqrt( c + d * f ) << endl;
```

calculates and prints the square root of 13.0 + 3.0 * 4.0 = 25.0—namely, 5.0. Some math library functions are summarized in Fig. 6.1 (variables x and y are of type double).

**Fig. 6.1. Math library functions.**

| **Function** | **Description** | **Example** |
|---|---|---|
| **ceil( x )** | rounds *x* to the smallest | ceil( ***9.2*** ) is 10.0 |
| | integer not less than *x* | *ceil(* ***-9.8*** ) is -9.0 |
| cos( x ) | trigonometric cosine of *x* (*x* in radians) | cos( ***0.0*** ) is 1.0 |
| exp( x ) | exponential function $e^x$ | *exp(* ***1.0*** ) is 2.71828 |
| | | exp( ***2.0*** ) is 7.38906 |
| fabs( x ) | absolute value of *x* | *fabs(* ***5.1*** ) is 5.1 |
| | | fabs( ***0.0*** ) is 0.0 |
| | | fabs( ***-8.76*** ) is 8.76 |

| **Function** | **Description** | **Example** |
|---|---|---|
| **floor( x )** | rounds *x* to the largest integer not greater than *x* | *floor(* ***9.2*** *)* is 9.0 |
| | | floor( ***-9.8*** ) is -10.0 |
| fmod( x, y ) | remainder of *x/y* as a floating-point number | fmod( ***2.6***, ***1.2*** ) is 0.2 |
| log( x ) | natural logarithm of *x* (base *e*) | log( ***2.718282*** ) is 1.0 |
| | | log( ***7.389056*** ) is 2.0 |
| log10( x ) | logarithm of *x* (base 10) | log10( ***10.0*** ) is 1.0 |
| | | log10( ***100.0*** ) is 2.0 |
| pow( x, y ) | *x* raised to power *y* ($x^y$) | pow( ***2***, ***7*** ) is 128 |
| | | pow( ***9***, ***.5*** ) is 3 |
| sin( x ) | trigonometric sine of *x* (*x* in radians) | sin( ***0.0*** ) is 0 |
| sqrt( x ) | square root of *x* (where *x* is a nonnegative value) | sqrt( ***9.0*** ) is 3.0 |
| tan( x ) | trigonometric tangent of *x* (*x* in radians) | tan( ***0.0*** ) is 0 |

### 6.4. Function Definitions with Multiple Parameters

The program in Figs. 6.2–6.4 modifies our GradeBook class by including a user-defined function called maximum that determines and returns the largest of three int values. When the application begins execution, the main function (lines 5–14 of Fig. 6.4 ) creates one object of class GradeBook (line 8) and calls the object's inputGrades member function (line 11) to read three integer grades from the user. In class GradeBook's implementation file (Fig. 6.3), lines 54–55 of member function inputGrades prompt the user to enter three integer values and read them from the user. Line 58 calls member function maximum (defined in lines 62–75). Function maximum determines the largest value, then the return statement (line 74) returns that value to the point at which function inputGrades invoked maximum (line 58). Member function inputGrades then stores maximum's return value in data member maximumGrade. This value is then output by calling function displayGradeReport (line 12 of Fig. 6.4). [*Note:* We named this function displayGradeReport because subsequent versions of class GradeBook will use this function to display a complete grade report, including the maximum and minimum grades.] In Chapter 7 , Arrays and Vectors, we'll enhance the GradeBook to process an arbitrary number of grades.

**Fig. 6.2. GradeBook header file.**

```
// Fig. 6.2: GradeBook.h
// Definition of class GradeBook that finds the maximum of three grades.
// Member functions are defined in GradeBook.cpp
#include <string> // program uses C++ standard string class
using std::string;

// GradeBook class definition
class GradeBook
{
public:
   GradeBook( string ); // constructor initializes course name
   void setCourseName( string ); // function to set the course name
   string getCourseName(); // function to retrieve the course name
   void displayMessage(); // display a welcome message
   void inputGrades(); // input three grades from user
   void displayGradeReport(); // display a report based on the grades
   int maximum( int, int, int ); // determine max of 3 values
private:
   string courseName; // course name for this GradeBook
   int maximumGrade; // maximum of three grades
}; // end class GradeBook
```

**Fig. 6.3. GradeBook class defines function maximum.**

```
// Fig. 6.3: GradeBook.cpp
// Member-function definitions for class GradeBook that
// determines the maximum of three grades.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "GradeBook.h" // include definition of class GradeBook

// constructor initializes courseName with string supplied as argument;
// initializes maximumGrade to 0
GradeBook::GradeBook( string name )
{
   setCourseName( name ); // validate and store courseName
   maximumGrade = 0; // this value will be replaced by the maximum grade
} // end GradeBook constructor

// function to set the course name; limits name to 25 or fewer characters
void GradeBook::setCourseName( string name )
{
   if ( name.length() <= 25 ) // if name has 25 or fewer characters
      courseName = name; // store the course name in the object
   else // if name is longer than 25 characters
   { // set courseName to first 25 characters of parameter name
      courseName = name.substr( 0, 25 ); // select first 25 characters
      cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
         << "Limiting courseName to first 25 characters.\n" << endl;
   } // end if...else
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName()
{
   return courseName;
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   // this statement calls getCourseName to get the
   // name of the course this GradeBook represents
   cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
      << endl;
} // end function displayMessage

// input three grades from user; determine maximum
void GradeBook::inputGrades()
{
   int grade1; // first grade entered by user
   int grade2; // second grade entered by user
   int grade3; // third grade entered by user

   cout << "Enter three integer grades: "    ;
   cin >> grade1 >> grade2 >> grade3;

   // store maximum in member maximumGrade
   maximumGrade = maximum( grade1, grade2, grade3 );
} // end function inputGrades

// returns the maximum of its three integer parameters
int GradeBook::maximum( int x, int y, int z )
{
   int maximumValue = x; // assume x is the largest to start

   // determine whether y is greater than maximumValue
   if ( y > maximumValue )
      maximumValue = y; // make y the new maximumValue

   // determine whether z is greater than maximumValue
   if ( z > maximumValue )
      maximumValue = z; // make z the new maximumValue

   return maximumValue;
} // end function maximum

// display a report based on the grades entered by user
void GradeBook::displayGradeReport()
{
   // output maximum of grades entered
   cout << "Maximum of grades entered: "     << maximumGrade << endl;
} // end function displayGradeReport
```

**Fig. 6.4. Demonstrating function maximum.**

```
// Fig. 6.4: fig06_04.cpp
// Create GradeBook object, input grades and display grade report.
#include "GradeBook.h" // include definition of class GradeBook

int main()
{
   // create GradeBook object
   GradeBook myGradeBook( "CS101 C++ Programming" );

   myGradeBook.displayMessage(); // display welcome message
   myGradeBook.inputGrades(); // read grades from user
   myGradeBook.displayGradeReport(); // display report based on grades
   return 0; // indicate successful termination
} // end main
```

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

Enter three integer grades: 86 67 75
Maximum of grades entered: 86
```

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

Enter three integer grades: 67 86 75
Maximum of grades entered: 86
```

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

Enter three integer grades: 67 75 86
Maximum of grades entered: 86
```

Software Engineering Observation 6.2

*The commas used in line 58 of Fig. 6.3 to separate the arguments to function maximum are not comma operators as discussed in Section 5.3. The comma operator guarantees that its operands are evaluated left to right. The order of evaluation of a function's arguments, however, is not specified by the C++ standard. Thus, different compilers can evaluate function arguments in different orders. The C++ standard does require that all arguments in a function call be evaluated before the called function executes.*

Portability Tip 6.1

*Sometimes when a function's arguments are more involved expressions, such as those with calls to other functions, the order in which the compiler evaluates the arguments could affect the values of one or more of the arguments. If the evaluation order changes between compilers, the argument values passed to the function could vary, causing subtle logic errors.*

Error-Prevention Tip 6.1

*If you have doubts about the order of evaluation of a function's arguments and whether the order would affect the values passed to the function, evaluate the arguments in separate assignment statements before the function call, assign the result of each expression to a local variable, then pass those variables as arguments to the function.*

The prototype of member function `maximum` (Fig. 6.2, line 17) indicates that the function returns an integer value, that the function's name is `maximum` and that the function requires three integer parameters to accomplish its task. Function `maximum`'s header (Fig. 6.3, line 62) matches the function prototype and indicates that the parameter names are `x`, `y` and `z`. When `maximum` is called (Fig. 6.3, line 58), the parameter `x` is initialized with the value of the argument `grade1`, the parameter `y` is initialized with the value of the argument `grade2` and the parameter `z` is initialized with the value of the argument `grade3`. There must be one argument in the function call for each parameter (also called a formal parameter) in the function definition.

Notice that multiple parameters are specified in both the function prototype and the function header as a comma-separated list. The compiler refers to the function prototype to check that calls to `maximum` contain the correct number and types of arguments and that the types of the arguments are in the correct order. In addition, the compiler uses the prototype to ensure that the value returned by the function can be used correctly in the expression that called the function (e.g., a function call that returns `void` cannot be used as the right side of an assignment statement). Each argument must be consistent with the type of the corresponding parameter. For example, a parameter of type `double` can receive values like 7.35, 22 or –0.03456, but not a string like `"hello"`. If the arguments passed to a function do not match the types specified in the function's prototype, the compiler attempts to convert the arguments to those types. Section 6.5 discusses this conversion.

Common Programming Error 6.1

*Declaring method parameters of the same type as `double x, y` instead of `double x, double y` is a syntax error—an explicit type is required for each parameter in the parameter list.*

Common Programming Error 6.2

*Compilation errors occur if the function prototype, function header and function calls do not all agree in the number, type and order of arguments and parameters, and in the return type.*

Software Engineering Observation 6.3

*A function that has many parameters may be performing too many tasks. Consider dividing the function into smaller functions that perform the separate tasks. Limit the function header to one line if possible.*

To determine the maximum value (lines 62–75 of Fig. 6.3), we begin with the assumption that parameter `x` contains the largest value, so line 64 of function `maximum` declares local variable `maximumValue` and initializes it with the value of parameter `x`. Of course, it is possible that parameter `y` or `z` contains the actual largest value, so we must compare each of these values with `maximumValue`. The `if` statement in lines 67–68 determines whether `y` is greater than `maximumValue` and, if so, assigns `y` to `maximumValue`. The `if` statement in lines 71–72 determines whether `z` is greater than `maximumValue` and, if so, assigns `z` to `maximumValue`. At this point the largest of the three values is in `maximumValue`, so line 74 returns that value to the call in line 58. When program control returns to the point in the program where `maximum` was called, `maximum`'s parameters `x`, `y` and `z` are no longer accessible to the program. We'll see why in the next section.

There are three ways to return control to the point at which a function was invoked. If the function does not return a result (i.e., the function has a `void` return type), control returns when the program reaches the function-ending right brace, or by execution of the statement

```
return;
```

If the function does return a result, the statement

```
return expression;
```

evaluates *expression* and returns the value of *expression* to the caller.

## 6.5. Function Prototypes and Argument Coercion

A function prototype (also called a function declaration) tells the compiler the name of a function, the type of data returned by the function, the number of parameters the function expects to receive, the types of those parameters and the order in which the parameters of those types are expected.

Software Engineering Observation 6.4

*Function prototypes are required in C++. Use* `#include` *preprocessor directives to obtain function prototypes for the C++ Standard Library functions from the header files for the appropriate libraries (e.g., the prototype for math function* `sqrt` *is in header file* `<cmath>`*; a partial list of C++ Standard Library header files appears in Section 6.6). Also use* `#include` *to obtain header files containing function prototypes written by you or your group members.*

Common Programming Error 6.3

*If a function is defined before it is invoked, then the function's definition also serves as the function's prototype, so a separate prototype is unnecessary. If a function is invoked before it is defined, and that function does not have a function prototype, a compilation error occurs.*

Software Engineering Observation 6.5

*Always provide function prototypes, even though it is possible to omit them when functions are defined before they are used (in which case the function header acts as the function prototype as well). Providing the prototypes avoids tying the code to the order in which functions are defined (which can easily change as a program evolves).*

### Function Signatures

The portion of a function prototype that includes the name of the function and the types of its arguments is called the function signature or simply the signature. The function signature does not specify the function's return type. Functions in the same scope must have unique signatures. The scope of a function is the region of a program in which the function is known and accessible. We'll say more about scope in Section 6.10.

Common Programming Error 6.4

*It is a compilation error if two functions in the same scope have the same signature but different return types.*

In Fig. 6.2 , if the function prototype in line 17 had been written

```
void maximum( int, int, int );
```

the compiler would report an error, because the void return type in the function prototype would differ from the int return type in the function header. Similarly, such a prototype would cause the statement

```
cout << maximum( 6, 7, 0 );
```

to generate a compilation error, because that statement depends on maximum to return a value to be displayed.

### Argument Coercion

An important feature of function prototypes is argument coercion —i.e., forcing arguments to the appropriate types specified by the parameter declarations. For example, a program can call a function with an integer argument, even though the function prototype specifies a double argument—the function will still work correctly.

### Argument Promotion Rules

Sometimes, argument values that do not correspond precisely to the parameter types in the function prototype can be converted by the compiler to the proper type before the function is called. These conversions occur as specified by C++'s promotion rules . The promotion rules indicate how to convert between types without losing data. An int can be converted to a double without changing its value. However, a double converted to an int truncates the fractional part of the double value. Keep in mind that double variables can hold numbers of much greater magnitude than int variables, so the loss of data may be considerable. Values may also be modified when converting large integer types to small integer types (e.g., long to short), signed to unsigned or unsigned to signed.

The promotion rules apply to expressions containing values of two or more data types; such expressions are also referred to as mixed-type expressions . The type of each value in a mixed-type expression is promoted to the "highest" type in the expression (actually a temporary version of each value is created and used for the expression—the original values remain unchanged). Promotion also occurs when the type of a function argument does not match the parameter type specified in the function definition or prototype. Figure 6.5 lists the fundamental data types in order from "highest type" to "lowest type."

**Fig. 6.5. Promotion hierarchy for fundamental data types.**

| **Data types** | |
|---|---|
| **long double** | |
| double | |
| float | |
| unsigned long int | (synonymous with unsigned long) |
| long int | (synonymous with long) |
| unsigned int | (synonymous with unsigned) |
| int | |
| unsigned short int | (synonymous with unsigned short) |
| short int | (synonymous with short) |
| unsigned char | |
| char | |
| bool | |

Converting values to lower fundamental types can result in incorrect values. Therefore, a value can be converted to a lower fundamental type only by explicitly assigning the value to a variable of lower type (some compilers will issue a warning in this case) or by using a cast operator (see Section 4.7 ). Function argument values are converted to the parameter types in a function prototype as if they were being assigned directly to variables of those types. If a square function that uses an integer parameter is called with a floating-point argument, the argument is converted to int (a lower type), and square could return an incorrect value. For example, square( 4.5 ) returns 16, not 20.25.

Common Programming Error 6.5

*Converting from a higher data type in the promotion hierarchy to a lower type, or between signed and unsigned, can corrupt the data value, causing a loss of information.*

Common Programming Error 6.6

*It is a compilation error if the arguments in a function call do not match the number and types of the parameters declared in the corresponding function prototype. It is also an error if the number of arguments in the call matches, but the arguments cannot be implicitly converted to the expected types.*

## 6.6. C++ Standard Library Header Files

The C++ Standard Library is divided into many portions, each with its own header file. The header files contain the function prototypes for the related functions that form each portion of the library. The header files also contain definitions of various class types and functions, as well as constants needed by those functions. A header file "instructs" the compiler on how to interface with library and user-written components.

Figure 6.6 lists some common C++ Standard Library header files, most of which are discussed later in the book. The term "macro" that is used several times in Fig. 6.6 is discussed in detail in Appendix D, Preprocessor. Header file names ending in .h are "old-style" header files that have been superseded by the C++ Standard Library header files. We use only the C++ Standard Library versions of each header file in this book to ensure that our examples will work on most standard C++ compilers.

**Fig. 6.6. C++ Standard Library header files.**

| C++ Standard Library header file | Explanation |
|---|---|
| **<iostream>** | Contains function prototypes for the C++ standard input and standard output functions, introduced in Chapter 2, and is covered in more detail in Chapter 15, Stream Input/Output. This header file replaces header file <iostream.h>. |
| <iomanip> | Contains function prototypes for stream manipulators that format streams of data. This header file is first used in Section 4.7 and is discussed in more detail in Chapter 15, Stream Input/Output. This header file replaces header file <iomanip.h>. |
| <cmath> | Contains function prototypes for math library functions (discussed in Section 6.3). This header file replaces header file <math.h>. |
| <cstdlib> | Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header file are covered in Section 6.7; Chapter 11, Operator Overloading; String and Array Objects; Chapter 16, Exception Handling; and Chapter 19, Bits, Characters, C Strings and structs. This header file replaces header file <stdlib.h>. |
| <ctime> | Contains function prototypes and types for manipulating the time and date. This header file replaces header file <time.h>. This header file is used in Section 6.7. |
| <vector><br><list><br><deque><br><queue><br><stack><br><map><br><set><br><bitset> | These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The <vector> header is first introduced in Chapter 7, Arrays and Vectors. We discuss all these header files in Chapter 20, Standard Template Library (STL). |
| <cctype> | Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. This header file replaces header file <ctype.h>. These topics are discussed in Chapter 8, Pointers and Pointer-Based Strings, and Chapter 19, Bits, Characters, C Strings and structs. |

| **C++ Standard Library header file** | **Explanation** |
|---|---|
| **<cstring>** | Contains function prototypes for C-style string-processing functions. This header file replaces header file <string.h> . This header file is used inChapter 11, Operator Overloading; String and Array Objects. |
| <typeinfo> | Contains classes for runtime type identification (determining data types at execution time). This header file is discussed in Section 13.8. |
| <exception><br><stdexcept> | These header files contain classes that are used for exception handling (discussed in Chapter 16, Exception Handling). |
| <memory> | Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 16, Exception Handling. |
| <fstream> | Contains function prototypes for functions that perform input from files on disk and output to files on disk (discussed in Chapter 17 , File Processing). This header file replaces header file <fstream.h>. |
| <string> | Contains the definition of classstring from the C++ Standard Library (discussed in Chapter 18, Class string and String Stream Processing). |
| <sstream> | Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 18, Class string and String Stream Processing). |
| <functional> | Contains classes and functions used by C++ Standard Library algorithms. This header file is used in Chapter 20, Standard Template Library (STL). |
| <iterator> | Contains classes for accessing data in the C++ Standard Library containers. This header file is used in Chapter 20. |
| <algorithm> | Contains functions for manipulating data in C++ Standard Library containers. This header file is used in Chapter 20. |
| <cassert> | Contains macros for adding diagnostics that aid program debugging. This replaces header file <assert.h> from pre-standard C++. This header file is used in Appendix D, Preprocessor. |
| <cfloat> | Contains the floating-point size limits of the system. This header file replaces header file <float.h>. |
| <climits> | Contains the integral size limits of the system. This header file replaces header file <limits.h>. |
| <cstdio> | Contains function prototypes for the C-style standard input/output library functions and information used by them. This header file replaces header file <stdio.h>. |
| <locale> | Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.). |
| <limits> | Contains classes for defining the numerical data type limits on each computer platform. |

| **C++ Standard Library header file** | **Explanation** |
|---|---|
| **<utility>** | Contains classes and functions that are used by many C++ Standard Library header files. |

## 6.7. Case Study: Random Number Generation

We now take a brief and hopefully entertaining diversion into a popular programming application, namely simulation and game playing. In this and the next section, we develop a game-playing program that includes multiple functions. The program uses many of the control statements and concepts discussed to this point.

The element of chance can be introduced into computer applications by using the C++ Standard Library function rand.

Consider the following statement:

```
i = rand();
```

The function rand generates an unsigned integer between 0 and RAND_MAX (a symbolic constant defined in the <cstdlib> header file). The value of RAND_MAX must be at least 32767—the maximum positive value for a two-byte (16-bit) integer. For GNU C++, the value of RAND_MAX is 2147483647; for Visual Studio, the value of RAND_MAX is 32767. If rand truly produces integers at random, every number between 0 and RAND_MAX has an equal *chance* (or probability) of being chosen each time rand is called.

The range of values produced directly by the function rand often is different than what a specific application requires. For example, a program that simulates coin tossing might require only 0 for "heads" and 1 for "tails." A program that simulates rolling a six-sided die would require random integers in the range 1 to 6. A program that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon in a video game might require random integers in the range 1 through 4.

### Rolling a Six-Sided Die

To demonstrate rand, let us develop a program (Fig. 6.7) to simulate 20 rolls of a six-sided die and print the value of each roll. The function prototype for the rand function is in <cstdlib>. To produce integers in the range 0 to 5, we use the modulus operator (%) with rand as follows:

```
rand() % 6
```

This is called scaling. The number 6 is called the scaling factor. We then shift the range of numbers produced by adding 1 to our previous result. Figure 6.7 confirms that the results are in the range 1 to 6.

**Fig. 6.7. Shifted, scaled integers produced by `1 + rand() % 6`.**

```
// Fig. 6.7: fig06_07.cpp
// Shifted and scaled random integers.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib> // contains function prototype for rand
using std::rand;

int main()
{
   // loop 20 times
   for ( int counter = 1; counter <= 20 ; counter++ )
   {
      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      // if counter is divisible by 5, start a new line of output
      if ( counter % 5 == 0 )
         cout << endl;
   } // end for

   return 0; // indicates successful termination
} // end main
```

```
         6         6         5         5         6
         5         1         1         5         3
         6         6         2         4         2
         6         2         3         4         1
```

### Rolling a Six-Sided Die 6,000,000 Times

To show that the numbers produced by function `rand` occur with approximately equal likelihood, Fig. 6.8 simulates 6,000,000 rolls of a die. Each integer in the range 1 to 6 should appear approximately 1,000,000 times. This is confirmed by the output window at the end of Fig. 6.8.

**Fig. 6.8. Rolling a six-sided die 6,000,000 times.**

```cpp
// Fig. 6.8: fig06_08.cpp
// Roll a six-sided die 6,000,000 times.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib> // contains function prototype for rand
using std::rand;

int main()
{
   int frequency1 = 0; // count of 1s rolled
   int frequency2 = 0; // count of 2s rolled
   int frequency3 = 0; // count of 3s rolled
   int frequency4 = 0; // count of 4s rolled
   int frequency5 = 0; // count of 5s rolled
   int frequency6 = 0; // count of 6s rolled

   int face; // stores most recently rolled value

   // summarize results of 6,000,000 rolls of a die
   for ( int roll = 1; roll <= 6000000; roll++ )
   {
      face = 1 + rand() % 6; // random number from 1 to 6

      // determine roll value 1-6 and increment appropriate counter
      switch ( face )
      {
         case 1:
            ++frequency1; // increment the 1s counter
            break;
         case 2:
            ++frequency2; // increment the 2s counter
            break;
         case 3:
            ++frequency3; // increment the 3s counter
            break;
         case 4:
            ++frequency4; // increment the 4s counter
            break;
         case 5:
            ++frequency5; // increment the 5s counter
            break;
         case 6:
            ++frequency6; // increment the 6s counter
            break;
         default: // invalid value
            cout << "Program should never get here!";
      } // end switch
   } // end for

   cout << "Face" << setw( 13 ) << "Frequency" << endl; // output headers
   cout << "   1" << setw( 13 ) << frequency1
57       << "\n   2"  << setw( 13 ) << frequency2
58       << "\n   3"  << setw( 13 ) << frequency3
59       << "\n   4"  << setw( 13 ) << frequency4
60       << "\n   5"  << setw( 13 ) << frequency5
61       << "\n   6"  << setw( 13 ) << frequency6 << endl;
62     return 0; // indicates successful termination
63  } // end main
```

```
Face   Frequency
   1      999702
   2     1000823
   3      999378
   4      998898
   5     1000777
   6     1000422
```

As the program output shows, we can simulate the rolling of a six-sided die by scaling and shifting the values produced by `rand`. Note that the program should never get to the `default` case (lines 50–51) provided in the `switch` structure, because the `switch`'s controlling expression (`face`) always has values in the range 1–6; however, we provide the `default` case as a matter of good practice. After we study arrays in Chapter 7, we show how to replace the entire `switch` structure in Fig. 6.8 elegantly with a single-line statement.

Error-Prevention Tip 6.2

*Provide a `default` case in a `switch` to catch errors even if you are absolutely, positively certain that you have no bugs!*

### Randomizing the Random Number Generator

Executing the program of Fig. 6.7 again produces the following output:

```
 6     6     5     5     6
 5     1     1     5     3
 6     6     2     4     2
 6     2     3     4     1
```

Notice that the program prints exactly the same sequence of values shown in Fig. 6.7. How can these be random numbers? Ironically, this repeatability is an important characteristic of function rand . When debugging a simulation program, this repeatability is essential for proving that corrections to the program work properly.

Function rand actually generates pseudorandom numbers. Repeatedly calling rand produces a sequence of numbers that appears to be random. However, the sequence repeats itself each time the program executes. Once a program has been thoroughly debugged, it can be conditioned to produce a different sequence of random numbers for each execution. This is called randomizing and is accomplished with the C++ Standard Library function srand. Function srand takes an unsigned integer argument and seeds the rand function to produce a different sequence of random numbers for each execution of the program.

Figure 6.9 demonstrates function srand . The program uses the data type unsigned , which is short for unsigned int. An int is stored in at least two bytes of memory (typically four bytes of memory on today's popular 32-bit systems) and can have positive and negative values. A variable of type unsigned int is also stored in at least two bytes of memory. A two-byte unsigned int can have only nonnegative values in the range 0–65535. A four-byte unsigned int can have only nonnegative values in the range 0–4294967295. Function srand takes an unsigned int value as an argument. The function prototype for the srand function is in header file <cstdlib>.

**Fig. 6.9. Randomizing the die-rolling program.**

```cpp
// Fig. 6.9: fig06_09.cpp
// Randomizing die-rolling program.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib> // contains prototypes for functions srand and rand
using std::rand;
using std::srand;

int main()
{
   unsigned seed; // stores the seed entered by the user

   cout << "Enter seed: " ;
   cin >> seed;
   srand( seed ); // seed random number generator

   // loop 10 times
   for ( int counter = 1; counter <= 10; counter++ )
   {
      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      // if counter is divisible by 5, start a new line of output
      if ( counter % 5 == 0 )
         cout << endl;
   } // end for

   return 0; // indicates successful termination
} // end main
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

```
Enter seed: 432
         4         6         3         1         6
         3         1         5         4         2
```

```
Enter seed: 67
       6       1       4       6       2
       1       6       1       6       4
```

Let's run the program several times and observe the results. Notice that the program produces a *different* sequence of random numbers each time it executes, provided that the user enters a different seed. We used the same seed in the first and third sample outputs, so the same series of 10 numbers is displayed in each of those outputs.

To randomize without having to enter a seed each time, we may use a statement like

```
srand( time( 0 ) );
```

This causes the computer to read its clock to obtain the value for the seed. Function time (with the argument 0 as written in the preceding statement) returns the current time as the number of seconds since January 1, 1970, at midnight Greenwich Mean Time (GMT). This value is converted to an unsigned integer and used as the seed to the random number generator. The function prototype for time is in <ctime>.

Common Programming Error 6.7

*Calling function srand more than once in a program restarts the pseudorandom number sequence and can affect the randomness of the numbers produced by rand.*

### Generalized Scaling and Shifting of Random Numbers

Previously, we demonstrated how to write a single statement to simulate the rolling of a six-sided die with the statement

```
face = 1 + rand() % 6;
```

which always assigns an integer (at random) to variable face in the range $1 \leq face \leq 6$. Note that the width of this range (i.e., the number of consecutive integers in the range) is 6 and the starting number in the range is 1. Referring to the preceding statement, we see that the width of the range is determined by the number used to scale rand with the modulus operator (i.e., 6), and the starting number of the range is equal to the number (i.e., 1) that is added to the expression rand % 6. We can generalize this result as

```
number = shiftingValue + rand() % scalingFactor;
```

where shiftingValue is equal to the first number in the desired range of consecutive integers and scalingFactor is equal to the width of the desired range of consecutive integers.

Common Programming Error 6.8

*Using* srand *in place of* rand *to attempt to generate random numbers is a compilation error—function* srand *does not return a value.*

## 6.8. Case Study: Game of Chance; Introducing `enum`

One of the most popular games of chance is a dice game known as "craps," which is played in casinos and back alleys worldwide. The rules of the game are straightforward:

> *A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5 and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first roll, the player wins. If the sum is 2, 3 or 12 on the first roll (called "craps"), the player loses (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first roll, then that sum becomes the player's "point." To win, you must continue rolling the dice until you "make your point." The player loses by rolling a 7 before making the point.*

The program in Fig. 6.10 simulates the game of craps.

**Fig. 6.10. Craps simulation.**

```
// Fig. 6.10: fig06_10.cpp
// Craps simulation.
#include <iostream>
using std::cout;
using std::endl;

#include <cstdlib> // contains prototypes for functions srand and rand
using std::rand;
using std::srand;

#include <ctime> // contains prototype for function time
using std::time;

int rollDice(); // rolls dice, calculates amd displays sum

int main()
{
   // enumeration with constants that represent the game status
   enum Status { CONTINUE, WON, LOST }; // all caps in constants

   int myPoint; // point if no win or loss on first roll
   Status gameStatus; // can contain CONTINUE, WON or LOST

   // randomize random number generator using current time
   srand( time( 0 ) );

   int sumOfDice = rollDice(); // first roll of the dice

   // determine game status and point (if needed) based on first roll
   switch ( sumOfDice )
   {
      case 7: // win with 7 on first roll
      case 11: // win with 11 on first roll
         gameStatus = WON;
         break;
      case 2: // lose with 2 on first roll
      case 3: // lose with 3 on first roll
      case 12: // lose with 12 on first roll
         gameStatus = LOST;
         break;
      default: // did not win or lose, so remember point
         gameStatus = CONTINUE; // game is not over
         myPoint = sumOfDice; // remember the point
         cout << "Point is " << myPoint << endl;
         break; // optional at end of switch
   } // end switch

   // while game is not complete
   while ( gameStatus == CONTINUE ) // not WON or LOST
   {
      sumOfDice = rollDice(); // roll dice again

      // determine game status
      if ( sumOfDice == myPoint ) // win by making point
         gameStatus = WON;
      else
         if ( sumOfDice == 7 ) // lose by rolling 7 before point
            gameStatus = LOST;
   } // end while

   // display won or lost message
   if ( gameStatus == WON )
      cout << "Player wins"   << endl;
   else
      cout << "Player loses"   << endl;

   return 0; // indicates successful termination
} // end main

// roll dice, calculate sum and display results
int rollDice()
{
   // pick random die values
   int die1 = 1 + rand() % 6; // first die roll
   int die2 = 1 + rand() % 6; // second die roll

   int sum = die1 + die2; // compute sum of die values

   // display results of this roll
   cout << "Player rolled "   << die1 << " + " << die2
      << " = " << sum << endl;
   return sum; // end function rollDice
} // end function rollDice
```

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

In the rules of the game, notice that the player must roll two dice on the first roll and on all subsequent rolls. We define function `rollDice` (lines 71–83) to roll the dice and compute and print their sum. Function `rollDice` is defined once, but it is called from two places (lines 27 and 51) in the program. Interestingly, `rollDice` takes no arguments, so we have indicated an empty parameter list in the prototype (line 14) and in the function header (line 71). Function `rollDice` does return the sum of the two dice, so return type `int` is indicated in the function prototype and function header.

The game is reasonably involved. The player may win or lose on the first roll or on any subsequent roll. The program uses variable `gameStatus` to keep track of this. Variable `gameStatus` is declared to be of new type `Status`. Line 19 declares a user-defined type called an enumeration. An enumeration, introduced by the keyword `enum` and followed by a type name (in this case, `Status`), is a set of integer constants represented by identifiers. The values of these enumeration constants start at `0`, unless specified otherwise, and increment by `1`. In the preceding enumeration, the constant `CONTINUE` has the value 0, `WON` has the value 1 and `LOST` has the value 2. The identifiers in an `enum` must be unique, but separate enumeration constants can have the same integer value (we show how to accomplish this momentarily).

Good Programming Practice 6.1

*Capitalize the first letter of an identifier used as a user-defined type name.*

Good Programming Practice 6.2

*Use only uppercase letters in the names of enumeration constants. This makes these constants stand out in a program and reminds you that enumeration constants are not variables.*

Variables of user-defined type Status can be assigned only one of the three values declared in the enumeration. When the game is won, the program sets variable gameStatus to WON (lines 34 and 55). When the game is lost, the program sets variable gameStatus to LOST (lines 39 and 58). Otherwise, the program sets variable gameStatus to CONTINUE (line 42) to indicate that the dice must be rolled again.

Another popular enumeration is

```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
   SEP, OCT, NOV, DEC };
```

which creates user-defined type Months with enumeration constants representing the months of the year. The first value in the preceding enumeration is explicitly set to 1, so the remaining values increment from 1, resulting in the values 1 through 12. Any enumeration constant can be assigned an integer value in the enumeration definition, and subsequent enumeration constants each have a value 1 higher than the preceding constant in the list until the next explicit setting.

After the first roll, if the game is won or lost, the program skips the body of the while statement (lines 49–59) because gameStatus is not equal to CONTINUE. The program proceeds to the if...else statement in lines 62–65, which prints "Player wins" if gameStatus is equal to WON and "Player loses" if gameStatus is equal to LOST.

After the first roll, if the game is not over, the program saves the sum in myPoint (line 43). Execution proceeds with the while statement, because gameStatus is equal to CONTINUE. During each iteration of the while, the program calls rollDice to produce a new sum. If sum matches myPoint, the program sets gameStatus to WON (line 55), the while-test fails, the if...else statement prints "Player wins" and execution terminates. If sum is equal to 7, the program sets gameStatus to LOST (line 58), the while-test fails, the if...else statement prints "Player loses" and execution terminates.

Note the interesting use of the various program control mechanisms we have discussed. The craps program uses two functions—main and rollDice—and the switch, while, if...else, nested if...else and nested if statements.

Good Programming Practice 6.3

*Using enumerations rather than integer constants can make programs clearer and more maintainable. You can set the value of an enumeration constant once in the enumeration declaration.*

Common Programming Error 6.9

*Assigning the integer equivalent of an enumeration constant (rather than the enumeration constant, itself) to a variable of the enumeration type is a compilation error.*

Common Programming Error 6.10

*After an enumeration constant has been defined, attempting to assign another* *value to the enumeration constant is a compilation error.*

## 6.9. Storage Classes

The programs you have seen so far use identifiers for variable names. The attributes of variables include name, type, size and value. This chapter also uses identifiers as names for user-defined functions. Actually, each identifier in a program has other attributes, including storage class, scope and linkage.

C++ provides five storage-class specifiers: auto, register, extern, mutable and static . This section discusses storage-class specifiers auto, register, extern and static. Storage-class specifier mutable (discussed in detail in Chapter 22 , Other Topics) is used exclusively with classes.

### Storage Class, Scope and Linkage

An identifier's storage class determines the period during which that identifier exists in memory. Some identifiers exist briefly, some are repeatedly created and destroyed and others exist for the entire execution of a program. First we discuss the storage classes static and automatic.

An identifier's scope is where the identifier can be referenced in a program. Some identifiers can be referenced throughout a program; others can be referenced from only limited portions of a program. Section 6.10 discusses the scope of identifiers.

An identifier's linkage determines whether it is known only in the source file where it is declared or across multiple files that are compiled, then linked together. An identifier's storage-class specifier helps determine its storage class and linkage.

### Storage Class Categories

The storage-class specifiers can be split into two storage classes: automatic storage class and static storage class. Keywords auto and register are used to declare variables of the automatic storage class. Such variables are created when program execution enters the block in which they are defined, they exist while the block is active and they are destroyed when the program exits the block.

### Local Variables

Only local variables of a function can be of automatic storage class. A function's local variables and parameters normally are of automatic storage class. The storage class specifier auto explicitly declares variables of automatic storage class. For example, the following declaration indicates that double variable x is a local variable of automatic storage class—it exists only in the nearest enclosing pair of curly braces within the body of the function in which the definition appears:

```
auto double x;
```

Local variables are of automatic storage class by default, so keyword auto rarely is used. For the remainder of the text, we refer to variables of automatic storage class simply as automatic variables.

Performance Tip 6.1

*Automatic storage is a means of conserving memory, because automatic storage class variables exist in memory only when the block in which they are defined is executing.*

Software Engineering Observation 6.6

*Automatic storage is an example of the* ***principle of least privilege*** *, which is fundamental to good software engineering. In the context of an application, the principle states that code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more. Why should we have variables stored in memory and accessible when they are not needed?*

### Register Variables

Data in the machine-language version of a program is normally loaded into registers for calculations and other processing.

Performance Tip 6.2

*The storage-class specifier `register` can be placed before an automatic variable declaration to suggest that the compiler maintain the variable in one of the computer's high-speed hardware registers rather than in memory. If intensely used variables such as counters or totals are maintained in hardware registers, the overhead of repeatedly loading the variables from memory into the registers and storing the results back into memory is eliminated.*

Common Programming Error 6.11

*Using multiple storage-class specifiers for an identifier is a syntax error. Only one storage class specifier can be applied to an identifier. For example, if you include `register`, do not also include `auto`.*

The compiler might ignore `register` declarations. For example, there might not be a sufficient number of registers available for the compiler to use. The following definition *suggests* that the integer variable `counter` be placed in one of the computer's registers; regardless of whether the compiler does this, `counter` is initialized to 1:

```
register int counter = 1;
```

The register keyword can be used only with local variables and function parameters.

Performance Tip 6.3

*Often, register is unnecessary. Optimizing compilers can recognize frequently used variables and may place them in registers without needing a register declaration.*

### Static Storage Class

Keywords extern and static declare identifiers for variables of the static storage class and for functions. Static-storage-class variables exist from the point at which the program begins execution and last for the duration of the program. A static-storage-class variable's storage is allocated when the program begins execution. Such a variable is initialized once when its declaration is encountered. For functions, the name of the function exists when the program begins execution, just as for all other functions. However, even though the variables and the function names exist from the start of program execution, this does not mean that these identifiers can be used throughout the program. Storage class and scope (where a name can be used) are separate issues, as we'll see in Section 6.10.

### Identifiers with Static Storage Class

There are two types of identifiers with static storage class—external identifiers (such as global variables and global function names) and local variables declared with the storage-class specifier static. Global variables are created by placing variable declarations outside any class or function definition. Global variables retain their values throughout the execution of the program. Global variables and global functions can be referenced by any function that follows their declarations or definitions in the source file.

Software Engineering Observation 6.7

*Declaring a variable as global rather than local allows unintended side effects to occur when a function that does not need access to the variable accidentally or maliciously modifies it. This is another example of the principle of least privilege. In general, except for truly global resources such as cin and cout, the use of global variables should be avoided except in certain situations with unique performance requirements.*

Software Engineering Observation 6.8

*Variables used only in a particular function should be declared as local variables in that function rather than as global variables.*

Local variables declared with the keyword static are still known only in the function in which they are declared, but, unlike automatic variables, static local variables retain their values when the function returns to its caller. The next time the function is called, the static local variables contain the values they had when the function last completed execution. The following statement declares local variable count to be static and to be initialized to 1:

```
static int count = 1;
```

All numeric variables of the static storage class are initialized to zero if you do not explicitly initialized them, but it is nevertheless a good practice to explicitly initialize all variables.

Storage-class specifiers extern and static have special meaning when they are applied explicitly to external identifiers such as global variables and global function names.

### 6.10. Scope Rules

The portion of the program where an identifier can be used is known as its scope. For example, when we declare a local variable in a block, it can be referenced only in that block and in blocks nested within that block. This section discusses four scopes for an identifier—function scope, file scope, block scope and function-prototype scope . Later we'll see two other scopes—class scope (Chapter 9) and namespace scope (Chapter 22).

An identifier declared outside any function or class has file scope. Such an identifier is "known" in all functions from the point at which it is declared until the end of the file. Global variables, function definitions and function prototypes placed outside a function all have file scope.

Labels (identifiers followed by a colon such as start: ) are the only identifiers with function scope. Labels can be used anywhere in the function in which they appear, but cannot be referenced outside the function body. Labels are used in goto statements, which we do not cover in this book. Labels are implementation details that functions hide from one another.

Identifiers declared inside a block have block scope. Block scope begins at the identifier's declaration and ends at the terminating right brace (}) of the block in which the identifier is declared. Local variables have block scope, as do function parameters, which are also local variables of the function. Any block can contain variable declarations. When blocks are nested and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is "hidden" until the inner block terminates. While executing in the inner block, the inner block sees the value of its own local identifier and not the value of the identically named identifier in the enclosing block. Local variables declared static still have block scope, even though they exist from the time the program begins execution. Storage duration does not affect the scope of an identifier.

The only identifiers with function prototype scope are those used in the parameter list of a function prototype. As mentioned previously, function prototypes do not require names in the parameter list—only types are required. Names appearing in the parameter list of a function prototype are ignored by the compiler. Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity. In a single prototype, a particular identifier can be used only once.

Common Programming Error 6.12

*Accidentally using the same name for an identifier in an inner block that is used for an identifier in an outer block, when in fact you want the identifier in the outer block to be active for the duration of the inner block, is normally a logic error.*

Good Programming Practice 6.4

*Avoid variable names that hide names in outer scopes. This can be accomplished by avoiding the use of duplicate identifiers in a program.*

The program of Fig. 6.11 demonstrates scoping issues with global variables, automatic local variables and static local variables.

**Fig. 6.11. Scoping example.**

```
// Fig. 6.11: fig06_11.cpp
// A scoping example.
#include <iostream>
using std::cout;
using std::endl;

void useLocal(); // function prototype
void useStaticLocal(); // function prototype
void useGlobal(); // function prototype

int x = 1; // global variable

int main()
{
   cout << "global x in main is " << x << endl;

   int x = 5; // local variable to main

   cout << "local x in main's outer scope is " << x << endl;

   { // start new scope
      int x = 7; // hides both x in outer scope and global x

      cout << "local x in main's inner scope is " << x << endl;
   } // end new scope

   cout << "local x in main's outer scope is " << x << endl;

   useLocal(); // useLocal has local x
   useStaticLocal(); // useStaticLocal has static local x
   useGlobal(); // useGlobal uses global x
   useLocal(); // useLocal reinitializes its local x
   useStaticLocal(); // static local x retains its prior value
   useGlobal(); // global x also retains its prior value

   cout << "\nlocal x in main is " << x << endl;
   return 0; // indicates successful termination
} // end main

// useLocal reinitializes local variable x during each call
void useLocal()
{
   int x = 25; // initialized each time useLocal is called

   cout << "\nlocal x is " << x << " on entering useLocal" << endl;
   x++;
   cout << "local x is " << x << " on exiting useLocal" << endl;
} // end function useLocal

// useStaticLocal initializes static local variable x only the
// first time the function is called; value of x is saved
// between calls to this function
void useStaticLocal()
```

```
54 {
55   static int x = 50; // initialized first time useStaticLocal is called
56
57   cout << "\nlocal static x is " << x << " on entering useStaticLocal"
58     << endl;
59   x++;
60   cout << "local static x is " << x << " on exiting useStaticLocal"
61     << endl;
62 } // end function useStaticLocal
63
64 // useGlobal modifies global variable x during each call
65 void useGlobal()
66 {
67   cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
68   x *= 10;
69   cout << "global x is " << x << " on exiting useGlobal" << endl;
70 } // end function useGlobal
```

```
global x in main is 1
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Line 11 declares and initializes global variable x to 1. This global variable is hidden in any block (or function) that declares a variable named x. In main, line 15 displays the value of global variable x. Line 17 declares a local variable x and initializes it to 5. Line 19 outputs this variable to show that the global x is hidden in main. Next, lines 21–25 define a new block in main in which another local variable x is initialized to 7 (line 22). Line 24 outputs this variable to show that it hides x in the outer block of main. When the block exits, the variable x with value 7 is destroyed automatically. Next, line 27 outputs the local variable x in the outer block of main to show that it is no longer hidden.

To demonstrate other scopes, the program defines three functions, each of which takes no arguments and returns nothing. Function useLocal (lines 41–48) declares automatic variable x (line 43) and initializes it to 25. When the program calls useLocal, the function prints the variable, increments it and prints it again before the function returns program control to its caller. Each time the program calls this function, the function recreates automatic variable x and reinitializes it to 25.

Function useStaticLocal (lines 53–62) declares static variable x and initializes it to 50. Local variables declared as static retain their values even when they are out of scope (i.e., the function in which they are declared is not executing). When the program calls useStaticLocal, the function prints x, increments it and prints it again before the function returns program control to its caller. In the next call to this function, static local variable x contains the value 51. The initialization in line 55 occurs only once—the first time useStaticLocal is called.

Function useGlobal (lines 65–70) does not declare any variables. Therefore, when it refers to variable x, the global x (line 11, preceding main) is used. When the program calls useGlobal, the function prints the global variable x, multiplies it by 10 and prints it again before the function returns program control to its caller. The next time the program calls useGlobal, the global variable has its modified value, 10. After executing functions useLocal, useStaticLocal and useGlobal twice each, the program prints the local variable x in main again to show that none of the function calls modified the value of x in main, because the functions all referred to variables in other scopes.

## 6.11. Function Call Stack and Activation Records

To understand how C++ performs function calls, consider a data structure (i.e., collection of related data items) known as a stack. Stacks are last-in, first-out (LIFO) data structures—the last item pushed (inserted) on the stack is the first item popped (removed) from it.

The function call stack (sometimes referred to as the program execution stack)—working "behind the scenes"—supports the function call/return mechanism. It also supports the creation, maintenance and destruction of each called function's automatic variables. As we'll see in Figs. 6.13–6.15, this LIFO behavior is exactly what a function does when returning to the function that called it.

As each function is called, it may, in turn, call other functions, which may, in turn, call other functions—all before any of the functions returns. Each function eventually must return control to the one that called it. So, somehow, we must keep track of the return addresses that each function needs to return control to its caller. The function call stack is the perfect data structure for handling this information. Each time a function is called, an entry is pushed onto the stack. This entry, called a stack frame or an activation record, contains the return address that the called function needs to return to the calling function. When the called function returns, the stack frame for the function call is popped, and control transfers to the return address in the popped stack frame.

The beauty of the call stack is that each called function always finds the information it needs to return to its caller at the top of the call stack. And, if a function makes a call to another function, a stack frame for the new function call is simply pushed onto the call stack. Thus, the return address required by the newly called function to return to its caller is now located at the top of the stack.

The stack frames have another important responsibility. Most functions have automatic variables—parameters and any local variables the function declares. Automatic variables need to exist while a function is executing. They need to remain active if the function makes calls to other functions. But when a called function returns to its caller, the called function's automatic variables need to "go away." The called function's stack frame is a perfect place to store the function's automatic variables. That stack frame exists as long as the called function is active. When that function returns—and no longer needs its local automatic variables—its stack frame is popped from the stack, and those local automatic variables are no longer known to the program.

Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the function call stack. If more function calls occur than can have their activation records stored on the function call stack, an error known as stack overflow occurs.

### Function Call Stack in Action

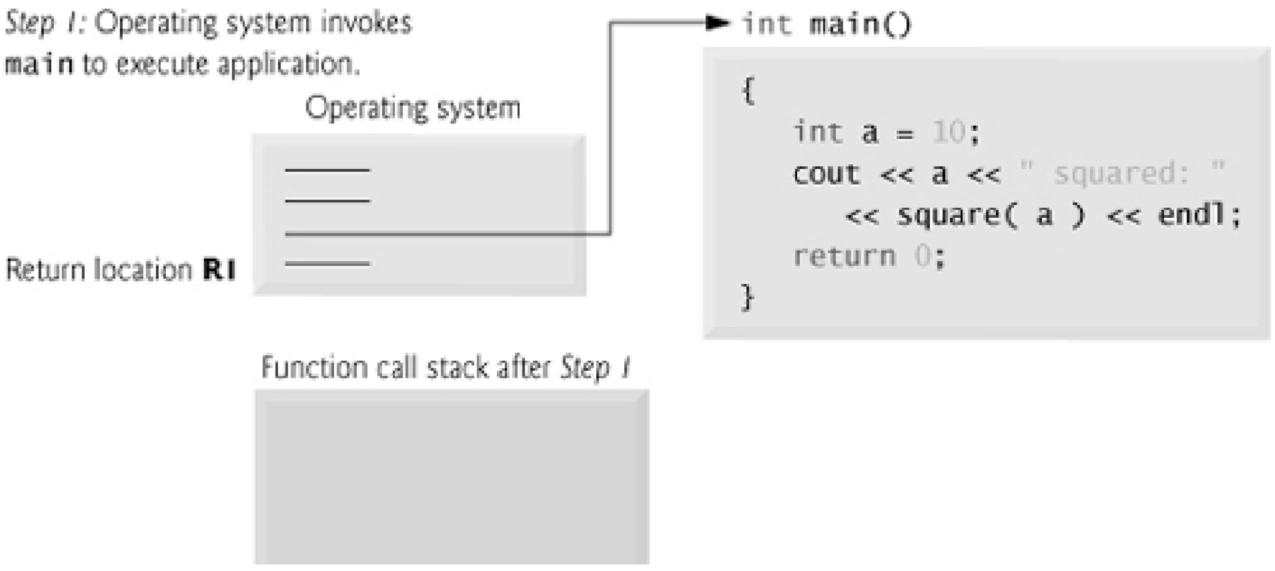
So, as we've seen, the call stack and activation records support the function call/return mechanism and the creation and destruction of automatic variables. Now let's consider how the call stack supports the operation of a square function called by main (lines 11–17 of Fig. 6.12). First the operating system calls main—this pushes an activation record onto the stack (shown in Fig. 6.13). The activation record tells main how to return to the operating system (i.e., transfer to return address R1) and contains the space for main's automatic variable (i.e., a, which is initialized to 10).

**Fig. 6.12. square function used to demonstrate the function call stack and activation records.**

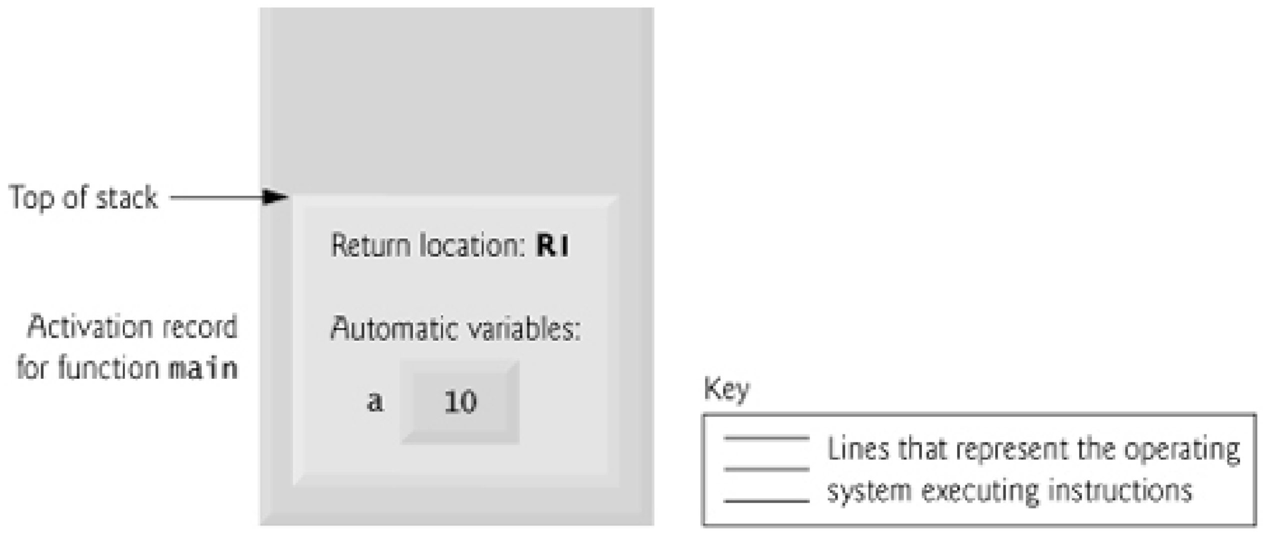
```
// Fig. 6.12: fig06_12.cpp
// square function used to demonstrate the function
// call stack and activation records.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int square( int ); // prototype for function square

int main()
{
   int a = 10; // value to square (local automatic variable in main)

   cout << a << " squared: "  << square( a ) << endl; // display a squared
   return 0; // indicate successful termination
} // end main

// returns the square of an integer
int square( int x ) // x is a local variable
{
   return x * x; // calculate square and return result
} // end function square
```

```
10 squared: 100
```

**Fig. 6.13. Function call stack after the operating system invokes main.**


Step 1: Operating system invokes main to execute application.
Operating system
Return location R1
Function call stack after Step 1

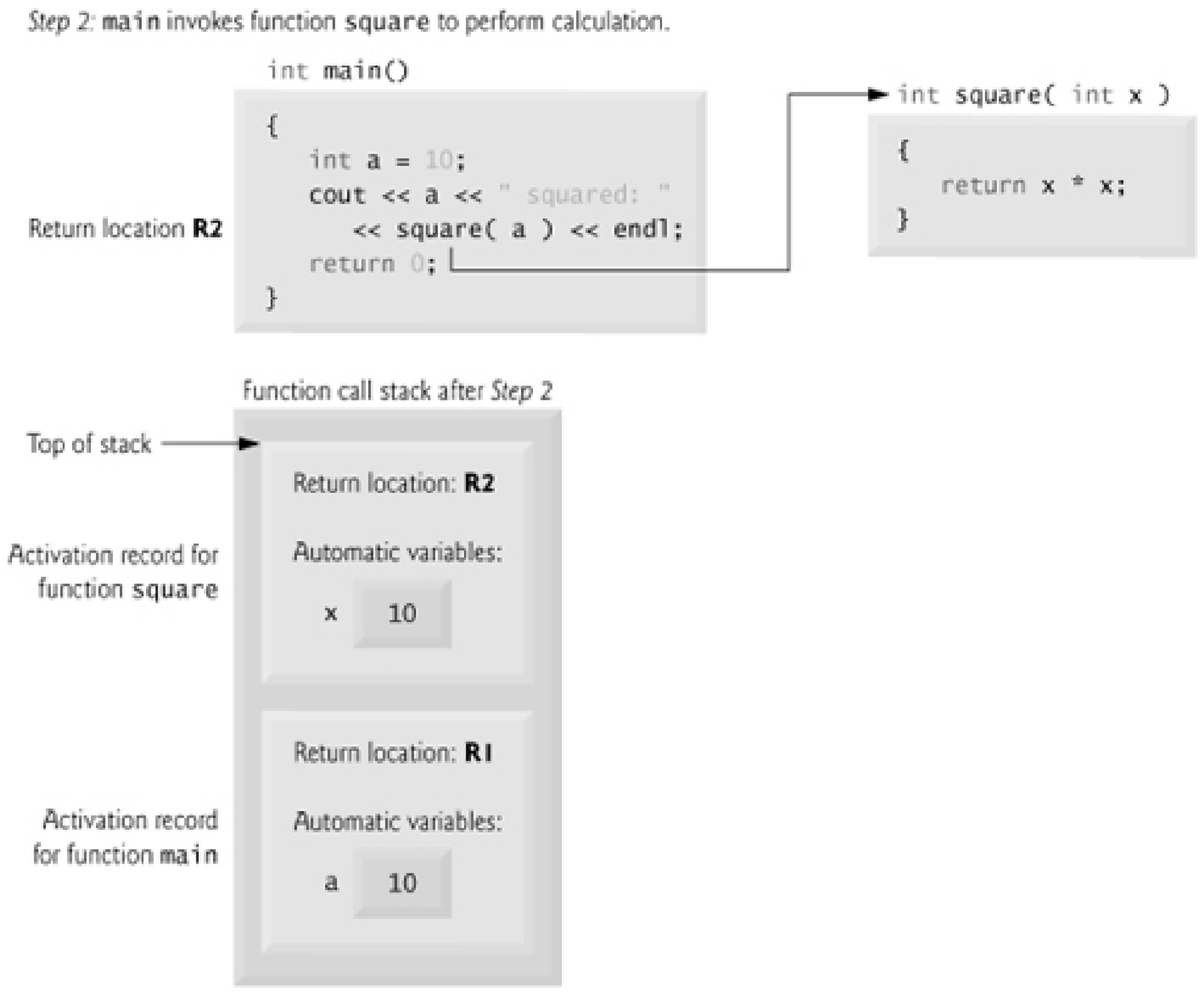
```
int main()
{
   int a = 10;
   cout << a << " squared: "
      << square( a ) << endl;
   return 0;
}
```



Top of stack
Return location: R1
Activation record for function main
Automatic variables:
a
10
Key
Lines that represent the operating system executing instructions

Function main—before returning to the operating system—now calls function square in line 15 of Fig. 6.12. This causes a stack frame for square (lines 20–23) to be pushed onto the function call stack (Fig. 6.14). This stack frame contains the return address that square needs to return to main (i.e., R2) and the memory for square's automatic variable (i.e., x).

**Fig. 6.14. Function call stack after main invokes function square to perform the calculation.**



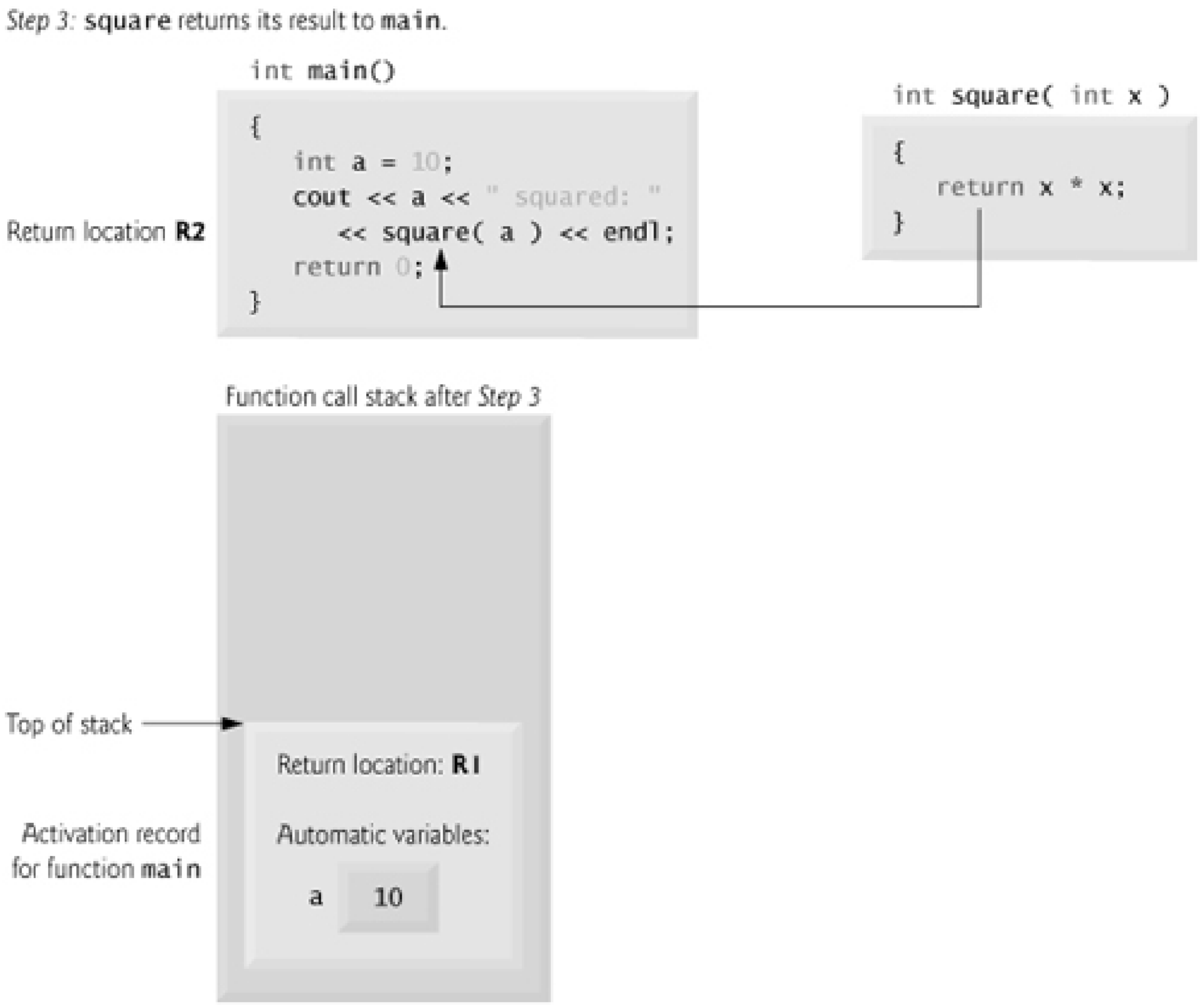
Step 2: main invokes function square to perform calculation.
Return location R2
Function call stack after Step 2
Top of stack
Return location: R2
Activation record for function square
Automatic variables:
x
10
Return location: R1
Activation record for function main
Automatic variables:
a
10

```
int main()
{
   int a = 10;
   cout << a << " squared: "
      << square( a ) << endl;
   return 0;
}
```

```
int square( int x )
{
   return x * x;
}
```

After square calculates the square of its argument, the function needs to return to main —and no longer needs the memory for its automatic variable x . So the stack is popped—giving square the return location in main (i.e., R2) and losing square's automatic variable. Figure 6.15 shows the function call stack after square 's activation record has been popped.

**Fig. 6.15. Function call stack after function square returns to main.**



```
int main()
{
   int a = 10;
   cout << a << " squared: "
      << square( a ) << endl;
   return 0;
}
```

```
int square( int x )
{
   return x * x;
}
```

Step 3: square returns its result to main.
Return location R2
Function call stack after Step 3
Top of stack
Activation record for function main
Return location: R1
Automatic variables:
a
10

Function main now displays the result of calling square (line 15), then executes the return statement (line 16). This causes the activation record for main to be popped from the stack. This gives main the address it needs to return to the operating system (i.e., R1 in Fig. 6.13) and causes the memory for main's automatic variable (i.e., a) to become unavailable.

## 6.12. Functions with Empty Parameter Lists

In C++, an empty parameter list is specified by writing either `void` or nothing at all in parentheses. The prototype

```
void print();
```

specifies that function `print` does not take arguments and does not return a value. Figure 6.16 demonstrates both ways to declare and use functions with empty parameter lists.

**Fig. 6.16. Functions that take no arguments.**

```cpp
// Fig. 6.16: fig06_16.cpp
// Functions that take no arguments.
#include <iostream>
using std::cout;
using std::endl;

void function1(); // function that takes no arguments
void function2( void ); // function that takes no arguments

int main()
{
   function1(); // call function1 with no arguments
   function2(); // call function2 with no arguments
   return 0; // indicates successful termination
} // end main

// function1 uses an empty parameter list to specify that
// the function receives no arguments
void function1()
{
   cout << "function1 takes no arguments" << endl;
} // end function1

// function2 uses a void parameter list to specify that
// the function receives no arguments
void function2( void )
{
   cout << "function2 also takes no arguments" << endl;
} // end function2
```

```
function1 takes no arguments
function2 also takes no arguments
```

Portability Tip 6.2

*The meaning of an empty function parameter list in C++ is dramatically different than in C. In C, it means all argument checking is disabled (i.e., the function call can pass any arguments it wants). In C++, it means that the function explicitly takes no arguments. Thus, C programs using this feature might cause compilation errors when compiled in C++.*

Common Programming Error 6.13

*C++ programs do not compile unless function prototypes are provided for every function or each function is defined before it is called.*

### 6.13. Inline Functions

Implementing a program as a set of functions is good from a software engineering standpoint, but function calls involve execution-time overhead. C++ provides inline functions to help reduce function call overhead—especially for small functions. Placing the qualifier inline before a function's return type in the function definition "advises" the compiler to generate a copy of the function's code in place (when appropriate) to avoid a function call. The trade-off is that multiple copies of the function code are inserted in the program (often making the program larger) rather than there being a single copy of the function to which control is passed each time the function is called. The compiler can ignore the inline qualifier and typically does so for all but the smallest functions.

Software Engineering Observation 6.9

*Any change to an inline function requires all clients of the function to be recompiled. This can be significant in some program development and maintenance situations.*

Good Programming Practice 6.5

*The inline qualifier should be used only with small, frequently used functions.*

Performance Tip 6.4

*Using inline functions can reduce execution time but may increase program size.*

Figure 6.17 uses inline function cube (lines 11–14) to calculate the volume of a cube of side side. Keyword const in the parameter list of function cube (line 11) tells the compiler that the function does not modify variable side. This ensures that the value of side is not changed by the function when the calculation is performed. (Keyword const is discussed in detail in Chapters 7, 8 and 10.) Notice that the complete definition of function cube appears before it is used in the program. This is required so that the compiler knows how to expand a cube function call into its inlined code. For this reason, reusable inline functions are typically placed in header files, so that their definitions can be included in each source file that uses them.

**Fig. 6.17. inline function that calculates the volume of a cube.**

```
// Fig. 6.17: fig06_17.cpp
// Using an inline function to calculate the volume of a cube.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

// Definition of inline function cube. Definition of function appears
// before function is called, so a function prototype is not required.
// First line of function definition acts as the prototype.
inline double cube( const double side )
{
   return side * side * side; // calculate cube
} // end function cube

int main()
{
   double sideValue; // stores value entered by user
   cout << "Enter the side length of your cube: ";
   cin >> sideValue; // read value from user

  // calculate cube of sideValue and display result
  cout << "Volume of cube with side "
     << sideValue << " is " << cube( sideValue ) << endl;
  return 0; // indicates successful termination
} // end main
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

Software Engineering Observation 6.10

*The* const *qualifier should be used to enforce the principle of least privilege. Using the principle of least privilege to properly design software can greatly reduce debugging time and improper side effects and can make a program easier to modify and maintain.*

## 6.14. References and Reference Parameters

Two ways to pass arguments to functions in many programming languages are pass-by-value and pass-by-reference . When an argument is passed by value, a *copy* of the argument's value is made and passed (on the function call stack) to the called function. Changes to the copy do not affect the original variable's value in the caller. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. Each argument that has been passed in the programs in this chapter so far has been passed by value.

Performance Tip 6.5

*One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.*

### Reference Parameters

This section introduces reference parameters —the first of the two means C++ provides for performing pass-by-reference. With pass-by-reference, the caller gives the called function the ability to access the caller's data directly, and to modify that data if the called function chooses to do so.

Performance Tip 6.6

*Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.*

Software Engineering Observation 6.11

*Pass-by-reference can weaken security, because the called function can corrupt the caller's data.*

Later, we'll show how to achieve the performance advantage of pass-by-reference while simultaneously achieving the software engineering advantage of protecting the caller's data from corruption.

A reference parameter is an alias for its corresponding argument in a function call. To indicate that a function parameter is passed by reference, simply follow the parameter's type in the function prototype by an ampersand (& ); use the same convention when listing the parameter's type in the function header. For example, the following declaration in a function header

```
int &count
```

when read from right to left is pronounced "count is a reference to an int." In the function call, simply mention the variable by name to pass it by reference. Then, mentioning the variable by its parameter name in the body of the called function actually refers to the original variable in the calling function, and the original variable can be modified directly by the called function. As always, the function prototype and header must agree.

### Passing Arguments by Value and by Reference

Figure 6.18 compares pass-by-value and pass-by-reference with reference parameters. The "styles" of the arguments in the calls to function squareByValue and function squareByReference are identical—both variables are simply mentioned by name in the function calls. Without checking the function prototypes or function definitions, it is not possible to tell from the calls alone whether either function can modify its arguments. Because function prototypes are mandatory, the compiler has no trouble resolving the ambiguity.

**Fig. 6.18. Passing arguments by value and by reference.**

```
// Fig. 6.18: fig06_18.cpp
// Comparing pass-by-value and pass-by-reference with references.
#include <iostream>
using std::cout;
using std::endl;

int squareByValue( int ); // function prototype (value pass)
void squareByReference( int & ); // function prototype (reference pass)

int main()
{
   int x = 2; // value to square using squareByValue
   int z = 4; // value to square using squareByReference

   // demonstrate squareByValue
   cout << "x = " << x << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
      << squareByValue( x ) << endl;
   cout << "x = " << x << " after squareByValue\n" << endl;

   // demonstrate squareByReference
   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;
   return 0; // indicates successful termination
} // end main

// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue( int number )
{
   return number *= number; // caller's argument not modified
} // end function squareByValue

// squareByReference multiplies numberRef by itself and stores the result
// in the variable to which numberRef refers in function main
void squareByReference( int &numberRef )
{
   numberRef *= numberRef; // caller's argument modified
} // end function squareByReference
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Common Programming Error 6.14

*Because reference parameters are mentioned only by name in the body of the called function, you might inadvertently treat reference parameters as pass-by-value parameters. This can cause unexpected side effects if the original copies of the variables are changed by the function.*

Chapter 8 discusses pointers; pointers enable an alternate form of pass-by-reference in which the style of the call clearly indicates pass-by-reference (and the potential for modifying the caller's arguments).

Performance Tip 6.7

*For passing large objects, use a constant reference parameter to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object.*

Software Engineering Observation 6.12

*Many programmers do not bother to declare parameters passed by value as `const`, even though the called function should not be modifying the passed argument. Keyword `const` in this context would protect only a copy of the original argument, not the original argument itself, which when passed by value is safe from modification by the called function.*

To specify a reference to a constant, place the `const` qualifier before the type specifier in the parameter declaration.

Note the placement of `&` in function `squareByReference`'s parameter list (line 37, Fig. 6.18). Some C++ programmers prefer to write the equivalent form `int& numberRef`.

Software Engineering Observation 6.13

*For the combined reasons of clarity and performance, many C++ programmers prefer that modifiable arguments be passed to functions by using pointers (which we study in Chapter 8), small nonmodifiable arguments be passed by value and large nonmodifiable arguments be passed to functions by using references to constants.*

### References as Aliases within a Function

References can also be used as aliases for other variables within a function (although they typically are used with functions as shown in Fig. 6.18). For example, the code

```
int count = 1; // declare integer variable count
int &cRef = count; // create cRef as an alias for count
cRef++; // increment count (using its alias cRef)
```

increments variable count by using its alias cRef. Reference variables must be initialized in their declarations (see Fig. 6.19 and Fig. 6.20 ) and cannot be reassigned as aliases to other variables. Once a reference is declared as an alias for another variable, all operations supposedly performed on the alias (i.e., the reference) are actually performed on the original variable. The alias is simply another name for the original variable. Taking the address of a reference and comparing references do not cause syntax errors; rather, each operation actually occurs on the variable for which the reference is an alias. Unless it is a reference to a constant, a reference argument must be an *lvalue* (e.g., a variable name), not a constant or expression that returns an *rvalue* (e.g., the result of a calculation). See Section 5.9 for definitions of the terms *lvalue* and *rvalue*.

**Fig. 6.19. Initializing and using a reference.**

```
// Fig. 6.19: fig06_19.cpp
// References must be initialized.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
  int x = 3;
   int &y = x; // y refers to (is an alias for) x

   cout << "x = " << x << endl << "y = " << y << endl;
   y = 7; // actually modifies x
   cout << "x = " << x << endl << "y = " << y << endl;
   return 0; // indicates successful termination
 } // end main
```

```
x = 3
y = 3
x = 7
y = 7
```

**Fig. 6.20. Uninitialized reference causes a syntax error.**

```
 1  // Fig. 6.20: fig06_20.cpp
 2  // References must be initialized.
 3  #include<iostream>
 4  usingstd::cout;
 5  usingstd::endl;
 6
 7  int main()
 8  {
 9     int x = 3;
10     int &y; // Error: y must be initialized
11
12     cout << "x = " << x << endl << "y = "  << y << endl;
13     y = 7;
14     cout << "x = " << x << endl << "y = "  << y << endl;
15     return 0; // indicates successful termination
16  } // end main
```

*Borland C++ command-line compiler error message:*

```
Error E2304 C:\cppfp_examples\ch06\Fig06_20\fig06_20.cpp 10:
   Reference variable 'y' must be initialized in function main()
```

*Microsoft Visual C++ compiler error message:*

```
C:\cppfp_examples\ch06\Fig06_20\fig06_20.cpp(10) : error C2530: 'y' :
   references must be initialized
```

*GNU C++ compiler error message:*

```
fig06_20.cpp:10: error: 'y' declared as a reference but not initialized
```

### Returning a Reference from a Function

Functions can return references, but this can be dangerous. When returning a reference to a variable declared in the called function, the variable should be declared `static` within that function. Otherwise, the reference refers to an automatic variable that is discarded when the function terminates; such a variable is said to be "undefined," and the program's behavior is unpredictable. References to undefined variables are called dangling references.

Common Programming Error 6.15

*Not initializing a reference variable when it is declared is a compilation error,* *unless the declaration is part of a function's parameter list. Reference parameters are initialized when the function in which they are declared is called.*

Common Programming Error 6.16

*Attempting to reassign a previously declared reference to be an alias to another variable is a logic error. The value of the other variable is simply assigned to the variable for which the reference is already an alias.*

Common Programming Error 6.17

*Returning a reference to an automatic variable in a called function is a logic error. Some compilers issue a warning when this occurs.*

### Error Messages for Uninitialized References

The C++ standard does not specify the error messages that compilers use to indicate particular errors. For this reason, Fig. 6.20 shows the error messages produced by the Borland C++ command-line compiler, Microsoft Visual C++ compiler and GNU C++ compiler when a reference is not initialized.