

- [C++ How to Program, Fifth Edition](#)
- [Table of Contents](#)
- [Copyright](#)
- [DeitelR Books, Cyber Classrooms, Complete Training Courses and Web-Based Training Courses published by Prentice Hall](#)
- [Preface](#)
 - [Features of C++ How to Program, 5/e](#)
 - [Teaching Approach](#)
 - [Tour of the Book](#)
 - [Object-Oriented Design of an ATM with the UML: A Tour of the Optional Software Engineering Case Study](#)
 - [Software Bundled with C++ How to Program, 5/e](#)
 - [Teaching Resources for C++ How to Program, 5/e](#)
 - [C++ Multimedia Cyber Classroom, 5/e, Online](#)
 - [C++ in the Lab](#)
 - [CourseCompassSM, WebCT™ and Blackboard™](#)
 - [PearsonChoices](#)
 - [The DeitelR Buzz Online Free E-mail Newsletter](#)
 - [Acknowledgments](#)
 - [About the Authors](#)
 - [About Deitel & Associates, Inc.](#)
- [Before You Begin](#)
 - [Resources on the CD That Accompanies C++ How to Program, Fifth Edition](#)
 - [Copying and Organizing Files](#)
 - [Copying the Book Examples from the CD](#)
 - [Changing the Read-Only Property of Files](#)
- [Chapter 1. Introduction to Computers, the Internet and World Wide Web](#)
 - [Section 1.1. Introduction](#)
 - [Section 1.2. What Is a Computer?](#)
 - [Section 1.3. Computer Organization](#)
 - [Section 1.4. Early Operating Systems](#)
 - [Section 1.5. Personal, Distributed and Client/Server Computing](#)
 - [Section 1.6. The Internet and the World Wide Web](#)
 - [Section 1.7. Machine Languages, Assembly Languages and High-Level Languages](#)
 - [Section 1.8. History of C and C++](#)
 - [Section 1.9. C++ Standard Library](#)
 - [Section 1.10. History of Java](#)
 - [Section 1.11. FORTRAN, COBOL, Pascal and Ada](#)
 - [Section 1.12. Basic, Visual Basic, Visual C++, C# and .NET](#)
 - [Section 1.13. Key Software Trend: Object Technology](#)
 - [Section 1.14. Typical C++ Development Environment](#)
 - [Section 1.15. Notes About C++ and C++ How to Program, 5/e](#)
 - [Section 1.16. Test-Driving a C++ Application](#)
 - [Section 1.17. Software Engineering Case Study: Introduction to Object Technology and the UML \(Required\)](#)
 - [Section 1.18. Wrap-Up](#)
 - [Section 1.19. Web Resources](#)
 - [Summary](#)
 - [Terminology](#)
 - [Self-Review Exercises](#)
 - [Answers to Self-Review Exercises](#)
 - [Exercises](#)
- [Chapter 2. Introduction to C++ Programming](#)
 - [Section 2.1. Introduction](#)
 - [Section 2.2. First Program in C++: Printing a Line of Text](#)
 - [Section 2.3. Modifying Our First C++ Program](#)
 - [Section 2.4. Another C++ Program: Adding Integers](#)
 - [Section 2.5. Memory Concepts](#)
 - [Section 2.6. Arithmetic](#)
 - [Section 2.7. Decision Making: Equality and Relational Operators](#)

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)



C++ How to Program, Fifth Edition

By H. M. Deitel - Deitel & Associates, Inc., P. J. Deitel - Deitel & Associates, Inc.

Publisher: Prentice Hall

Pub Date: January 05, 2005

Print ISBN-10: 0-13-185757-6

eText ISBN-10: 0-13-186103-4

Print ISBN-13: 978-0-13-185757-5

eText ISBN-13: 978-0-13-186103-9

Pages: 1536

- [Table of Contents](#)
- [Index](#)

Best-selling C++ text significantly revised to include new early objects coverage and new streamlined case studies.

page footer

The CHM file was converted to HTML by [chm2web](#) software.



Section 2.8. (Optional) Software Engineering Case Study: Examining the ATM Requirements Document
Section 2.9. Wrap-Up
Summary
Terminology
Self-Review Exercises
Answers to Self-Review Exercises
Exercises
 Chapter 3. Introduction to Classes and Objects
Section 3.1. Introduction
Section 3.2. Classes, Objects, Member Functions and Data Members
Section 3.3. Overview of the Chapter Examples
Section 3.4. Defining a Class with a Member Function
Section 3.5. Defining a Member Function with a Parameter
Section 3.6. Data Members, set Functions and get Functions
Section 3.7. Initializing Objects with Constructors
Section 3.8. Placing a Class in a Separate File for Reusability
Section 3.9. Separating Interface from Implementation
Section 3.10. Validating Data with set Functions
Section 3.11. (Optional) Software Engineering Case Study: Identifying the Classes in the ATM Requirements Document
Section 3.12. Wrap-Up
Summary
Terminology
Self-Review Exercises
Answers to Self-Review Exercises
Exercises
 Chapter 4. Control Statements: Part 1
Section 4.1. Introduction
Section 4.2. Algorithms
Section 4.3. Pseudocode
Section 4.4. Control Structures
Section 4.5. if Selection Statement
Section 4.6. if...else Double-Selection Statement
Section 4.7. while Repetition Statement
Section 4.8. Formulating Algorithms: Counter-Controlled Repetition
Section 4.9. Formulating Algorithms: Sentinel-Controlled Repetition
Section 4.10. Formulating Algorithms: Nested Control Statements
Section 4.11. Assignment Operators
Section 4.12. Increment and Decrement Operators
Section 4.13. (Optional) Software Engineering Case Study: Identifying Class Attributes in the ATM System
Section 4.14. Wrap-Up
Summary
Terminology
Self-Review Exercises
Answers to Self-Review Exercises
Exercises
 Chapter 5. Control Statements: Part 2
Section 5.1. Introduction
Section 5.2. Essentials of Counter-Controlled Repetition
Section 5.3. for Repetition Statement
Section 5.4. Examples Using the for Statement
Section 5.5. do...while Repetition Statement
Section 5.6. switch Multiple-Selection Statement
Section 5.7. break and continue Statements
Section 5.8. Logical Operators
Section 5.9. Confusing Equality (==) and Assignment (=) Operators
Section 5.10. Structured Programming Summary
Section 5.11. (Optional) Software Engineering Case Study: Identifying Objects' States and Activities in the ATM System
Section 5.12. Wrap-Up
Summary
Terminology

- Self-Review Exercises
 - Answers to Self-Review Exercises
 - Exercises
 -  Chapter 6. Functions and an Introduction to Recursion
 - Section 6.1. Introduction
 - Section 6.2. Program Components in C++
 - Section 6.3. Math Library Functions
 - Section 6.4. Function Definitions with Multiple Parameters
 - Section 6.5. Function Prototypes and Argument Coercion
 - Section 6.6. C++ Standard Library Header Files
 - Section 6.7. Case Study: Random Number Generation
 - Section 6.8. Case Study: Game of Chance and Introducing enum
 - Section 6.9. Storage Classes
 - Section 6.10. Scope Rules
 - Section 6.11. Function Call Stack and Activation Records
 - Section 6.12. Functions with Empty Parameter Lists
 - Section 6.13. Inline Functions
 - Section 6.14. References and Reference Parameters
 - Section 6.15. Default Arguments
 - Section 6.16. Unary Scope Resolution Operator
 - Section 6.17. Function Overloading
 - Section 6.18. Function Templates
 - Section 6.19. Recursion
 - Section 6.20. Example Using Recursion: Fibonacci Series
 - Section 6.21. Recursion vs. Iteration
 - Section 6.22. (Optional) Software Engineering Case Study: Identifying Class Operations in the ATM System
 - Section 6.23. Wrap-Up
 - Summary
 - Terminology
 - Self-Review Exercises
 - Answers to Self-Review Exercises
 - Exercises
-  Chapter 7. Arrays and Vectors
 - Section 7.1. Introduction
 - Section 7.2. Arrays
 - Section 7.3. Declaring Arrays
 - Section 7.4. Examples Using Arrays
 - Section 7.5. Passing Arrays to Functions
 - Section 7.6. Case Study: Class GradeBook Using an Array to Store Grades
 - Section 7.7. Searching Arrays with Linear Search
 - Section 7.8. Sorting Arrays with Insertion Sort
 - Section 7.9. Multidimensional Arrays
 - Section 7.10. Case Study: Class GradeBook Using a Two-Dimensional Array
 - Section 7.11. Introduction to C++ Standard Library Class Template vector
 - Section 7.12. (Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM System
 - Section 7.13. Wrap-Up
- Summary
- Terminology
- Self-Review Exercises
- Answers to Self-Review Exercises
- Exercises
- Recursion Exercises
- vector Exercises
-  Chapter 8. Pointers and Pointer-Based Strings
 - Section 8.1. Introduction
 - Section 8.2. Pointer Variable Declarations and Initialization
 - Section 8.3. Pointer Operators
 - Section 8.4. Passing Arguments to Functions by Reference with Pointers
 - Section 8.5. Using const with Pointers
 - Section 8.6. Selection Sort Using Pass-by-Reference
 - Section 8.7. sizeof Operators

Section 8.8. Pointer Expressions and Pointer Arithmetic
Section 8.9. Relationship Between Pointers and Arrays
Section 8.10. Arrays of Pointers
Section 8.11. Case Study: Card Shuffling and Dealing Simulation
Section 8.12. Function Pointers
Section 8.13. Introduction to Pointer-Based String Processing
Section 8.14. Wrap-Up
Summary
Terminology
Self-Review Exercises
Answers to Self-Review Exercises
Exercises
Special Section: Building Your Own Computer
More Pointer Exercises
String-Manipulation Exercises
Special Section: Advanced String-Manipulation Exercises
A Challenging String-Manipulation Project
 Chapter 9. Classes: A Deeper Look, Part 1
Section 9.1. Introduction
Section 9.2. Time Class Case Study
Section 9.3. Class Scope and Accessing Class Members
Section 9.4. Separating Interface from Implementation
Section 9.5. Access Functions and Utility Functions
Section 9.6. Time Class Case Study: Constructors with Default Arguments
Section 9.7. Destructors
Section 9.8. When Constructors and Destructors Are Called
Section 9.9. Time Class Case Study: A Subtle TrapReturning a Reference to a private Data Member
Section 9.10. Default Memberwise Assignment
Section 9.11. Software Reusability
Section 9.12. (Optional) Software Engineering Case Study: Starting to Program the Classes of the ATM System
Section 9.13. Wrap-Up
Summary
Terminology
Self-Review Exercises
Answers to Self-Review Exercises
Exercises
 Chapter 10. Classes: A Deeper Look, Part 2
Section 10.1. Introduction
Section 10.2. const (Constant) Objects and const Member Functions
Section 10.3. Composition: Objects as Members of Classes
Section 10.4. friend Functions and friend Classes
Section 10.5. Using the this Pointer
Section 10.6. Dynamic Memory Management with Operators new and delete
Section 10.7. static Class Members
Section 10.8. Data Abstraction and Information Hiding
Section 10.9. Container Classes and Iterators
Section 10.10. Proxy Classes
Section 10.11. Wrap-Up
Summary
Terminology
Self-Review Exercises
Answers to Self-Review Exercises
Exercises
 Chapter 11. Operator Overloading: String and Array Objects
Section 11.1. Introduction
Section 11.2. Fundamentals of Operator Overloading
Section 11.3. Restrictions on Operator Overloading
Section 11.4. Operator Functions as Class Members vs. Global Functions
Section 11.5. Overloading Stream Insertion and Stream Extraction Operators
Section 11.6. Overloading Unary Operators

- Section 11.7. Overloading Binary Operators
 - Section 11.8. Case Study: Array Class
 - Section 11.9. Converting between Types
 - Section 11.10. Case Study: String Class
 - Section 11.11. Overloading ++ and --
 - Section 11.12. Case Study: A Date Class
 - Section 11.13. Standard Library Class string
 - Section 11.14. explicit Constructors
 - Section 11.15. Wrap-Up
 - Summary
 - Terminology
 - Self-Review Exercises
 - Answers to Self-Review Exercises
 - Exercises
- Chapter 12. Object-Oriented Programming: Inheritance
 - Section 12.1. Introduction
 - Section 12.2. Base Classes and Derived Classes
 - Section 12.3. protected Members
 - Section 12.4. Relationship between Base Classes and Derived Classes
 - Section 12.5. Constructors and Destructors in Derived Classes
 - Section 12.6. public, protected and private Inheritance
 - Section 12.7. Software Engineering with Inheritance
 - Section 12.8. Wrap-Up
 - Summary
 - Terminology
 - Self-Review Exercises
 - Answers to Self-Review Exercises
 - Exercises
- Chapter 13. Object-Oriented Programming: Polymorphism
 - Section 13.1. Introduction
 - Section 13.2. Polymorphism Examples
 - Section 13.3. Relationships Among Objects in an Inheritance Hierarchy
 - Section 13.4. Type Fields and switch Statements
 - Section 13.5. Abstract Classes and Pure virtual Functions
 - Section 13.6. Case Study: Payroll System Using Polymorphism
 - Section 13.7. (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"
 - Section 13.8. Case Study: Payroll System Using Polymorphism and Run-Time Type Information with Downcasting, dynamic_cast, typeid and type_info
 - Section 13.9. Virtual Destructors
 - Section 13.10. (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System
 - Section 13.11. Wrap-Up
 - Summary
 - Terminology
 - Self-Review Exercises
 - Answers to Self-Review Exercises
 - Exercises
- Chapter 14. Templates
 - Section 14.1. Introduction
 - Section 14.2. Function Templates
 - Section 14.3. Overloading Function Templates
 - Section 14.4. Class Templates
 - Section 14.5. Nontype Parameters and Default Types for Class Templates
 - Section 14.6. Notes on Templates and Inheritance
 - Section 14.7. Notes on Templates and Friends
 - Section 14.8. Notes on Templates and static Members
 - Section 14.9. Wrap-Up
 - Summary
 - Terminology
 - Self-Review Exercises
 - Answers to Self-Review Exercises
 - Exercises
- Chapter 15. Stream Input/Output

— Section 15.1. Introduction
— Section 15.2. Streams
— Section 15.3. Stream Output
— Section 15.4. Stream Input
— Section 15.5. Unformatted I/O using read, write and gcount
— Section 15.6. Introduction to Stream Manipulators
— Section 15.7. Stream Format States and Stream Manipulators
— Section 15.8. Stream Error States
— Section 15.9. Tying an Output Stream to an Input Stream
— Section 15.10. Wrap-Up
— Summary
— Terminology
— Self-Review Exercises
— Answers to Self-Review Exercises
— Exercises
— Chapter 16. Exception Handling
— Section 16.1. Introduction
— Section 16.2. Exception-Handling Overview
— Section 16.3. Example: Handling an Attempt to Divide by Zero
— Section 16.4. When to Use Exception Handling
— Section 16.5. Rethrowing an Exception
— Section 16.6. Exception Specifications
— Section 16.7. Processing Unexpected Exceptions
— Section 16.8. Stack Unwinding
— Section 16.9. Constructors, Destructors and Exception Handling
— Section 16.10. Exceptions and Inheritance
— Section 16.11. Processing new Failures
— Section 16.12. Class auto_ptr and Dynamic Memory Allocation
— Section 16.13. Standard Library Exception Hierarchy
— Section 16.14. Other Error-Handling Techniques
— Section 16.15. Wrap-Up
— Summary
— Terminology
— Self-Review Exercises
— Answers to Self-Review Exercises
— Exercises
— Chapter 17. File Processing
— Section 17.1. Introduction
— Section 17.2. The Data Hierarchy
— Section 17.3. Files and Streams
— Section 17.4. Creating a Sequential File
— Section 17.5. Reading Data from a Sequential File
— Section 17.6. Updating Sequential Files
— Section 17.7. Random-Access Files
— Section 17.8. Creating a Random-Access File
— Section 17.9. Writing Data Randomly to a Random-Access File
— Section 17.10. Reading from a Random-Access File Sequentially
— Section 17.11. Case Study: A Transaction-Processing Program
— Section 17.12. Input/Output of Objects
— Section 17.13. Wrap-Up
— Summary
— Terminology
— Self-Review Exercises
— Answers to Self-Review Exercises
— Exercises
— Chapter 18. Class string and String Stream Processing
— Section 18.1. Introduction
— Section 18.2. string Assignment and Concatenation
— Section 18.3. Comparing strings
— Section 18.4. Substrings

└ Section 18.5. Swapping strings
└ Section 18.6. string Characteristics
└ Section 18.7. Finding Strings and Characters in a string
└ Section 18.8. Replacing Characters in a string
└ Section 18.9. Inserting Characters into a string
└ Section 18.10. Conversion to C-Style Pointer-Based char * Strings
└ Section 18.11. Iterators
└ Section 18.12. String Stream Processing
└ Section 18.13. Wrap-Up
└ Summary
└ Terminology
└ Self-Review Exercises
└ Answers to Self-Review Exercises
└ Exercises
└ Chapter 19. Web Programming
└ Section 19.1. Introduction
└ Section 19.2. HTTP Request Types
└ Section 19.3. Multitier Architecture
└ Section 19.4. Accessing Web Servers
└ Section 19.5. Apache HTTP Server
└ Section 19.6. Requesting XHTML Documents
└ Section 19.7. Introduction to CGI
└ Section 19.8. Simple HTTP Transactions
└ Section 19.9. Simple CGI Scripts
└ Section 19.10. Sending Input to a CGI Script
└ Section 19.11. Using XHTML Forms to Send Input
└ Section 19.12. Other Headers
└ Section 19.13. Case Study: An Interactive Web Page
└ Section 19.14. Cookies
└ Section 19.15. Server-Side Files
└ Section 19.16. Case Study: Shopping Cart
└ Section 19.17. Wrap-Up
└ Section 19.18. Internet and Web Resources
└ Summary
└ Terminology
└ Self-Review Exercises
└ Answers to Self-Review Exercises
└ Exercises
└ Chapter 20. Searching and Sorting
└ Section 20.1. Introduction
└ Section 20.2. Searching Algorithms
└ Section 20.3. Sorting Algorithms
└ Section 20.4. Wrap-Up
└ Summary
└ Terminology
└ Self-Review Exercises
└ Answers to Self-Review Exercises
└ Exercises
└ Chapter 21. Data Structures
└ Section 21.1. Introduction
└ Section 21.2. Self-Referential Classes
└ Section 21.3. Dynamic Memory Allocation and Data Structures
└ Section 21.4. Linked Lists
└ Section 21.5. Stacks
└ Section 21.6. Queues
└ Section 21.7. Trees
└ Section 21.8. Wrap-Up
└ Summary
└ Terminology
└ Self-Review Exercises
└ Answers to Self-Review Exercises

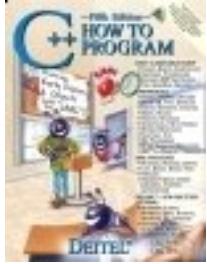
Exercises
Special Section: Building Your Own Compiler
Chapter 22. Bits, Characters, C-Strings and structs
Section 22.1. Introduction
Section 22.2. Structure Definitions
Section 22.3. Initializing Structures
Section 22.4. Using Structures with Functions
Section 22.5. <code>typedef</code>
Section 22.6. Example: High-Performance Card Shuffling and Dealing Simulation
Section 22.7. Bitwise Operators
Section 22.8. Bit Fields
Section 22.9. Character-Handling Library
Section 22.10. Pointer-Based String-Conversion Functions
Section 22.11. Search Functions of the Pointer-Based String-Handling Library
Section 22.12. Memory Functions of the Pointer-Based String-Handling Library
Section 22.13. Wrap-Up
Summary
Terminology
Self-Review Exercises
Answers to Self-Review Exercises
Exercises
Chapter 23. Standard Template Library (STL)
Section 23.1. Introduction to the Standard Template Library (STL)
Section 23.2. Sequence Containers
Section 23.3. Associative Containers
Section 23.4. Container Adapters
Section 23.5. Algorithms
Section 23.6. Class <code>bitset</code>
Section 23.7. Function Objects
Section 23.8. Wrap-Up
Section 23.9. STL Internet and Web Resources
Summary
Terminology
Self-Review Exercises
Answers to Self-Review Exercises
Exercises
Recommended Reading
Chapter 24. Other Topics
Section 24.1. Introduction
Section 24.2. <code>const_cast</code> Operator
Section 24.3. namespaces
Section 24.4. Operator Keywords
Section 24.5. <code>mutable</code> Class Members
Section 24.6. Pointers to Class Members (<code>.*</code> and <code>->*</code>)
Section 24.7. Multiple Inheritance
Section 24.8. Multiple Inheritance and virtual Base Classes
Section 24.9. Wrap-Up
Section 24.10. Closing Remarks
Summary
Terminology
Self-Review Exercises
Answers to Self-Review Exercises
Exercises
Appendix A. Operator Precedence and Associativity Chart
Section A.1. Operator Precedence
Appendix B. ASCII Character Set
Appendix C. Fundamental Types
Appendix D. Number Systems
Section D.1. Introduction
Section D.2. Abbreviating Binary Numbers as Octal and Hexadecimal Numbers

- Section D.3. Converting Octal and Hexadecimal Numbers to Binary Numbers
 - Section D.4. Converting from Binary, Octal or Hexadecimal to Decimal
 - Section D.5. Converting from Decimal to Binary, Octal or Hexadecimal
 - Section D.6. Negative Binary Numbers: Two's Complement Notation
 - Summary
 - Terminology
 - Self-Review Exercises
 - Answers to Self-Review Exercises
 - Exercises
- Appendix E. C Legacy Code Topics
 - Section E.1. Introduction
 - Section E.2. Redirecting Input/Output on UNIX/LINUX/Mac OS X and Windows Systems
 - Section E.3. Variable-Length Argument Lists
 - Section E.4. Using Command-Line Arguments
 - Section E.5. Notes on Compiling Multiple-Source-File Programs
 - Section E.6. Program Termination with exit and atexit
 - Section E.7. The volatile Type Qualifier
 - Section E.8. Suffixes for Integer and Floating-Point Constants
 - Section E.9. Signal Handling
 - Section E.10. Dynamic Memory Allocation with calloc and realloc
 - Section E.11. The Unconditional Branch: goto
 - Section E.12. Unions
 - Section E.13. Linkage Specifications
 - Section E.14. Wrap-Up
 - Summary
 - Terminology
 - Self-Review Exercises
 - Answers to Self-Review Exercises
 - Exercises
- Appendix F. Preprocessor
 - Section F.1. Introduction
 - Section F.2. The #include Preprocessor Directive
 - Section F.3. The #define Preprocessor Directive: Symbolic Constants
 - Section F.4. The #define Preprocessor Directive: Macros
 - Section F.5. Conditional Compilation
 - Section F.6. The #error and #pragma Preprocessor Directives
 - Section F.7. The # and ## Operators
 - Section F.8. Predefined Symbolic Constants
 - Section F.9. Assertions
 - Section F.10. Wrap-Up
 - Summary
 - Terminology
 - Self-Review Exercises
 - Answers to Self-Review Exercises
 - Exercises
- Appendix G. ATM Case Study Code
 - Section G.1. ATM Case Study Implementation
 - Section G.2. Class ATM
 - Section G.3. Class Screen
 - Section G.4. Class Keypad
 - Section G.5. Class CashDispenser
 - Section G.6. Class DepositSlot
 - Section G.7. Class Account
 - Section G.8. Class BankDatabase
 - Section G.9. Class Transaction
 - Section G.10. Class BalanceInquiry
 - Section G.11. Class Withdrawal
 - Section G.12. Class Deposit
 - Section G.13. Test Program ATMCaseStudy.cpp
 - Section G.14. Wrap-Up
- Appendix H. UML 2: Additional Diagram Types

Section H.1. Introduction
Section H.2. Additional Diagram Types
Appendix I. C++ Internet and Web Resources
Section I.1. Resources
Section I.2. Tutorials
Section I.3. FAQs
Section I.4. Visual C++
Section I.5. Newsgroups
Section I.6. Compilers and Development Tools
Section I.7. Standard Template Library
Appendix J. Introduction to XHTML
Section J.1. Introduction
Section J.2. Editing XHTML
Section J.3. First XHTML Example
Section J.4. Headers
Section J.5. Linking
Section J.6. Images

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

NEXT ►



C++ How to Program, Fifth Edition

By H. M. Deitel - Deitel & Associates, Inc., P. J. Deitel - Deitel & Associates, Inc.

Publisher: Prentice Hall

Pub Date: January 05, 2005

Print ISBN-10: 0-13-185757-6

eText ISBN-10: 0-13-186103-4

Print ISBN-13: 978-0-13-185757-5

eText ISBN-13: 978-0-13-186103-9

Pages: 1536

- [Table of Contents](#)
- [Index](#)

Best-selling C++ text significantly revised to include new early objects coverage and new streamlined case studies.

NEXT ►

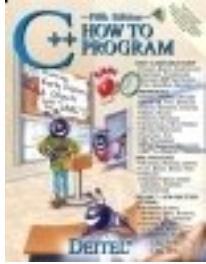
page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[PREV](#)

[NEXT](#)



C++ How to Program, Fifth Edition

By H. M. Deitel - Deitel & Associates, Inc., P. J. Deitel - Deitel & Associates, Inc.

Publisher: Prentice Hall

Pub Date: January 05, 2005

Print ISBN-10: 0-13-185757-6

eText ISBN-10: 0-13-186103-4

Print ISBN-13: 978-0-13-185757-5

eText ISBN-13: 978-0-13-186103-9

Pages: 1536

- Table of Contents
- Index

Copyright

iv

Deitel® Books, Cyber Classrooms, Complete Training Courses and Web-Based Training

ii

Courses published by Prentice Hall

xxiii

Preface

xxiii

Features of C++ How to Program, 5/e

xxvii

Teaching Approach

xxxi

Tour of the Book

xxxi

Object-Oriented Design of an ATM with the UML: A Tour of the Optional Software

xliv

Engineering Case Study

xlvi

Software Bundled with C++ How to Program, 5/e

xlvi

Teaching Resources for C++ How to Program, 5/e

xlix

C++ Multimedia Cyber Classroom, 5/e, Online

xlix

C++ in the Lab

xlix

CourseCompassSM, WebCT™ and Blackboard™

li

PearsonChoices

lii

The Deitel® Buzz Online Free E-mail Newsletter

liii

Acknowledgments

liii

About the Authors

lvii

About Deitel & Associates, Inc.

lvii

Before You Begin

lix

Resources on the CD That Accompanies C++ How to Program, Fifth Edition

lix

Copying and Organizing Files	lix
Copying the Book Examples from the CD	lx
Changing the Read-Only Property of Files	lx
Chapter 1. Introduction to Computers, the Internet and World Wide Web	1
Section 1.1. Introduction	2
Section 1.2. What Is a Computer?	3
Section 1.3. Computer Organization	4
Section 1.4. Early Operating Systems	5
Section 1.5. Personal, Distributed and Client/Server Computing	5
Section 1.6. The Internet and the World Wide Web	6
Section 1.7. Machine Languages, Assembly Languages and High-Level Languages	6
Section 1.8. History of C and C++	8
Section 1.9. C++ Standard Library	8
Section 1.10. History of Java	9
Section 1.11. FORTRAN, COBOL, Pascal and Ada	10
Section 1.12. Basic, Visual Basic, Visual C++, C# and .NET	11
Section 1.13. Key Software Trend: Object Technology	11
Section 1.14. Typical C++ Development Environment	12
Section 1.15. Notes About C++ and C++ How to Program, 5/e	15
Section 1.16. Test-Driving a C++ Application	16
Section 1.17. Software Engineering Case Study: Introduction to Object Technology and the UML (Required)	22
Section 1.18. Wrap-Up	27
Section 1.19. Web Resources	27
Summary	29
Terminology	31
Self-Review Exercises	33
Answers to Self-Review Exercises	34
Exercises	34
Chapter 2. Introduction to C++ Programming	36
Section 2.1. Introduction	37
Section 2.2. First Program in C++: Printing a Line of Text	37
Section 2.3. Modifying Our First C++ Program	41
Section 2.4. Another C++ Program: Adding Integers	42
Section 2.5. Memory Concepts	46
Section 2.6. Arithmetic	48

Section 2.7. Decision Making: Equality and Relational Operators	51
Section 2.8. (Optional) Software Engineering Case Study: Examining the ATM Requirements Document	56
Section 2.9. Wrap-Up	65
Summary	65
Terminology	67
Self-Review Exercises	68
Answers to Self-Review Exercises	69
Exercises	70
Chapter 3. Introduction to Classes and Objects	74
Section 3.1. Introduction	75
Section 3.2. Classes, Objects, Member Functions and Data Members	75
Section 3.3. Overview of the Chapter Examples	77
Section 3.4. Defining a Class with a Member Function	77
Section 3.5. Defining a Member Function with a Parameter	81
Section 3.6. Data Members, set Functions and get Functions	84
Section 3.7. Initializing Objects with Constructors	91
Section 3.8. Placing a Class in a Separate File for Reusability	95
Section 3.9. Separating Interface from Implementation	99
Section 3.10. Validating Data with set Functions	105
Section 3.11. (Optional) Software Engineering Case Study: Identifying the Classes in the ATM Requirements Document	110
Section 3.12. Wrap-Up	118
Summary	118
Terminology	121
Self-Review Exercises	121
Answers to Self-Review Exercises	122
Exercises	122
Chapter 4. Control Statements: Part 1	124
Section 4.1. Introduction	125
Section 4.2. Algorithms	125
Section 4.3. Pseudocode	126
Section 4.4. Control Structures	127
Section 4.5. if Selection Statement	131
Section 4.6. if...else Double-Selection Statement	132
Section 4.7. while Repetition Statement	137
Section 4.8. Formulating Algorithms: Counter-Controlled Repetition	139

—	Section 4.9. Formulating Algorithms: Sentinel-Controlled Repetition	145
—	Section 4.10. Formulating Algorithms: Nested Control Statements	156
—	Section 4.11. Assignment Operators	161
—	Section 4.12. Increment and Decrement Operators	161
—	Section 4.13. (Optional) Software Engineering Case Study: Identifying Class Attributes in the ATM System	165
—	Section 4.14. Wrap-Up	169
—	Summary	170
—	Terminology	172
—	Self-Review Exercises	173
—	Answers to Self-Review Exercises	174
—	Exercises	177
—	Chapter 5. Control Statements: Part 2	185
—	— Section 5.1. Introduction	186
—	— Section 5.2. Essentials of Counter-Controlled Repetition	186
—	— Section 5.3. for Repetition Statement	188
—	— Section 5.4. Examples Using the for Statement	193
—	— Section 5.5. do...while Repetition Statement	197
—	— Section 5.6. switch Multiple-Selection Statement	199
—	— Section 5.7. break and continue Statements	209
—	— Section 5.8. Logical Operators	211
—	— Section 5.9. Confusing Equality (==) and Assignment (=) Operators	216
—	— Section 5.10. Structured Programming Summary	217
—	— Section 5.11. (Optional) Software Engineering Case Study: Identifying Objects' States and Activities in the ATM System	222
—	— Section 5.12. Wrap-Up	226
—	Summary	228
—	Terminology	230
—	Self-Review Exercises	230
—	Answers to Self-Review Exercises	231
—	Exercises	233
—	Chapter 6. Functions and an Introduction to Recursion	238
—	— Section 6.1. Introduction	239
—	— Section 6.2. Program Components in C++	240
—	— Section 6.3. Math Library Functions	241
—	— Section 6.4. Function Definitions with Multiple Parameters	243
—	— Section 6.5. Function Prototypes and Argument Coercion	248

—	Section 6.6. C++ Standard Library Header Files	250
—	Section 6.7. Case Study: Random Number Generation	252
—	Section 6.8. Case Study: Game of Chance and Introducing enum	258
—	Section 6.9. Storage Classes	262
—	Section 6.10. Scope Rules	265
—	Section 6.11. Function Call Stack and Activation Records	268
—	Section 6.12. Functions with Empty Parameter Lists	272
—	Section 6.13. Inline Functions	273
—	Section 6.14. References and Reference Parameters	275
—	Section 6.15. Default Arguments	280
—	Section 6.16. Unary Scope Resolution Operator	282
—	Section 6.17. Function Overloading	283
—	Section 6.18. Function Templates	286
—	Section 6.19. Recursion	288
—	Section 6.20. Example Using Recursion: Fibonacci Series	292
—	Section 6.21. Recursion vs. Iteration	295
—	Section 6.22. (Optional) Software Engineering Case Study: Identifying Class Operations in the ATM System	298
—	Section 6.23. Wrap-Up	305
—	Summary	306
—	Terminology	309
—	Self-Review Exercises	311
—	Answers to Self-Review Exercises	313
—	Exercises	316
—	Chapter 7. Arrays and Vectors	326
—	Section 7.1. Introduction	327
—	Section 7.2. Arrays	328
—	Section 7.3. Declaring Arrays	329
—	Section 7.4. Examples Using Arrays	330
—	Section 7.5. Passing Arrays to Functions	346
—	Section 7.6. Case Study: Class GradeBook Using an Array to Store Grades	351
—	Section 7.7. Searching Arrays with Linear Search	358
—	Section 7.8. Sorting Arrays with Insertion Sort	359
—	Section 7.9. Multidimensional Arrays	362
—	Section 7.10. Case Study: Class GradeBook Using a Two-Dimensional Array	365
—	Section 7.11. Introduction to C++ Standard Library Class Template vector	372

Section 7.12. (Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM System	377
Section 7.13. Wrap-Up	385
Summary	385
Terminology	387
Self-Review Exercises	388
Answers to Self-Review Exercises	389
Exercises	390
Recursion Exercises	400
vector Exercises	400
Chapter 8. Pointers and Pointer-Based Strings	401
Section 8.1. Introduction	402
Section 8.2. Pointer Variable Declarations and Initialization	403
Section 8.3. Pointer Operators	404
Section 8.4. Passing Arguments to Functions by Reference with Pointers	407
Section 8.5. Using const with Pointers	411
Section 8.6. Selection Sort Using Pass-by-Reference	418
Section 8.7. sizeof Operators	421
Section 8.8. Pointer Expressions and Pointer Arithmetic	424
Section 8.9. Relationship Between Pointers and Arrays	427
Section 8.10. Arrays of Pointers	431
Section 8.11. Case Study: Card Shuffling and Dealing Simulation	432
Section 8.12. Function Pointers	438
Section 8.13. Introduction to Pointer-Based String Processing	443
Section 8.14. Wrap-Up	454
Summary	455
Terminology	456
Self-Review Exercises	457
Answers to Self-Review Exercises	459
Exercises	461
Special Section: Building Your Own Computer	464
More Pointer Exercises	469
String-Manipulation Exercises	474
Special Section: Advanced String-Manipulation Exercises	475
A Challenging String-Manipulation Project	479
Chapter 9. Classes: A Deeper Look, Part 1	480

—Section 9.1. Introduction	481
—Section 9.2. Time Class Case Study	482
—Section 9.3. Class Scope and Accessing Class Members	487
—Section 9.4. Separating Interface from Implementation	489
—Section 9.5. Access Functions and Utility Functions	491
—Section 9.6. Time Class Case Study: Constructors with Default Arguments	493
—Section 9.7. Destructors	499
—Section 9.8. When Constructors and Destructors Are Called	500
Section 9.9. Time Class Case Study: A Subtle TrapReturning a Reference to a private Data Member	503
—Section 9.10. Default Memberwise Assignment	506
—Section 9.11. Software Reusability	508
Section 9.12. (Optional) Software Engineering Case Study: Starting to Program the Classes of the ATM System	509
—Section 9.13. Wrap-Up	516
—Summary	517
—Terminology	519
—Self-Review Exercises	519
—Answers to Self-Review Exercises	520
—Exercises	520
—Chapter 10. Classes: A Deeper Look, Part 2	523
Section 10.1. Introduction	524
Section 10.2. const (Constant) Objects and const Member Functions	524
Section 10.3. Composition: Objects as Members of Classes	534
Section 10.4. friend Functions and friend Classes	541
Section 10.5. Using the this Pointer	545
Section 10.6. Dynamic Memory Management with Operators new and delete	550
Section 10.7. static Class Members	552
Section 10.8. Data Abstraction and Information Hiding	558
Section 10.9. Container Classes and Iterators	561
Section 10.10. Proxy Classes	562
Section 10.11. Wrap-Up	565
Summary	566
Terminology	567
Self-Review Exercises	568
Answers to Self-Review Exercises	569
Exercises	569

<u>Chapter 11. Operator Overloading; String and Array Objects</u>	571
<u>Section 11.1. Introduction</u>	572
<u>Section 11.2. Fundamentals of Operator Overloading</u>	573
<u>Section 11.3. Restrictions on Operator Overloading</u>	574
<u>Section 11.4. Operator Functions as Class Members vs. Global Functions</u>	576
<u>Section 11.5. Overloading Stream Insertion and Stream Extraction Operators</u>	577
<u>Section 11.6. Overloading Unary Operators</u>	581
<u>Section 11.7. Overloading Binary Operators</u>	581
<u>Section 11.8. Case Study: Array Class</u>	582
<u>Section 11.9. Converting between Types</u>	594
<u>Section 11.10. Case Study: String Class</u>	595
<u>Section 11.11. Overloading ++ and --</u>	607
<u>Section 11.12. Case Study: A Date Class</u>	609
<u>Section 11.13. Standard Library Class string</u>	613
<u>Section 11.14. explicit Constructors</u>	617
<u>Section 11.15. Wrap-Up</u>	621
<u>Summary</u>	621
<u>Terminology</u>	624
<u>Self-Review Exercises</u>	624
<u>Answers to Self-Review Exercises</u>	625
<u>Exercises</u>	625
<u>Chapter 12. Object-Oriented Programming: Inheritance</u>	633
<u>Section 12.1. Introduction</u>	634
<u>Section 12.2. Base Classes and Derived Classes</u>	635
<u>Section 12.3. protected Members</u>	638
<u>Section 12.4. Relationship between Base Classes and Derived Classes</u>	638
<u>Section 12.5. Constructors and Destructors in Derived Classes</u>	670
<u>Section 12.6. public, protected and private Inheritance</u>	678
<u>Section 12.7. Software Engineering with Inheritance</u>	678
<u>Section 12.8. Wrap-Up</u>	680
<u>Summary</u>	681
<u>Terminology</u>	682
<u>Self-Review Exercises</u>	682
<u>Answers to Self-Review Exercises</u>	683
<u>Exercises</u>	683
<u>Chapter 13. Object-Oriented Programming: Polymorphism</u>	686

—Section 13.1. Introduction	687
—Section 13.2. Polymorphism Examples	689
—Section 13.3. Relationships Among Objects in an Inheritance Hierarchy	690
—Section 13.4. Type Fields and switch Statements	707
—Section 13.5. Abstract Classes and Pure virtual Functions	708
—Section 13.6. Case Study: Payroll System Using Polymorphism	710
Section 13.7. (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"	728
Section 13.8. Case Study: Payroll System Using Polymorphism and Run-Time Type Information with Downcasting, dynamic_cast, typeid and type_info	732
—Section 13.9. Virtual Destructors	735
Section 13.10. (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System	736
—Section 13.11. Wrap-Up	744
—Summary	744
—Terminology	746
—Self-Review Exercises	746
—Answers to Self-Review Exercises	747
—Exercises	747
—Chapter 14. Templates	749
—Section 14.1. Introduction	750
—Section 14.2. Function Templates	751
—Section 14.3. Overloading Function Templates	754
—Section 14.4. Class Templates	754
—Section 14.5. Nontype Parameters and Default Types for Class Templates	761
—Section 14.6. Notes on Templates and Inheritance	762
—Section 14.7. Notes on Templates and Friends	762
—Section 14.8. Notes on Templates and static Members	763
—Section 14.9. Wrap-Up	764
—Summary	764
—Terminology	765
—Self-Review Exercises	766
—Answers to Self-Review Exercises	766
—Exercises	766
—Chapter 15. Stream Input/Output	769
—Section 15.1. Introduction	770
—Section 15.2. Streams	771

—Section 15.3. Stream Output	775
—Section 15.4. Stream Input	776
—Section 15.5. Unformatted I/O using read, write and gcount	780
—Section 15.6. Introduction to Stream Manipulators	781
—Section 15.7. Stream Format States and Stream Manipulators	787
—Section 15.8. Stream Error States	797
—Section 15.9. Tying an Output Stream to an Input Stream	800
—Section 15.10. Wrap-Up	800
—Summary	800
—Terminology	803
—Self-Review Exercises	804
—Answers to Self-Review Exercises	806
—Exercises	808
—Chapter 16. Exception Handling	810
—Section 16.1. Introduction	811
—Section 16.2. Exception-Handling Overview	812
—Section 16.3. Example: Handling an Attempt to Divide by Zero	812
—Section 16.4. When to Use Exception Handling	819
—Section 16.5. Rethrowing an Exception	820
—Section 16.6. Exception Specifications	821
—Section 16.7. Processing Unexpected Exceptions	822
—Section 16.8. Stack Unwinding	823
—Section 16.9. Constructors, Destructors and Exception Handling	824
—Section 16.10. Exceptions and Inheritance	825
—Section 16.11. Processing new Failures	825
—Section 16.12. Class auto_ptr and Dynamic Memory Allocation	829
—Section 16.13. Standard Library Exception Hierarchy	832
—Section 16.14. Other Error-Handling Techniques	834
—Section 16.15. Wrap-Up	834
—Summary	835
—Terminology	837
—Self-Review Exercises	838
—Answers to Self-Review Exercises	838
—Exercises	839
—Chapter 17. File Processing	841
—Section 17.1. Introduction	842

—Section 17.2. The Data Hierarchy	842
—Section 17.3. Files and Streams	844
—Section 17.4. Creating a Sequential File	845
—Section 17.5. Reading Data from a Sequential File	849
—Section 17.6. Updating Sequential Files	856
—Section 17.7. Random-Access Files	856
—Section 17.8. Creating a Random-Access File	857
—Section 17.9. Writing Data Randomly to a Random-Access File	862
—Section 17.10. Reading from a Random-Access File Sequentially	864
—Section 17.11. Case Study: A Transaction-Processing Program	867
—Section 17.12. Input/Output of Objects	874
—Section 17.13. Wrap-Up	874
—Summary	874
—Terminology	876
—Self-Review Exercises	876
—Answers to Self-Review Exercises	877
—Exercises	878
—Chapter 18. Class string and String Stream Processing	883
—Section 18.1. Introduction	884
—Section 18.2. string Assignment and Concatenation	885
—Section 18.3. Comparing strings	887
—Section 18.4. Substrings	890
—Section 18.5. Swapping strings	891
—Section 18.6. string Characteristics	892
—Section 18.7. Finding Strings and Characters in a string	894
—Section 18.8. Replacing Characters in a string	896
—Section 18.9. Inserting Characters into a string	898
—Section 18.10. Conversion to C-Style Pointer-Based char * Strings	899
—Section 18.11. Iterators	901
—Section 18.12. String Stream Processing	902
—Section 18.13. Wrap-Up	905
—Summary	906
—Terminology	907
—Self-Review Exercises	907
—Answers to Self-Review Exercises	908
—Exercises	908

—	Chapter 19. Web Programming	911
—	Section 19.1. Introduction	912
—	Section 19.2. HTTP Request Types	913
—	Section 19.3. Multitier Architecture	914
—	Section 19.4. Accessing Web Servers	915
—	Section 19.5. Apache HTTP Server	916
—	Section 19.6. Requesting XHTML Documents	917
—	Section 19.7. Introduction to CGI	917
—	Section 19.8. Simple HTTP Transactions	918
—	Section 19.9. Simple CGI Scripts	920
—	Section 19.10. Sending Input to a CGI Script	928
—	Section 19.11. Using XHTML Forms to Send Input	928
—	Section 19.12. Other Headers	938
—	Section 19.13. Case Study: An Interactive Web Page	939
—	Section 19.14. Cookies	943
—	Section 19.15. Server-Side Files	949
—	Section 19.16. Case Study: Shopping Cart	954
—	Section 19.17. Wrap-Up	969
—	Section 19.18. Internet and Web Resources	969
—	Summary	970
—	Terminology	972
—	Self-Review Exercises	973
—	Answers to Self-Review Exercises	973
—	Exercises	974
—	Chapter 20. Searching and Sorting	975
—	Section 20.1. Introduction	976
—	Section 20.2. Searching Algorithms	976
—	Section 20.3. Sorting Algorithms	982
—	Section 20.4. Wrap-Up	992
—	Summary	992
—	Terminology	994
—	Self-Review Exercises	994
—	Answers to Self-Review Exercises	995
—	Exercises	995
—	Chapter 21. Data Structures	998
—	Section 21.1. Introduction	999

—Section 21.2. Self-Referential Classes	1000
—Section 21.3. Dynamic Memory Allocation and Data Structures	1001
—Section 21.4. Linked Lists	1001
—Section 21.5. Stacks	1016
—Section 21.6. Queues	1021
—Section 21.7. Trees	1025
—Section 21.8. Wrap-Up	1033
—Summary	1034
—Terminology	1035
—Self-Review Exercises	1036
—Answers to Self-Review Exercises	1037
—Exercises	1038
—Special Section: Building Your Own Compiler	1043
—Chapter 22. Bits, Characters, C-Strings and structs	1057
—Section 22.1. Introduction	1058
—Section 22.2. Structure Definitions	1058
—Section 22.3. Initializing Structures	1061
—Section 22.4. Using Structures with Functions	1061
—Section 22.5. typedef	1061
—Section 22.6. Example: High-Performance Card Shuffling and Dealing Simulation	1062
—Section 22.7. Bitwise Operators	1065
—Section 22.8. Bit Fields	1074
—Section 22.9. Character-Handling Library	1078
—Section 22.10. Pointer-Based String-Conversion Functions	1084
—Section 22.11. Search Functions of the Pointer-Based String-Handling Library	1089
—Section 22.12. Memory Functions of the Pointer-Based String-Handling Library	1094
—Section 22.13. Wrap-Up	1099
—Summary	1099
—Terminology	1101
—Self-Review Exercises	1102
—Answers to Self-Review Exercises	1103
—Exercises	1104
—Chapter 23. Standard Template Library (STL)	1110
—Section 23.1. Introduction to the Standard Template Library (STL)	1112
—Section 23.2. Sequence Containers	1124
—Section 23.3. Associative Containers	1138

—Section 23.4. Container Adapters	1147
—Section 23.5. Algorithms	1152
—Section 23.6. Class bitset	1183
—Section 23.7. Function Objects	1187
—Section 23.8. Wrap-Up	1190
—Section 23.9. STL Internet and Web Resources	1191
—Summary	1192
—Terminology	1196
—Self-Review Exercises	1197
—Answers to Self-Review Exercises	1198
—Exercises	1198
—Recommended Reading	1199
—Chapter 24. Other Topics	1200
—Section 24.1. Introduction	1201
—Section 24.2. const_cast Operator	1201
—Section 24.3. namespaces	1203
—Section 24.4. Operator Keywords	1207
—Section 24.5. mutable Class Members	1209
—Section 24.6. Pointers to Class Members (.* and ->*)	1211
—Section 24.7. Multiple Inheritance	1213
—Section 24.8. Multiple Inheritance and virtual Base Classes	1218
—Section 24.9. Wrap-Up	1222
—Section 24.10. Closing Remarks	1223
—Summary	1223
—Terminology	1225
—Self-Review Exercises	1225
—Answers to Self-Review Exercises	1226
—Exercises	1226
—Appendix A. Operator Precedence and Associativity Chart	1228
—Section A.1. Operator Precedence	1228
—Appendix B. ASCII Character Set	1231
—Appendix C. Fundamental Types	1232
—Appendix D. Number Systems	1234
—Section D.1. Introduction	1235
—Section D.2. Abbreviating Binary Numbers as Octal and Hexadecimal Numbers	1238
—Section D.3. Converting Octal and Hexadecimal Numbers to Binary Numbers	1239

—	Section D.4. Converting from Binary, Octal or Hexadecimal to Decimal	1239
—	Section D.5. Converting from Decimal to Binary, Octal or Hexadecimal	1240
—	Section D.6. Negative Binary Numbers: Two's Complement Notation	1242
—	Summary	1243
—	Terminology	1243
—	Self-Review Exercises	1244
—	Answers to Self-Review Exercises	1245
—	Exercises	1246
—	Appendix E. C Legacy Code Topics	1247
—	Section E.1. Introduction	1248
—	Section E.2. Redirecting Input/Output on UNIX/LINUX/Mac OS X and Windows Systems	1248
—	Section E.3. Variable-Length Argument Lists	1249
—	Section E.4. Using Command-Line Arguments	1252
—	Section E.5. Notes on Compiling Multiple-Source-File Programs	1253
—	Section E.6. Program Termination with exit and atexit	1255
—	Section E.7. The volatile Type Qualifier	1257
—	Section E.8. Suffixes for Integer and Floating-Point Constants	1257
—	Section E.9. Signal Handling	1257
—	Section E.10. Dynamic Memory Allocation with calloc and realloc	1260
—	Section E.11. The Unconditional Branch: goto	1261
—	Section E.12. Unions	1262
—	Section E.13. Linkage Specifications	1265
—	Section E.14. Wrap-Up	1266
—	Summary	1267
—	Terminology	1269
—	Self-Review Exercises	1269
—	Answers to Self-Review Exercises	1270
—	Exercises	1270
—	Appendix F. Preprocessor	1272
—	Section F.1. Introduction	1273
—	Section F.2. The #include Preprocessor Directive	1273
—	Section F.3. The #define Preprocessor Directive: Symbolic Constants	1274
—	Section F.4. The #define Preprocessor Directive: Macros	1275
—	Section F.5. Conditional Compilation	1277
—	Section F.6. The #error and #pragma Preprocessor Directives	1278
—	Section F.7. The # and ## Operators	1278

—	Section F.8. Predefined Symbolic Constants	1279
—	Section F.9. Assertions	1279
—	Section F.10. Wrap-Up	1280
—	Summary	1280
—	Terminology	1281
—	Self-Review Exercises	1281
—	Answers to Self-Review Exercises	1282
—	Exercises	1283
—	Appendix G. ATM Case Study Code	1285
—	Section G.1. ATM Case Study Implementation	1285
—	Section G.2. Class ATM	1286
—	Section G.3. Class Screen	1293
—	Section G.4. Class Keypad	1294
—	Section G.5. Class CashDispenser	1295
—	Section G.6. Class DepositSlot	1297
—	Section G.7. Class Account	1298
—	Section G.8. Class BankDatabase	1300
—	Section G.9. Class Transaction	1304
—	Section G.10. Class BalanceInquiry	1306
—	Section G.11. Class Withdrawal	1308
—	Section G.12. Class Deposit	1313
—	Section G.13. Test Program ATMCaseStudy.cpp	1316
—	Section G.14. Wrap-Up	1317
—	Appendix H. UML 2: Additional Diagram Types	1318
—	Section H.1. Introduction	1318
—	Section H.2. Additional Diagram Types	1318
—	Appendix I. C++ Internet and Web Resources	1320
—	Section I.1. Resources	1320
—	Section I.2. Tutorials	1322
—	Section I.3. FAQs	1322
—	Section I.4. Visual C++	1322
—	Section I.5. Newsgroups	1323
—	Section I.6. Compilers and Development Tools	1323
—	Section I.7. Standard Template Library	1324
—	Appendix J. Introduction to XHTML	1325
—	Section J.1. Introduction	1326

—Section J.2. Editing XHTML	1326
—Section J.3. First XHTML Example	1327
—Section J.4. Headers	1330
—Section J.5. Linking	1331
—Section J.6. Images	1333
—Section J.7. Special Characters and More Line Breaks	1338
—Section J.8. Unordered Lists	1340
—Section J.9. Nested and Ordered Lists	1340
—Section J.10. Basic XHTML Tables	1341
—Section J.11. Intermediate XHTML Tables and Formatting	1346
—Section J.12. Basic XHTML Forms	1349
—Section J.13. More Complex XHTML Forms	1352
—Section J.14. Internet and World Wide Web Resources	1359
—Summary	1359
—Terminology	1361
—Appendix K. XHTML Special Characters	1363
—Appendix L. Using the Visual Studio .NET Debugger	1364
—Section L.1. Introduction	1365
—Section L.2. Breakpoints and the Continue Command	1365
—Section L.3. The Locals and Watch Windows	1371
—Section L.4. Controlling Execution Using the Step Into, Step Over, Step Out and Continue Commands	1374
—Section L.5. The Autos Window	1377
—Section L.6. Wrap-Up	1378
—Summary	1379
—Terminology	1380
—Self-Review Exercises	1380
—Answers to Self-Review Exercises	1380
—Appendix M. Using the GNU C++ Debugger	1381
—Section M.1. Introduction	1382
—Section M.2. Breakpoints and the run, stop, continue and print Commands	1382
—Section M.3. The print and set Commands	1389
—Section M.4. Controlling Execution Using the step, finish and next Commands	1391
—Section M.5. The watch Command	1393
—Section M.6. Wrap-Up	1396
—Summary	1397

Terminology	1398
Self-Review Exercises	1398
Answers to Self-Review Exercises	1398
Bibliography	1399
End User License Agreements	EULA-1
Prentice Hall License Agreement and Limited Warranty	EULA-1
License Agreement and Limited Warranty	EULA-3
Using the CD-ROM	EULA-3
Contents of the CD-ROM	EULA-3
Software and Hardware System Requirements	EULA-3
Index	

 PREVNEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page vi]

Trademarks

Borland and C++ Builder are trademarks or registered trademarks of Borland.

Cygwin is a trademark and copyrighted work of Red Hat, Inc. in the United States and other countries.

Dive Into is a registered trademark of Deitel & Associates, Inc.

GNU is a trademark of the Free Software Foundation.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. Pearson Education is independent of Sun Microsystems, Inc.

Linux is a registered trademark of Linus Torvalds.

Microsoft, Microsoft® Internet Explorer and the Windows logo are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Netscape browser window © 2004 Netscape Communications Corporation. Used with permission.
Netscape Communications has not authorized, sponsored, endorsed, or approved this publication and is not responsible for its content.

Object Management Group, OMG, Unified Modeling Language and UML are trademarks of Object Management Group, Inc.

[Page vii]

Dedication

To:

Stephen Clamage

Chairman of the J16 committee, "Programming Language C++" that is responsible for the C++ standard; Senior Staff Engineer, Sun Microsystems, Inc., Software Division.

Don Kostuch

Independent Consultant

and Mike Miller

Former Vice Chairman and Core Language Working Group Chairman of the J16 committee,
"Programming Language C++;" Software Design Engineer, Edison Design Group, Inc.

For your mentorship, friendship, and tireless devotion to insisting that we "get it right" and helping us do so.

It is a privilege to work with such consummate C++ professionals.

Harvey M. Deitel and Paul J. Deitel

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page iii]

Multimedia Cyber Classroom and Web-Based Training Series

C++ Multimedia Cyber Classroom, 5/E

C# Multimedia Cyber Classroom

e-Business and e-Commerce Multimedia Cyber Classroom

Internet and World Wide Web Multimedia Cyber Classroom, 2/E

Java™ Multimedia Cyber Classroom, 6/E

Perl Multimedia Cyber Classroom

Python Multimedia Cyber Classroom

Visual Basic® 6 Multimedia Cyber Classroom

Visual Basic® .NET Multimedia Cyber Classroom, 2/E

Wireless Internet & Mobile Business Programming Multimedia Cyber Classroom

XML Multimedia Cyber Classroom

The Complete Training Course Series

The Complete C++ Training Course, 4/E

The Complete C# Training Course

The Complete e-Business and

e-Commerce Programming Training Course

The Complete Internet and World Wide Web Programming Training Course, 2/E

The Complete Java™ 2 Training Course, 5/E

The Complete Perl Training Course

The Complete Python Training Course

The Complete Visual Basic® 6 Training Course

The Complete Visual Basic® .NET Training Course, 2/E

The Complete Wireless Internet & Mobile Business Programming Training Course

The Complete XML Programming Training Course

To follow the Deitel publishing program, please register at:

www.deitel.com/newsletter/subscribe.html

for the free DEITEL® BUZZ ONLINE e-mail newsletter.

To communicate with the authors, send e-mail to:

deitel@deitel.com

For information on corporate on-site seminars offered by Deitel & Associates, Inc. worldwide, visit:

www.deitel.com or write to deitel@deitel.com

For continuing updates on Prentice Hall/Deitel publications visit:

www.deitel.com,
www.prenhall.com/deitel or
www.InformIT.com/deitel

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page xxiii]

Preface

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

Galen

Welcome to C++ and C++ How to Program, Fifth Edition! C++ is a world-class programming language for developing industrial-strength, high-performance computer applications. We believe that this book and its support materials have everything instructors and students need for an informative, interesting, challenging and entertaining C++ educational experience. In this Preface, we overview the many new features of C++ How to Program, 5/e. The Tour of the Book section of the Preface gives instructors, students and professionals a sense of C++ How to Program, 5/e's coverage of C++ and object-oriented programming. We also overview various conventions used in the book, such as syntax coloring the code examples, "code washing" and code highlighting. We provide information about free compilers that you can find on the Web. We also discuss the comprehensive suite of educational materials that help instructors maximize their students' learning experience, including the Instructor's Resource CD, PowerPoint® Slide lecture notes, course management systems, SafariX (Pearson Education's WebBook publications) and more.

[◀ PREV](#)[NEXT ▶](#)

[page footer](#)

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page xxiv]

- Integrated Case Studies. We have added several case studies spanning multiple sections and chapters that often build on a class introduced earlier in the book to demonstrate new programming concepts later in the book. These case studies include the development of the GradeBook class in [Chapters 37](#), the Time class in several sections of [Chapters 910](#), the Employee class in [Chapters 1213](#), and the optional OOD/UML ATM case study in [chapters 1-7, 9, 13](#) and [Appendix G](#).
- Integrated GradeBook Case Study. We added a new GradeBook case study to reinforce our early classes presentation. It uses classes and objects in [Chapters 37](#) to incrementally build a GradeBook class that represents an instructor's grade book and performs various calculations based on a set of student grades, such as calculating the average grade, finding the maximum and minimum and printing a bar chart.
- Unified Modeling Language™ 2.0 (UML 2.0) Introducing the UML 2.0. The Unified Modeling Language (UML) has become the preferred graphical modeling language for designers of object-oriented systems. All the UML diagrams in the book comply with the new UML 2.0 specification. 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 make especially heavy use of the UML in the optional OOD/UML ATM case study
- Optional OOD/UML ATM Case Study. We replaced the optional elevator simulator case study from the previous edition with a new optional OOD/UML automated teller machine (ATM) case study in the Software Engineering Case Study sections of [Chapters 17, 9](#) and [13](#). The new case study is simpler, smaller, more "real world" and more appropriate for first and second programming courses. The nine case study sections present a carefully paced introduction to object-oriented design using the UML. We introduce a concise, simplified subset of the UML 2.0, then guide the reader through a first design experience intended for the novice object-oriented designer/programmer. Our goal in this case study is to help students develop an object-oriented design to complement the object-oriented programming concepts they begin learning in [Chapter 1](#) and implementing in [Chapter 3](#). The case study was reviewed by a distinguished team of OOD/ UML academic and industry professionals. The case study is not an exercise; rather, it is 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.
- Compilation and Linking Process for Multiple-Source-File Programs. [Chapter 3](#) includes a detailed diagram and discussion of the compilation and linking process that produces an executable application.
- 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.

[Page xxv]

- Early Introduction of C++ Standard Library `string` and `vector` Objects. The `string` and `vector`

classes are used to make earlier examples more object-oriented.

- Class `string`. We use class `string` instead of C-like pointer-based `char *` strings for most string manipulations throughout the book. We continue to include discussions of `char *` strings in [Chapter 8](#), [10](#), [11](#) and [22](#) to give students practice with pointer manipulations, to illustrate dynamic memory allocation with `new` and `delete`, to build our own `String` class, and to prepare students for assignments in industry where they will work 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 students for working with C and C++ legacy code in industry and to use as a basis for building our own customized `Array` class in [Chapter 11](#), Operating Overloading.
- Tuned Treatment of Inheritance and Polymorphism. [Chapters 12](#)[13](#) have been carefully tuned, making the treatment of inheritance and polymorphism clearer and more accessible for students who are new to OOP. An `Employee` hierarchy replaces the `Point/Circle/Cylinder` hierarchy used in prior editions to introduce inheritance and polymorphism. The new hierarchy is more natural.
- 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 students a solid understanding of how these capabilities really work. More importantly, it helps students appreciate the overhead of polymorphism in terms of additional memory consumption and processor time. This helps students determine when to use polymorphism and when to avoid it.
- Web Programming. [Chapter 19](#), Web Programming, has everything readers need to begin developing their own Web-based applications that will run on the Internet! Students will learn how to build so-called n-tier applications, in which the functionality provided by each tier can be distributed to separate computers across the Internet or executed on the same computer. Using the popular Apache HTTP server (which is available free for download from www.apache.org) we present the CGI (common Gateway Interface) protocol and discuss how CGI allows a Web server to communicate with the top tier (e.g., a Web browser running on the user's computer) and CGI scripts (i.e., our C++ programs) executing on a remote system. The chapter examples conclude with an e-business case study of an online bookstore that allows users to add books to an electronic shopping cart.
- Standard Template Library (STL). This might be one of the most important topic in the book in terms of your appreciation 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 23](#) introduces the STL and discusses its three key components containers, iterators and algorithms. We show that using STL components provides tremendous expressive power and can reduce many lines of code to a single statement.

[Page xxvi]

- XHTML. The World Wide Web Consortium (W3C) has declared HyperText Markup Language (HTML) to be a legacy technology that will undergo no further development. HTML is being replaced by the Extensible HyperText Markup Language (XHTML) an XML-based technology that rapidly is becoming the standard for describing Web content. We use XHTML in [Chapter 19](#), Web Programming; [Appendix J](#) and [Appendix K](#) introduce XHTML.
- ANSI/ISO C++ Standard Compliance. We have audited our presentation against the most recent ANSI/ISO C++ standard document for completeness and accuracy. [Note: If you need additional technical

details on C++, you may want to read the C++ standard document. An electronic PDF copy of the C++ standard document, number INCITS/ISO/IEC 14882-2003, is available for \$18 at webstore.ansi.org/ansidocstore/default.asp.]

- New Debugger Appendices. We include two new Using the Debugger appendices [Appendix L](#), Using the Visual Studio .NET Debugger, and [Appendix M](#), Using the GNU C++ Debugger.
- New Interior Design. Working with the creative services team at Prentice Hall, we redesigned the interior styles for our How to Program Series. The new fonts are easier on the eyes and the new art package is more appropriate for the more detailed illustrations. We now place the defining occurrence of each key term both in the text and in the index in blue, bold style 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).
- Syntax Coloring. We syntax color all the C++ code, which is consistent with most C++ integrated development environments and code editors. This greatly improves code readability an especially important goal, given that this book contains 17,292 lines of code. Our syntax-coloring conventions are as follows:

comments appear in green
 keywords appear in dark blue
 constants and literal values appear in light blue
 errors appear in red
 all other code appears in black

- Code Highlighting. Extensive code highlighting makes it easy for readers to locate each program's new features and helps students review the material rapidly when preparing for exams or labs.
- "Code washing." This is our term for using extensive and meaningful comments, using meaningful identifiers, applying uniform indentation conventions, aligning curly braces vertically, using a // end... comment on every line with a right curly brace and using vertical spacing to highlight significant program units such as control statements and functions. This process results in programs that are easy to read and self-documenting. We have extensively "code washed" all of the source-code programs in both the text and the book's ancillaries. We have worked hard to make our code exemplary.

[Page xxvii]

- Code Testing on Multiple Platforms. We tested the code examples on various popular C++ platforms. For the most part, all of the book's examples port easily to all popular ANSI/ISO standard-compliant compilers. We will post any problems at www.deitel.com/books/cpphtp5/index.html.
- 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.
- Large Review Team. The book has been carefully scrutinized by a team of 30 distinguished academic and industry reviewers (listed later in the Preface).
- Free Web-Based Cyber Classroom. We've converted our popular interactive multimedia version of the text (which we call a Cyber Classroom) from a for-sale, CD-based product to a free online supplement, available with new books purchased from Prentice Hall for fall 2005 classes.
- Free Student Solutions Manual. We've converted our Student Solutions Manual, which contains solutions to approximately half of the exercises, from a for-sale softcover book to a free online supplement, available with new books purchased from Prentice Hall for fall 2005 classes.
- Free Lab Manual. We've converted our Lab Manual, C++ in the Lab, from a for-sale softcover book to a free online supplement included with the Cyber Classroom, available with new books purchased from

Prentice Hall for fall 2005 classes.

As you read this book, if you have questions, send an e-mail to deitel&deitel.com; we will respond promptly. Please visit our Web site, www.deitel.com and be sure to sign up for the free DEITEL® Buzz Online e-mail newsletter at www.deitel.com/newsletter/subscribe.html for updates to this book and the latest information on C++. We also use the Web site and the newsletter to keep our readers and industry clients informed of the latest news on Deitel publications and services. Please check the following Web site regularly for errata, updates regarding the C++ software, free downloads and other resources:

www.deitel.com/books/cpphttp5/index.html

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page xxviii]

World Wide Web Access

All the source-code examples for C++ How to Program, 5/e (and our other publications) are available on the Internet as downloads from

www.deitel.com

Registration is quick and easy, and the downloads are free. We suggest downloading all the examples (or copying them from the CD included in the back of this book), then running each program as you read the corresponding text. Making changes to the examples and immediately seeing the effects of those changes is a great way to enhance your C++ learning experience.

Objectives

Each chapter begins with a statement of objectives. This lets students know what to expect and gives them an opportunity, after reading the chapter, to determine if they have met these objectives. This is a confidence builder and a source of positive reinforcement.

Quotations

The learning objectives are followed by quotations. Some are humorous, some philosophical and some offer interesting insights. We hope that you will enjoy relating the quotations to the chapter material. Many of the quotations are worth a second look after reading the chapter.

Outline

The chapter outline helps students approach the material in a top-down fashion, so they can anticipate what is to come, and set a comfortable and effective learning pace.

17,292 Lines of Syntax-Colored Code in 260 Example Programs with Program Inputs and Outputs)

Our LIVE-CODE programs range in size from just a few lines of code to more substantial examples. Each program is followed by a window containing the input/output dialogue produced when the program is run, so students can confirm that the programs run as expected. Relating outputs to the program statements that produce them is an excellent way to learn and to reinforce concepts. Our programs demonstrate the diverse features of C++. The code is line-numbered and syntax colored with C++ keywords, comments and other program text each appearing in different colors. This facilitates reading the code students will

especially appreciate the syntax coloring when they read the larger programs.

735 Illustrations/Figures

An abundance of charts, tables, line drawings, programs and program outputs is included. We model the flow of control in control statements with UML activity diagrams. UML class diagrams model the data members, constructors and member functions of classes. We use additional types of UML diagrams throughout our optional OOD/UML ATM Software Engineering Case Study.

[Page xxix]

571 Programming Tips

We include programming tips to help students focus on important aspects of program development. We highlight these tips in the form of Good Programming Practices, Common Programming Errors, Performance Tips, Portability Tips, Software Engineering Observations and Error-Prevention Tips. These tips and practices represent the best we have gleaned from a combined six decades of programming and teaching experience. One of our students, a mathematics major, told us that she feels this approach is like the highlighting of axioms, theorems, lemmas and corollaries in mathematics booksit provides a basis on which to build good software.

Good Programming Practices



Good Programming Practices are tips for writing clear programs. These techniques help students produce programs that are more readable, self-documenting and easier to maintain.

Common Programming Errors



Students who are new to programming (or a programming language) tend to make certain errors frequently. Focusing on these Common Programming Errors reduces the likelihood that students will make the same mistakes and shortens long lines outside instructors' offices during office hours!

Performance Tips



In our experience, teaching students to write clear and understandable programs is by far the most important goal for a first programming course. But students want to write the programs that run the fastest, use the least memory, require the smallest number of keystrokes or dazzle in other nifty ways. Students really care about performance. They want to know what they can do to "turbo charge" their programs. So we highlight opportunities for improving program performance making programs run faster or minimizing the amount of memory that they occupy.

Portability Tips



Software development is a complex and expensive activity. Organizations that develop software must often produce versions customized to a variety of computers and operating systems. So there is a strong emphasis today on portability, i.e., on producing software that will run on a variety of computer systems with few, if any, changes. Some programmers assume that if they implement an application in standard C++, the application will be portable. This simply is not the case. Achieving portability requires careful and cautious design. There are many pitfalls. We include Portability Tips to help students write portable code and to provide insights on how C++ achieves its high degree of portability.

Software Engineering Observations



The object-oriented programming paradigm necessitates a complete rethinking of the way we build software systems. C++ is an effective language for achieving good software engineering. The Software Engineering Observations highlight architectural and design issues that affect the construction of software systems, especially large-scale systems. Much of what the student learns here will be useful in upper-level courses and in industry as the student begins to work with large, complex real-world systems.

[Page xxx]

Error-Prevention Tips



When we first designed this "tip type," we thought we would use it strictly to tell people how to test and debug C++ programs. In fact, many of the tips describe aspects of C++ that reduce the likelihood of "bugs" and thus simplify the testing and debugging processes.

Wrap-Up Sections

Each chapter ends with additional pedagogical devices. New in this edition, each chapter ends with a brief "wrap-up" section that recaps the topics that were presented. The wrap-ups also help the student transition to the next chapter.

Summary (1126 Summary bullets)

We present a thorough, bullet-list-style summary at the end of every chapter. On average, there are 40 summary bullets per chapter. This focuses the student's review and reinforces key concepts.

Terminology (1682 Terms)

We include an alphabetized list of the important terms defined in each chapter again, for further reinforcement. There is an average of 82 terms per chapter. Each term also appears in the index, and the defining occurrence of each term is highlighted in the index with a blue, bold page number so the student can locate the definitions of key terms quickly.

609 Self-Review Exercises and Answers (Count Includes Separate Parts)

Extensive self-review exercises and answers are included in each chapter. This gives the student a chance to build confidence with the material and prepare for the regular exercises. We encourage students to do all the self-review exercises and check their answers.

849 Exercises (Solutions in Instructor's Manual; Count Includes Separate Parts)

Each chapter concludes with a substantial set of exercises including simple recall of important terminology and concepts; writing individual C++ statements; writing small portions of C++ functions and classes; writing complete C++ functions, classes and programs; and writing major term projects. The large number of exercises enables instructors to tailor their courses to the unique needs of their audiences and to vary course assignments each semester. Instructors can use these exercises to form homework assignments, short quizzes and major examinations. The solutions for the vast majority of the exercises are included on the Instructor's Resource CD (IRCD), which is available only to instructors through their Prentice Hall representatives. [NOTE: Please do not write to us requesting the Instructor's CD. Distribution of this ancillary is limited strictly to college instructors teaching from the book. Instructors may obtain the solutions manual only from their Prentice Hall representatives.] Students will have access to approximately half the exercises in the book in the free, Web-based Cyber Classroom which will be available in late spring 2005. For more information about the Cyber Classroom, please visit www.deitel.com or sign up for the free Deitel® Buzz Online e-mail newsletter at www.deitel.com/newsletter/subscribe.html.

Approximately 6,000 Index Entries

We have included an extensive index. This helps students find terms or concepts by keyword. The index is

useful to people reading the book for the first time and is especially useful to practicing programmers who use the book as a reference.

[Page xxxi]

"Double Indexing" of All C++ LIVE-CODE Examples

C++ How to Program, 5/e has 260 live-code examples and 849 exercises (including separate parts). We have double indexed each of the live-code examples and most of the more substantial exercises. For every source-code program in the book, we indexed the figure caption both alphabetically and as a subindex item under "Examples." This makes it easier to find examples using particular features. The more substantial exercises are also indexed both alphabetically and as subindex items under "Exercises."

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

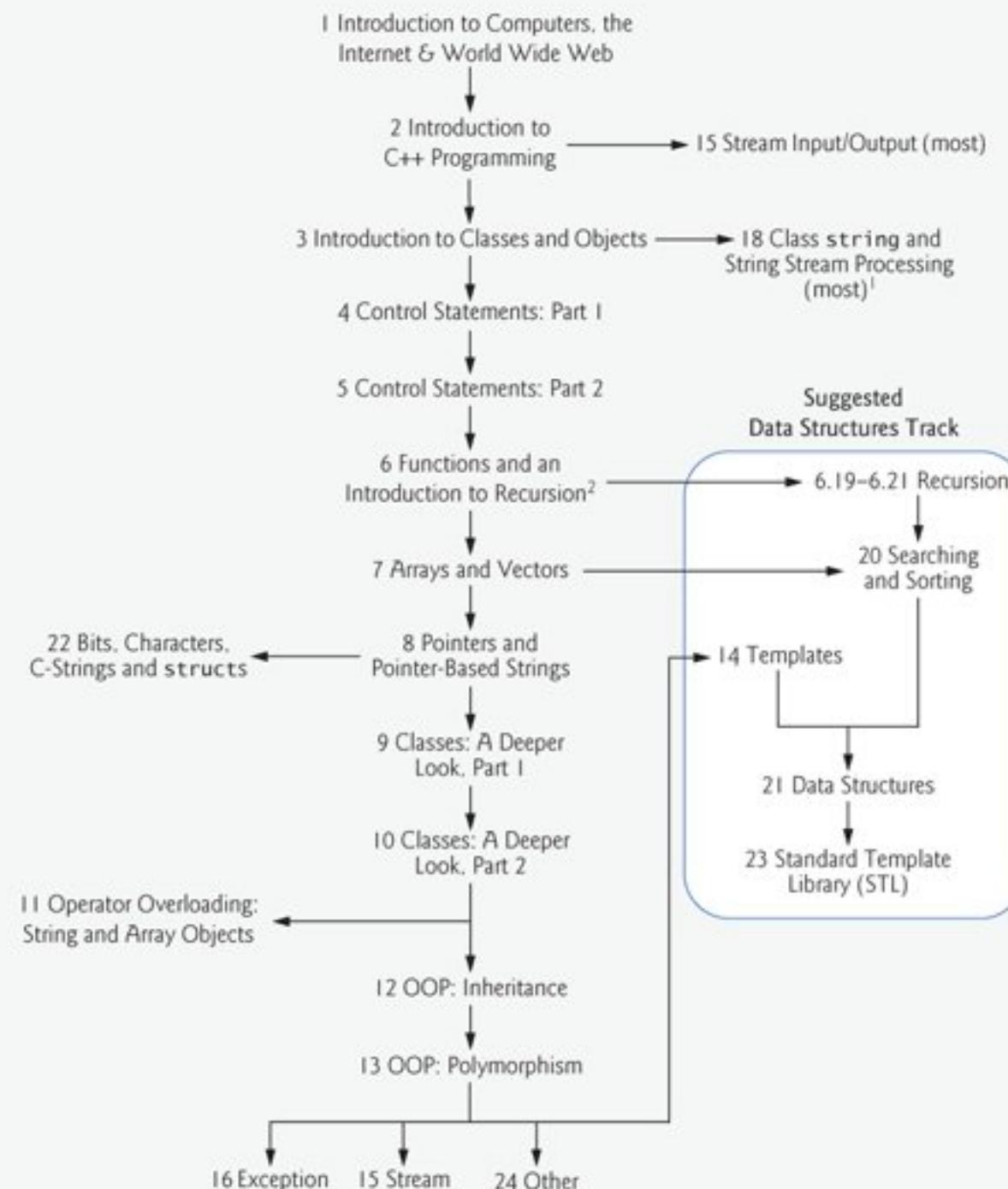
[Page xxxii]

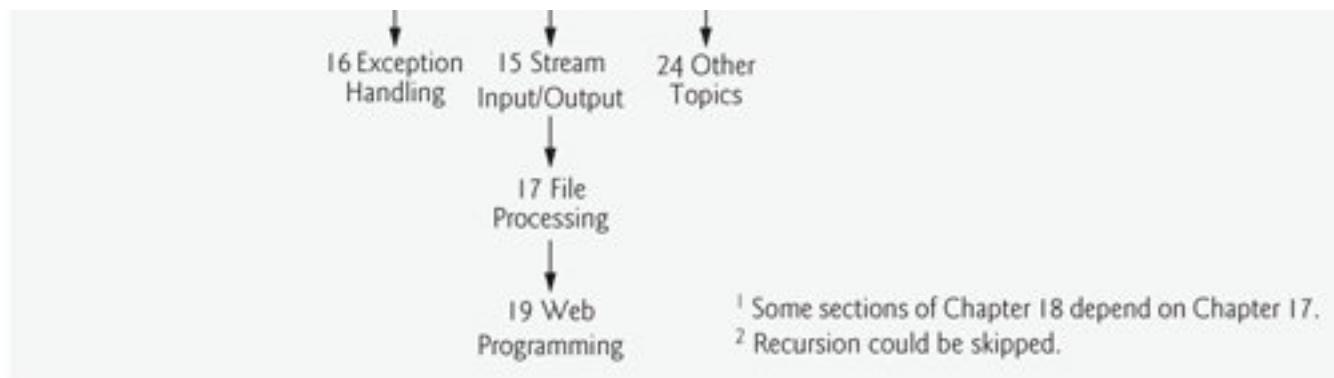
[Page xxxiii]

Figure 1. Flowchart illustrating the dependencies among chapters in C++ How to Program, 5/e.

(This item is displayed on page xxxii in the print version)

[\[View full size image\]](#)





Chapter 4 Control Statements: Part 1 focuses on the program-development process involved in creating useful classes. The chapter discusses how to take a problem statement and develop a working C++ program from it, including performing intermediate steps in pseudocode. The chapter introduces some simple 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. These versions each include a member function that uses control statements to calculate the average of a set of student grades. In the first version, the member function uses counter-controlled repetition to input 10 student grades from the user, then determines the average grade. In the second version, the member function uses sentinel-controlled repetition to input an arbitrary number of grades from the user, then calculates the average of the grades that were entered. 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. This version uses sentinel-controlled repetition to input the grades. While reading the grades from the user, a member function modifies data members that keep track of the count of grades in each letter grade category. Another member function of the class then uses these data members to display a summary report based on the grades entered. The chapter includes a discussion of 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 students can build their own functions. The techniques presented in [Chapter 6](#) are essential to the production of properly organized programs, especially the kinds of larger programs and software that system programmers and application programmers are likely to develop in real-world applications. The "divide and conquer" strategy is presented as an effective means for solving complex problems by dividing them into simpler interacting components. The chapter's first example continues the `GradeBook` class case study with an example of a function with multiple parameters. Students will 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 the reader will use throughout

the book. In this new edition, 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. The chapter then offers a solid introduction to recursion and includes a table summarizing the recursion examples and exercises distributed throughout the remainder of the book. Some texts leave recursion for a chapter late in the book; we feel this topic is best covered gradually throughout the text. The extensive collection of exercises at the end of the chapter includes several classic recursion problems, including the Towers of Hanoi.

[Page xxxiv]

[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 will see in [Chapter 8](#), are really pointers to the array contents in memory. We then present arrays as full-fledged objects in the last section of the chapter, where we introduce 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 and can produce a report containing the average of the grades, the minimum and maximum grades and a bar chart representing the grade distribution. The second version (i.e., the final version in the case study) uses a two-dimensional array to store the grades of a number of students on multiple exams in a semester. This version can calculate each student's semester average, as well as the minimum and maximum grades across all grades received for the semester. The class also produces a bar chart displaying the overall grade distribution for the semester. Another key feature of this chapter is the discussion of elementary sorting and searching techniques. The end-of-chapter exercises include a variety of interesting and challenging problems, such as improved sorting techniques, the design of a simple airline reservations system, an introduction to the concept of turtle graphics (made famous in the LOGO language) and the Knight's Tour and Eight Queens problems that introduce the notion of heuristic programming widely employed in the field of artificial intelligence. The exercises conclude with many recursion problems including selection sort, palindromes, linear search, the Eight Queens, printing an array, printing a string backwards and finding the minimum value in an array.

[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 also introduce and use 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). In this new edition, we use `string` objects (introduced in [Chapter 3](#)) in place of C-style, `char *` pointer-based strings wherever possible. However, we include `char *` strings in [Chapter 8](#) to help the reader master pointers and prepare for the professional world in which the reader will see a great deal of C legacy code that has been implemented over the last three decades. Thus, the reader will become familiar with the two most prevalent methods of creating and manipulating strings in C++. Many people find that the topic of pointers is, by far, the most difficult part of an introductory programming course. In C and "raw C++" arrays and strings are pointers to array and string contents in memory (even function names are pointers). Studying this chapter carefully should reward you with a deep understanding of pointers. The chapter is loaded with challenging exercises. The chapter exercises include a simulation of the classic race between the tortoise and the hare, card-shuffling and dealing algorithms, recursive quicksort and recursive maze traversals. A special section entitled Building Your Own Computer also is included. This section explains machine-language programming and proceeds with a project involving the design and implementation of a computer simulator that leads the student to write and run machine-language programs. This unique feature of the text will be especially useful to the reader who wants to understand how computers really work. Our students enjoy this project and often implement substantial enhancements, many of which are suggested in the exercises. A second special section includes challenging string-manipulation exercises related to text analysis, word processing, printing dates in various formats, check protection, writing the word equivalent of a check amount, Morse Code and metric-to-English conversions.

[Page xxxv]

[Chapter 9](#)Classes: A Deeper Look, Part 1continues 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. Students learn 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. The chapter exercises challenge the student to develop classes for times, dates, rectangles and playing tic-tac-toe. Students generally enjoy game-playing programs. Mathematically inclined readers will enjoy the exercises on creating class `Complex` (for complex numbers), class `Rational` (for rational numbers) and class `HugeInteger` (for arbitrarily large integers).

[Chapter 10](#)Classes: A Deeper Look, Part 2continues the study of classes and presents additional object-oriented programming concepts. The chapter discusses declaring and using constant objects, constant member functions, compositionthe 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 example. 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. The chapter exercises include developing a savings-account class and a class for holding sets of integers.

[Page xxxvi]

[Chapter 11](#) Operator Overloading; String and Array Objects presents one of the most popular topics in our C++ courses. Students really enjoy this material. They find it a perfect match with the detailed discussion of crafting valuable classes in [Chapters 9 and 10](#). Operator overloading enables the programmer to tell the compiler how to use existing operators with objects of new types. C++ already knows how to use these operators with objects of 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](#), the programmer will learn 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 `Strings`. 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 the collection of substantial case studies including an array class, a `String` class, a date class, a huge integer class and a complex numbers class (the last two appear with full source code in the exercises). Mathematically inclined students will enjoy creating the `polynomial` class in the exercises. This material is different from most programming languages and courses. Operator overloading is a complex topic, but an enriching one. Using operator overloading wisely helps you add extra "polish" to your classes. The discussions of class `Array` and class `String` are particularly valuable to students who have already used the C++ Standard Library `string` class and `vector` class template that provide similar capabilities. The exercises encourage the student to add operator overloading to classes `Complex`, `Rational` and `HugeInteger` to enable convenient manipulation of objects of these classes with operator symbols as in mathematics rather than with function calls as the student did in the [Chapter 10](#) exercises.

[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 substantially revised chapter presents a five-example sequence demonstrating private data, protected data and good software engineering with inheritance. We begin by demonstrating a class with private data members and public member functions to manipulate that data. Next, we implement a second class with additional capabilities, intentionally and tediously duplicating much of the first example's code. The third example begins our discussion of inheritance and software reuse we use the class from the first example as a base class and quickly and simply inherit its data and functionality into a new derived class. This example introduces the inheritance mechanism and demonstrates that a derived class cannot access its base class's private members directly. This motivates our fourth example, in which we introduce protected data in the base class and demonstrate that the derived class can indeed access the

protected data inherited from the base class. The last example in the sequence demonstrates proper software engineering by defining the base class's data as private and using the base class's public member functions (that were inherited by the derived class) to manipulate the base class's private data in the derived class. 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.

[Page xxxvii]

[Chapter 13](#) Object-Oriented Programming: Polymorphism deals with another fundamental capability of object-oriented programming: polymorphic behavior. The completely revised [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. Polymorphism enables programs to eliminate complex switch logic in favor of simpler "straight-line" logic. A screen manager of a video game, for example, can send a draw message to every object in a linked list of objects to be drawn. Each object knows how to draw itself. An object of a new class can be added to the program without modifying that program (as long as that new object also knows how to draw itself). 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 demonstrate abstract classes and polymorphic behavior by revisiting the Employee hierarchy of [Chapter 12](#). We introduce an abstract Employee base class, from which classes CommissionEmployee, HourlyEmployee and SalariedEmployee inherit directly and class BasePlusCommissionEmployee inherits indirectly. In the past, our professional clients have insisted that we provide a deeper explanation that shows precisely how polymorphism is implemented in C++, and hence, precisely what execution time and memory "costs" are incurred when programming with this powerful capability. We responded by developing 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. Using RTTI and dynamic casting, we give a 10% pay increase to employees of a specific type, then calculate the earnings for such employees. For all other employee types, we calculate their earnings polymorphically.

[Chapter 14](#) Templates discusses one of C++'s more powerful software reuse features, namely templates. Function templates and class templates enable programmers 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. Function templates were introduced in [Chapter 6](#). This chapter presents additional discussions and examples on function template. 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. The exercises challenge the student to write a variety of function templates and class templates and to employ these in complete programs. We greatly enhance the treatment of templates in our discussion of the Standard Template Library (STL) containers, iterators and algorithms in [Chapter 23](#).

[Page xxxviii]

[Chapter 15](#) C++ 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. This style of I/O makes use of other C++ features, such as references, function overloading and operator overloading. 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 (>>). This extensibility is one of C++'s most valuable features. 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 programmers 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. The exercises in this chapter show the student the diversity and power of C++'s exception-handling capabilities. We discuss rethrowing an exception, and we 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. Students in our industry seminars have mentioned that, after studying the material on file processing, they were able to produce substantial file-processing programs that were immediately useful in their organizations. The exercises ask the student to implement a variety of programs that build and process both sequential files and random-access files.

[Page xxxix]

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 the reader's 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 Web Programming This optional chapter has everything you need to begin developing your own Web-based applications that will really run on the Internet! You will learn how to build so-called n-tier applications, in which the functionality provided by each tier can be distributed to separate computers across the Internet or executed on the same computer. In particular, we build a three-tier online bookstore application. The bookstore's information is stored in the application's bottom tier, also called the data tier. In industrial-strength applications, the data tier is typically a database such as Oracle, Microsoft® SQL Server or MySQL. For simplicity, we use text files and employ the file-processing techniques of [Chapter 17](#) to access and modify these files. The user enters requests and receives responses at the application's top tier, also called the user-interface tier or the client tier, which is typically a computer running a popular Web browser such as Microsoft Internet Explorer, Mac® OS X Safari™, Mozilla Firefox, Opera or Netscape®. Web browsers, of course, know how to communicate with Web sites throughout the Internet. The middle tier, also called the business-logic tier, contains both a Web server and an application-specific C++ program (e.g., our bookstore application). The Web server communicates with the C++ program (and vice versa) via the CGI (Common Gateway Interface) protocol. This program is referred to as a CGI script. We use the popular Apache HTTP server, which is available free for download from the Apache Web site, www.apache.org. Apache installation instructions for many popular platforms, including Linux and Windows systems, are available at that site and at www.deitel.com and www.prenhall.com/deitel. The Web server knows how to talk to the client tier across the Internet using a protocol called HTTP (Hypertext Transfer Protocol). We discuss the two most popular HTTP methods for

sending data to a Web server GET and POST. We then discuss the crucial role of the Web server in Web programming and provide a simple example that requests an Extensible HyperText Markup Language (XHTML)^[1] document from a Web server. We discuss CGI and how it allows a Web server to communicate with the top tier and CGI applications. We provide a simple example that gets the server's time and renders it in a browser. Other examples demonstrate how to process form-based user input via the string processing techniques introduced in [Chapter 18](#). In our forms-based examples we use buttons, password fields, check boxes and text fields. We present an example of an interactive portal for a travel company that displays airfares to various cities. Travel-club members can log in and view discounted airfares. We also discuss various methods of storing client-specific data, which include hidden fields (i.e., information stored in a Web page but not rendered by the Web browser) and cookies small text files that the browser stores on the client's machine. The chapter examples conclude with a case study of an online book store that allows users to add books to a shopping cart. This case study contains several CGI scripts that interact to form a complete application. The online book store is password protected, so users first must log in to gain access. The chapter's Web resources include information about the CGI specification, C++ CGI libraries and Web sites related to the Apache HTTP server.

^[1] XHTML is a markup language for identifying the elements of an XHTML document (Web page) so that a browser can render (i.e., display) that page on your computer screen. XHTML is a new technology designed by the World Wide Web Consortium to replace the HyperText Markup Language (HTML) as the primary means of specifying Web content. In Appendices J and K, we introduce XHTML.

[Page xl]

[Chapter 20](#) Searching and Sorting discusses two of the most important classes of algorithms in computer science. We consider a variety of specific algorithms for each and compare them with regard to their memory consumption and processor consumption (introducing Big O notation, which indicates how hard an algorithm may have to work to solve a problem). Searching data involves determining whether a value (referred to as the search key) is present in the data and, if so, finding the value's location. In the examples and exercises of this chapter, we discuss a variety of searching algorithms, including: binary search and recursive versions of linear search and binary search. Through examples and exercises, [Chapter 20](#) discusses the recursive merge sort, bubble sort, bucket sort and the recursive quicksort.

[Chapter 21](#) Data Structures discusses the techniques used to create and manipulate dynamic data structures. The chapter begins with discussions of self-referential classes and dynamic memory allocation, then proceeds with a discussion of how to create and maintain various dynamic data structures, including linked lists, queues (or waiting lines), stacks and trees. For each type of data structure, we present complete, working programs and show sample outputs. The chapter also helps the student master pointers. The chapter includes abundant examples that use indirection and double indirection a particularly difficult concept. One problem when working with pointers is that students have trouble visualizing the data structures and how their nodes are linked together. We have included illustrations that show the links and the sequence in which they are created. The binary-tree example is a superb capstone for the study of pointers and dynamic data structures. This example creates a binary tree, enforces duplicate elimination and introduces recursive preorder, inorder and postorder tree traversals. Students have a genuine sense of accomplishment when they study and implement this example. They particularly

appreciate seeing that the inorder traversal prints the node values in sorted order. We include a substantial collection of exercises. A highlight of the exercises is the special section Building Your Own Compiler. The exercises walk the student through the development of an infix-to-postfix-conversion program and a postfix-expression-evaluation program. We then modify the postfix-evaluation algorithm to generate machine-language code. The compiler places this code in a file (using the techniques of [Chapter 17](#)). Students then run the machine language produced by their compilers on the software simulators they built in the exercises of [Chapter 8](#)! The 35 exercises include recursively searching a list, recursively printing a list backwards, binary-tree node deletion, level-order traversal of a binary tree, printing trees, writing a portion of an optimizing compiler, writing an interpreter, inserting/deleting anywhere in a linked list, implementing lists and queues without tail pointers, analyzing the performance of binary-tree searching and sorting, implementing an indexed-list class and a supermarket simulation that uses queueing. After studying [Chapter 21](#), the reader is prepared for the treatment of STL containers, iterators and algorithms in [Chapter 23](#). The STL containers are prepackaged, templated data structures that most programmers will find sufficient for the vast majority of applications they will need to implement. The STL is a giant leap forward in achieving the vision of reuse.

[Page xli]

[Chapter 22](#) Bits, Characters, Strings and Structures represents a variety of important features. This chapter 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. The chapter features a high-performance card-shuffling and dealing simulation. This is an excellent opportunity for the instructor to emphasize the quality of algorithms. C++'s powerful bit-manipulation capabilities enable programmers 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. As you recall, we introduced C-style `char *` string manipulation in [Chapter 8](#) and presented the most popular string-manipulation functions. In [Chapter 22](#), 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; as always, every function is presented in the context of a complete, working C++ program. The 32 exercises encourage the student to try out most of the capabilities discussed in the chapter. The feature exercise leads the student through the development of a spelling-checker program. This chapter presents a deeper treatment of C-like, `char *` strings for the benefit of C++ programmers who are likely to work with C legacy code.

[Chapter 23](#) Standard Template Library (STL) Throughout this book, we discuss the importance of software reuse. Recognizing that many data structures and algorithms are commonly used by C++ programmers, the C++ standard committee added the Standard Template Library (STL) to the C++ Standard Library. The STL defines 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](#) and demonstrated in detail in [Chapter 21](#). This chapter introduces the STL and discusses its three key components containers (popular templated

data structures), iterators and algorithms. The STL containers are data structures capable of storing objects of any data type. We will see that there are three container categoriesfirst-class containers, adapters and near containers. STL iterators, which have similar properties to those of pointers, are used by programs to manipulate the STL-container elements. In fact, standard arrays can be manipulated as STL containers, using standard pointers as iterators. We will see that manipulating containers with iterators is convenient and provides tremendous expressive power when combined with STL algorithmsin 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). There are approximately 70 algorithms implemented in the STL. Most of these use iterators to access container elements. We will see that each first-class container supports specific iterator types, some of which are more powerful than others. A container's supported iterator type determines whether the container can be used with a specific algorithm. Iterators encapsulate the mechanism used to access container elements. This encapsulation enables many of the STL algorithms to be applied to several containers without regard for the underlying container implementation. As long as a container's iterators support the minimum requirements of the algorithm, then the algorithm can process that container's elements. This also enables programmers to create algorithms that can process the elements of multiple different container types. [Chapter 21](#) discusses how to implement data structures with pointers, classes and dynamic memory. Pointer-based code is complex, and the slightest omission or oversight can lead to serious memory-access violations and memory-leak errors with no compiler complaints. Implementing additional data structures such as deques, priority queues, sets, maps, etc. requires substantial additional work. In addition, if many programmers on a large project implement similar containers and algorithms for different tasks, the code becomes difficult to modify, maintain and debug. An advantage of the STL is that programmers can reuse the STL containers, iterators and algorithms to implement common data representations and manipulations. This reuse results in substantial development-time and resource savings. This is a friendly, accessible chapter that should convince you of the value of the STL and encourage further study.

[Page xlii]

[Chapter 24](#)**Other Topics**is a collection of miscellaneous C++ topics. This chapter discusses one additional cast operator**const_cast**. This operator, along with **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, especially for those who build systems from class libraries. Namespaces prevent naming collisions, which can hinder such large software efforts. The Chapter discusses the operator keywords, which are useful for programmers who have keyboards that do not support certain characters used in operator symbols, such as !, &, ^, ~ and |. These operators can also be used by programmers who do not like cryptic operator symbols. 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.

[Page xliii]

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 all fundamental types defined in the C++ Standard.

Appendix D Number Systems discusses the binary, octal, decimal and hexadecimal number systems. It considers how to convert numbers between bases and explains the one's complement and two's complement binary representations.

Appendix E Legacy-Code Topics presents additional topics including several advanced topics not ordinarily covered in introductory courses. We show how to redirect program input to come from a file, redirect program output to be placed in a file, redirect the output of one program to be the input of another program (piping) and append the output of a program to an existing file. We develop functions that use variable-length argument lists and show how to pass command-line arguments to function `main` and use them in a program. We discuss how to compile programs whose components are spread across multiple files, register functions with `atexit` to be executed at program termination and terminate program execution with function `exit`. We also discuss the `const` and `volatile` type qualifiers, specifying the type of a numeric constant using the integer and floating-point suffixes, using the signal-handling library to trap unexpected events, creating and using dynamic arrays with `calloc` and `realloc`, using `unions` as a space-saving technique and using linkage specifications when C++ programs are to be linked with legacy C code. As the title suggests, this chapter is intended primarily for C++ programmers who will be working with C legacy code as most C++ programmers are almost certain to do at one point in their careers.

Appendix F Preprocessor provides detailed discussions of the preprocessor directives. The chapter 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 chapter explains conditional compilation for enabling the programmer 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 G ATM Case Study Code contains the implementation of our case study on object-oriented design with the UML. This appendix is discussed in the overview of the case study (presented shortly).

Appendix H UML 2 Diagrams **Overs**views the UML 2 diagram types that are not found in the OOD/UML Case Study.

Appendix I C++ Internet and Web Resources contains a listing of valuable C++ resources, such as demos, information about popular compilers (including "freebies"), books, articles, conferences, job banks, journals, magazines, help, tutorials, FAQs (frequently asked questions), newsgroups, Web-based courses, product news and C++ development tools.

Appendix J Introduction to XHTML provides an introduction to XHTML a markup language for describing the elements of a Web page so that a browser, such as Microsoft Internet Explorer or Netscape, can render that page. The reader should be familiar with the contents of this appendix before studying **Chapter 16**, Web Programming with CGI. This appendix does not contain any C++ programming. Some key topics covered include incorporating text and images into an XHTML document, linking to other XHTML documents, incorporating special characters (such as copyright and trademark symbols) into an XHTML document, separating parts of an XHTML document with horizontal lines (called horizontal rules), presenting information in lists and tables, and collecting information from users browsing a site.

Appendix K XHTML Special Characters lists many commonly used XHTML special characters, called character entity references.

Appendix L Using the Visual C++ .NET Debugger demonstrates key features of the Visual Studio .NET 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 students learn how to use the debugger in a hands-on manner.

Appendix M Using the GNU C++ Debugger demonstrates key features of the GNU C++ 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 students learn how to use the debugger in a hands-on manner.

Bibliography lists over 100 books and articles to encourage the student to do further reading on C++ and OOP.

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

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page xlv]

Section 1.17 Software Engineering Case Study: Introduction to Object Technology and the UMLintroduces 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 2.8(Optional) Software Engineering Case Study: Examining the ATM Requirements

Documentdiscusses a requirements document that specifies the requirements for a system that we will design and implementthe 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 include a list of URLs and book references on object-oriented design with the UML. We discuss the interaction between the ATM system specified by the requirements document and its user. Specifically, we investigate the scenarios that may occur between the user and the system itselfthese 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 Documentsbegins to design the ATM system. We identify its classes, or "building blocks," by extracting the nouns and noun phrases from the requirements document. 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 4.13(Optional) Software Engineering Case Study: Identifying Class Attributes in the ATM Systemfocuses on the attributes of the classes discussed in [Section 3.11](#). A class contains both attributes (data) and operations (behaviors). As we will 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 document, then place the attributes in the class diagram we created in [Section 3.11](#).

Section 5.11(Optional) Software Engineering Case Study: Identifying Objects' States and Activities in the ATM Systemdiscusses 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 activitythe work it performs in its lifetime. The UML provides the activity diagrama 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 Systemidentifies the operations, or services, of our classes. We extract from the requirements document

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 document. However, as future chapters introduce such topics as inheritance, we will modify our classes and diagrams.

[Page xlvi]

Section 7.12(Optional) Software Engineering Case Study: Collaboration Among Objects in the ATM Systemprovides 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 collaborationsmessages 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 diagramthe 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 9.12(Optional) Software Engineering Case Study: Starting to Program the Classes of the ATM Systemtakes a break from designing the behavior of our system. 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 4.13](#) and [Section 6.22](#), we show how to implement a class in C++ from a design. We do not implement all classesbecause 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 Systemcontinues our discussion of object-oriented programming. We consider inheritanceclasses 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 relationshipsthe 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 G](#).

Appendix GATM Case Study CodeThe 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, students will have completed the design and implementation of a real-world system, and should feel confident tackling larger systems, such as those that professional software engineers build.

Appendix HUML 2 Additional DiagramsOverviews the UML 2 diagram types that are not found in the OOD/ UML Case Study.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page xlvii]

Free C++ Compilers and Trial-Edition C++ Compilers on the Web

Many C++ compilers are available for download from the Web. We discuss several that are available for free or as free-trial versions. Please keep in mind that in many cases, the trial-edition software cannot be used after the (often brief) trial period has expired.

One popular organization that develops free software is the GNU Project (www.gnu.org), originally created to develop a free operating system similar to UNIX. GNU offers developer resources, including editors, debuggers and compilers. Many developers use the GCC (GNU Compiler Collection) compilers, available for download from gcc.gnu.org. The GCC contains compilers for C, C++, Java and other languages. The GCC compiler is a command-line compiler (i.e., it does not provide a graphical user interface). Many Linux and UNIX systems come with the GCC compiler installed. Red Hat has developed Cygwin (www.cygwin.com), an emulator that allows developers to use UNIX commands on Windows. Cygwin includes the GCC compiler.

Borland provides a Windows-based C++ developer product called C++Builder (www.borland.com/cbuilder/cppcomp/index.html). The basic C++Builder compiler (a command-line compiler) is free for download. Borland also provides several versions of C++Builder that contain graphical user interfaces (GUIs). These GUIs are formally called integrated development environments (IDEs) and enable the developer to edit, debug and test programs quickly and conveniently. Using an IDE, many of the tasks that involved tedious commands can now be executed via menus and buttons. Some of these products are available on a free-trial basis. For more information on C++Builder, visit

www.borland.com/products/downloads/download_cbuilder.html

For Linux developers, Borland provides the Borland Kylix development environment. The Borland Kylix Open Edition, which includes an IDE, can be downloaded from

www.borland.com/products/downloads/download_kylix.html

Borland also provides C++BuilderXa cross-platform integrated C++ development environment. The free Personal Edition is available from

www.borland.com/products/downloads/download_cbuilderx.html

The command-line compiler (version 5.6.4) that comes with C++BuilderX was one of several compilers we used to test the programs in this book. Many of the downloads available from Borland require users to register.

The Digital Mars C++ Compiler (www.digitalmars.com), is available for Windows and DOS, and includes tutorials and documentation. Readers can download a command-line or IDE version of the compiler. The DJGPP C/C++ development system is available for computers running DOS. DJGPP stands for DJ's GNU Programming Platform, where DJ is for DJ Delorie, the creator of DJGPP. Information on DJGPP can be found at www.delorie.com/djgpp. Locations where the compiler can be downloaded are provided at www.delorie.com/djgpp/getting.html.

[Page xlviii]

For a list of other compilers that are available free for download, visit the following sites:

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 compilersthis is normal.

Some older C++ compilers, such as Microsoft Visual C++ 6, Borland C++ 5.5 and various earlier versions of GNU C++ 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. The Web site for this book (www.deitel.com/books/cpphtp5/index.html) lists the warnings and error messages that are produced by several older compilers and what, if anything, you can do to fix the warnings and errors.

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. We will post a list of these issues at www.deitel.com/books/cpphtp5/index.html.

DIVE-INTO™ Series Tutorials for Popular C++ Environments

Our free Dive-Into™ Series publications, which are available with the resources for C++ How to Program, 5/e at www.deitel.com/books/downloads.html, help students and instructors familiarize themselves with various C++ development tools. These publications include:

- Dive-Into Microsoft® Visual C++® 6
- Dive-Into Microsoft® Visual C++® .NET
- Dive-Into Borland™ C++Builder™ Compiler (command-line version)
- Dive-Into Borland™ C++Builder™ Personal (IDE version)
- Dive-Into GNU C++ on Linux and Dive-Into GNU C++ via Cygwin on Windows (Cygwin is a UNIX emulator for Windows. It includes the GNU C++ compiler)

Each of these tutorials shows how to compile, execute and debug C++ applications in that particular compiler product. Many of these documents also provide step-by-step instructions with screenshots to help readers install the software. Each document overviews the compiler and its online documentation.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page xl ix]

Teaching Resources for C++ How to Program, 5/e

C++ How to Program, 5/e, has extensive resources for instructors. The Instructor's Resource CD (IRCD) contains the Solutions Manual with solutions to the vast majority of the end-of-chapter exercises, a Test Item File of multiple-choice questions (approximately two per book section) and PowerPoint slides containing all the code and figures in the text, plus bulleted items that summarize the key points in the text. Instructors can customize the slides. [Note: The IRCD is available only to instructors through their Prentice Hall representatives. To find your local sales representative, visit

vig.prenhall.com/replocator

If you need additional help or if you have any questions about the IRCD, please e-mail us at deitel@deitel.com. We will respond promptly.]

[◀ PREV](#)[NEXT ▶](#)[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page xlix (continued)]

C++ Multimedia Cyber Classroom, 5/e, Online

C++ How to Program, 5/e and Small C++ How to Program, 5/e each include a free, Web-based interactive multimedia ancillary to the book *The C++ Multimedia Cyber Classroom, 5/e* available with new books purchased from Prentice Hall for fall 2005 classes. Our Web-based Cyber Classroom will include audio walkthroughs of code examples in the text, solutions to about half of the exercises in the book, a free lab manual and more. For more information about the new Web-based Cyber Classroom, please visit our Web site at www.deitel.com or sign up for the free Deitel® Buzz Online e-mail newsletter at

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

Students who use our Cyber Classrooms tell us that they like the interactivity and that the Cyber Classroom is a powerful reference tool. Professors tell us that their students enjoy using the Cyber Classroom and consequently spend more time on the courses, mastering more of the material than in textbook-only courses. For a complete list of our current CD-ROM-based Cyber Classrooms, see the Deitel® Series page at the beginning of this book, the product listing and ordering information at the end of this book, or visit www.deitel.com, www.prenhall.com/deitel or www.InformIT.com/deitel.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page I]

Solutions to the lab manual's Prelab Activities, Lab Exercises and Postlab Activities are available in electronic form. Instructors can obtain these materials from their regular Prentice Hall representatives; the solutions are not available to students.

Prelab Activities

Prelab Activities are intended to be completed by students after studying each chapter of Small C++ How to Program, 5/e. Prelab Activities test students' understanding of the textbook material and prepare students for the programming exercises in the lab session. The exercises focus on important terminology and programming concepts and are effective for self-review. Prelab Activities include Matching Exercises, Fill-in-the-Blank Exercises, Short-Answer Questions, Programming-Output Exercises (determine what short code segments do without actually running the program) and Correct-the-Code Exercises (identify and correct all errors in short code segments).

Lab Exercises

The most important section in each chapter is the Lab Exercises. These teach students how to apply the material learned in C++ How to Program, 5/e, and prepare them for writing C++ programs. Each lab contains one or more lab exercises and a debugging problem. The Lab Exercises contain the following:

- Lab Objectives highlight specific concepts on which the lab exercise focuses.
- Problem Descriptions provide the details of the exercise and hints to help students implement the program.
- Sample Outputs illustrate the desired program behavior, which further clarifies the problem descriptions and aids the students with writing programs.
- Program Templates take complete C++ programs and replace key lines of code with comments describing the missing code.
- Problem-Solving Tips highlight key issues that students need to consider when solving the lab exercises.
- Follow-Up Questions and Activities ask students to modify solutions to the lab exercises, write new programs that are similar to their lab-exercise solutions or explain the implementation choices that were made when solving lab exercises.
- Debugging Problems consist of blocks of code that contain syntax errors and/or logic errors. These alert students to the types of errors they are likely to encounter while programming.

Postlab Activities

Professors typically assign Postlab Activities to reinforce key concepts or to provide students with more programming experience outside the lab. Postlab Activities test the students' understanding of the Prelab

and Lab Exercise material, and ask students to apply their knowledge to creating programs from scratch. The section provides two types of programming activities: coding exercises and programming challenges. Coding exercises are short and serve as review after the Prelab Activities and Lab Exercises have been completed. The coding exercises ask students to write programs or program segments using key concepts from the textbook. Programming Challenges allow students to apply their knowledge to substantial programming exercises. Hints, sample outputs and pseudocode are provided to aid students with these problems. Students who successfully complete the Programming Challenges for a chapter have mastered the chapter material. Answers to the programming challenges are available at www.deitel.com/books/downloads.html.

[Page li]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page li (continued)]

CourseCompassSM, WebCTTM and BlackboardTM

Selected content from the Deitels' introductory programming language How to Program series textbooks, including C++ How to Program, 5/e, is available to integrate into various popular course management systems, including CourseCompass, Blackboard and WebCT. Course management systems help faculty create, manage and use sophisticated Web-based educational tools and programs. Instructors can save hours of inputting data by using the Deitel course-management-systems content. Blackboard, CourseCompass and WebCT offer:

- Features to create and customize an online course, such as areas to post course information (e.g., policies, syllabi, announcements, assignments, grades, performance evaluations and progress tracking), class and student management tools, a gradebook, reporting tools, page tracking, a calendar and assignments.
- Communication tools to help create and maintain interpersonal relationships between students and instructors, including chat rooms, whiteboards, document sharing, bulletin boards and private e-mail.
- Flexible testing tools that allow an instructor to create online quizzes and tests from questions directly linked to the text, and that grade and track results effectively. All tests can be inputted into the gradebook for efficient course management. WebCT also allows instructors to administer timed online quizzes.
- Support materials for instructors are available in print and online formats.

In addition to the types of tools found in Blackboard and WebCT, CourseCompass from Prentice Hall includes:

- CourseCompass course home page, which makes the course as easy to navigate as a book. An expandable table of contents allows instructors to view course content at a glance and to link to any section.
- Hosting on Prentice Hall's centralized servers, which allows course administrators to avoid separate licensing fees or server-space issues. Access to Prentice Hall technical support is available.
- "How Do I" online-support sections are available for users who need help personalizing course sites, including step-by-step instructions for adding PowerPoint slides, video and more.
- Instructor Quick Start Guide helps instructors create online courses using a simple, step-by-step process.

To view free online demonstrations and learn more about these Course Management Systems, which support Deitel content, visit the following Web sites:

- Blackboard: www.blackboard.com and www.prenhall.com/blackboard
- WebCT: www.webct.com and www.prenhall.com/webct
- CourseCompass: www.coursecompass.com and www.prenhall.com/coursecompass

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page liii]

Appendices in Only C++ How to Program, 5/e

C Legacy-Code Topics

Preprocessor

ATM Case Study Code

UML 2 Diagrams

Introduction to XHTML

XHTML Special Characters

SafariX WebBooks

SafariX Textbooks Online is a new service for college students looking to save money on required or recommended textbooks for academic courses. This secure WebBooks platform creates a new option in the higher education market; an additional choice for students alongside conventional textbooks and online learning services. Pearson provides students with a WebBook at 50% of the cost of its conventional print equivalent.

SafariX WebBooks are viewed through a Web browser connected to the Internet. No special plug-ins are required and no applications need to be downloaded. Students simply log in, purchase access and begin studying. With SafariX Textbooks Online students can search the text, make notes online, print out reading assignments that incorporate their professors' lecture notes and bookmark important passages they want to review later. They can navigate easily to a page number, reading assignment, or chapter. The Table of Contents of each WebBook appears in the left-hand column alongside the text.

We are pleased to offer students the C++ How to Program, 5/e SafariX WebBook available for fall 2005 classes. Visit www.pearsonchoices.com for more information. Other Deitel titles available as SafariX WebBooks include Java How to Program, 6/e, Small Java How to Program, 6/e, Small C++ How to Program, 5/e and Simply C++: An Application-Driven Tutorial Approach. Visit www.safarix.com/tour.html for more information.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page liii (continued)]

The Deitel® Buzz Online Free E-mail Newsletter

Our free e-mail newsletter, the DEITEL® Buzz Online is sent to approximately 38,000 opt-in, registered subscribers 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 our way to notify our readers rapidly about issues related to C++ How to Program, 5/e. To subscribe, visit

www.deitel.com/newsletter/subscribe.html

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page liv]

- Jeff Listfield is a Computer Science graduate of Harvard College. Jeff contributed to [Chapters 18, 20](#) and [22, Appendices AF](#) and co-authored [Appendices L](#) and [M](#).
- Su Zhang holds B.Sc. and a M.Sc. degrees in Computer Science from McGill University. Su contributed to [Chapters 1424](#).
- Cheryl Yaeger graduated from Boston University in three years with a bachelor's degree in Computer Science. Cheryl contributed to [Chapters 4, 6, 8, 9](#) and [13](#).
- Barbara Deitel, Chief Financial Officer at Deitel & Associates, Inc. researched the quotes at the beginning of each chapter and applied copyedits to the book.
- Abbey Deitel, President of Deitel & Associates, Inc., is an Industrial Management graduate of Carnegie Mellon University. She contributed to the [Preface](#) and [Chapter 1](#). She applied copyedits to several chapters in the book, managed the review process and suggested the theme and bug names for the cover of the book.
- Christi Kelsey is a graduate of Purdue University with a bachelor's degree in Management and a minor in Information Systems. Christi contributed to the [Preface](#) and [Chapter 1](#). She edited the Index, paged the manuscript and coordinated many aspects of our publishing relationship with Prentice Hall.

We are fortunate to have worked on this project with the talented and dedicated team of publishing professionals at Prentice Hall. We especially appreciate the extraordinary efforts of our Computer Science Editor, Kate Hargett and her boss and our mentor in publishing Marcia Horton, Editorial Director of Prentice Hall's Engineering and Computer Science Division. Jennifer Cappello did an extraordinary job recruiting the review team and managing the review process from the Prentice Hall side. Vince O'Brien, Tom Mansreck and John Lovell did a marvelous job managing the production of the book. The talents of Paul Belfanti, Carole Anson, Xiaohong Zhu and Geoffrey Cassar are evident in the redesign of the book's interior and the new cover art, and Sarah Parker managed the publication of the book's extensive ancillary package.

We sincerely appreciate the efforts of our fourth-edition post-publication reviewers and our fifth-edition reviewers:

Academic Reviewers

Richard Albright, Goldey Beacom College

Karen Arlien, Bismarck State College

David Branigan, DeVry University, Illinois

Jimmy Chen, Salt Lake Community College

Martin Dulberg, North Carolina State University

Ric Heishman, Northern Virginia Community College

Richard Holladay, San Diego Mesa College

William Honig, Loyola University

Earl LaBatt, OPNET Technologies, Inc./ University of New Hampshire

Brian Larson, Modesto Junior College

Robert Myers, Florida State University

Gavin Osborne, Saskatchewan Institute of Applied Science and Technology

[Page Iv]

Wolfgang Pelz, The University of Akron

Donna Reese, Mississippi State University

Industry Reviewers

Curtis Green, Boeing Integrated Defense Systems

Mahesh Hariharan, Microsoft

James Huddleston, Independent Consultant

Ed James-Beckham, Borland Software Corporation

Don Kostuch, Independent Consultant

Meng Lee, Hewlett-Packard

Kriang Lerdsuwanakij, Siemens Limited

William Mike Miller, Edison Design Group, Inc.

Mark Schimmel, Borland International

Vicki Scott, Metrowerks

James Snell, Boeing Integrated Defense Systems

Raymond Stephenson, Microsoft

OOD/UML Optional Software Engineering Case Study Reviewers

Sinan Si Alhir, Independent Consultant

Karen Arlien, Bismarck State College

David Branigan, DeVry University, Illinois

Martin Dulberg, North Carolina State University

Ric Heishman, Northern Virginia Community College

Richard Holladay, San Diego Mesa College

Earl LaBatt, OPNET Technologies, Inc./ University of New Hampshire

Brian Larson, Modesto Junior College

Gavin Osborne, Saskatchewan Institute of Applied Science and Technology

Praveen Sadhu, Infodat International, Inc.

Cameron Skinner, Embarcadero Technologies, Inc. / OMG

Steve Tockey, Construx Software

C++ 4/e Post-Publication Reviewers

Butch Anton, Wi-Tech Consulting

Karen Arlien, Bismarck State College

Jimmy Chen, Salt Lake Community College

Martin Dulberg, North Carolina State University

William Honig, Loyola University

Don Kostuch, Independent Consultant

Earl LaBatt, OPNET Technologies, Inc./ University of New Hampshire

Brian Larson, Modesto Junior College

Kriang Lerdswananakij, Siemens Limited

Robert Myers, Florida State University

Gavin Osborne, Saskatchewan Institute of Applied Science and Technology

Wolfgang Pelz, The University of Akron

David Papurt, Independent Consultant

Donna Reese, Mississippi State University

[Page lvi]

Catherine Wyman, DeVry University, Phoenix

Salih Yurttas, Texas A&M University

Under tight deadline pressure, they 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. Good luck! 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 will respond promptly, and we will post corrections and clarifications on:

www.deitel.com/books/cpphtp5/index.html

We hope you enjoy learning with C++ How to Program, Fifth Edition as much as we enjoyed writing it!

Dr. Harvey M. Deitel

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page Ivii]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page Ivii (continued)]

About Deitel & Associates, Inc.

Deitel & Associates, Inc., is an internationally recognized corporate training and content-creation organization specializing in computer programming languages, Internet/World Wide Web software technology and object technology education. The company provides instructor-led courses on major programming languages and platforms such as Java, Advanced Java, C, C++, .NET programming languages, XML, Perl, Python; object technology; and Internet and World Wide Web programming. The founders of Deitel & Associates, Inc., are Dr. Harvey M. Deitel and Paul J. Deitel. The company's clients include many of the world's largest computer companies, government agencies, branches of the military and business organizations. Through its 29-year publishing partnership with Prentice Hall, Deitel & Associates, Inc. publishes leading-edge programming textbooks, professional books, interactive multimedia Cyber Classrooms, Complete Training Courses, Web-based training courses and course management systems e-content for popular CMSs such as WebCT, Blackboard and Pearson's CourseCompass. 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 worldwide DIVE INTO™ Series Corporate Training curriculum, see the last few pages of this book or visit:

www.deitel.com

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

www.deitel.com/newsletter/subscribe.html

Individuals wishing to purchase Deitel books, Cyber Classrooms, Complete Training Courses and Web-based training courses can do so through:

www.deitel.com/books/index.html

Bulk orders by corporations and academic institutions should be placed directly with Prentice Hall. See the last few pages of this book for worldwide ordering details.

[◀ PREV](#)[NEXT ▶](#)

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page lix]

Before You Begin

Please follow the instructions in this section to ensure that the book's examples are copied properly to your computer before you begin using this book.

Font and Naming Conventions

We use fonts to distinguish between on-screen components (such as menu names and menu items) and C++ code or commands. Our convention is to emphasize on-screen components in a sans-serif bold Helvetica font (for example, File menu) and to emphasize C++ code and commands in a sans-serif Lucida font (for example, cout << "Hello" ;).

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page lix (continued)]

Resources on the CD That Accompanies C++ How to Program, Fifth Edition

The CD that accompanies this book includes:

- Hundreds of C++ LIVE-CODE examples
- Links to free C++ compilers and integrated development environments (IDEs)
- Hundreds of Web resources, including general references, tutorials, FAQs, newsgroups and STL information.

If you have any questions, please feel free to email us at deitel@deitel.com. We will respond promptly.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page lix (continued)]

Copying and Organizing Files

All of the examples for C++ How To Program, Fifth Edition are included on the CD that accompanies this book. Follow the steps in the next section, Copying the Book Examples from the CD, to copy the examples directory from the CD onto your hard drive. We suggest that you work from your hard drive rather than your CD drive for two reasons: The CD is read-only, so you cannot save your applications to the book's CD, and files can be accessed faster from a hard drive than from a CD. The examples from the book are also available for download from:

www.deitel.com/books/cpphtp5/index.html

www.prenhall.com/deitel

We assume for the purpose of this Before You Begin section that you are using a computer running Microsoft Windows. Screen shots in the following section might differ slightly from what you see on your computer, depending on whether you are using Windows 2000 or Windows XP. If you are running a different operating system and have questions about copying the example files to your computer, please see your instructor.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page Ix]

Copying the Book Examples from the CD

1. Inserting the CD. Insert the CD that accompanies C++ How To Program, Fifth Edition into your computer's CD drive. The window displayed in [Fig. 1](#) should appear. If the page appears, proceed to Step 3 of this section. If the page does not appear, proceed to Step 2.

Figure 1. Welcome page for C++ How to Program CD.

(This item is displayed on page Ixi in the print version)

[\[View full size image\]](#)

Welcome to C++ How to Program, 5th Edition - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Home Search Favorites Favorites Go

D:\welcome.htm

Student CD to Accompany

C++
How to Program,
5th Edition

Deitel and Deitel
CD ISBN 0-13-185951-X
© 2005 Pearson Education, Inc.

Software Downloads Examples Web Resources

Browse CD Contents

My Computer

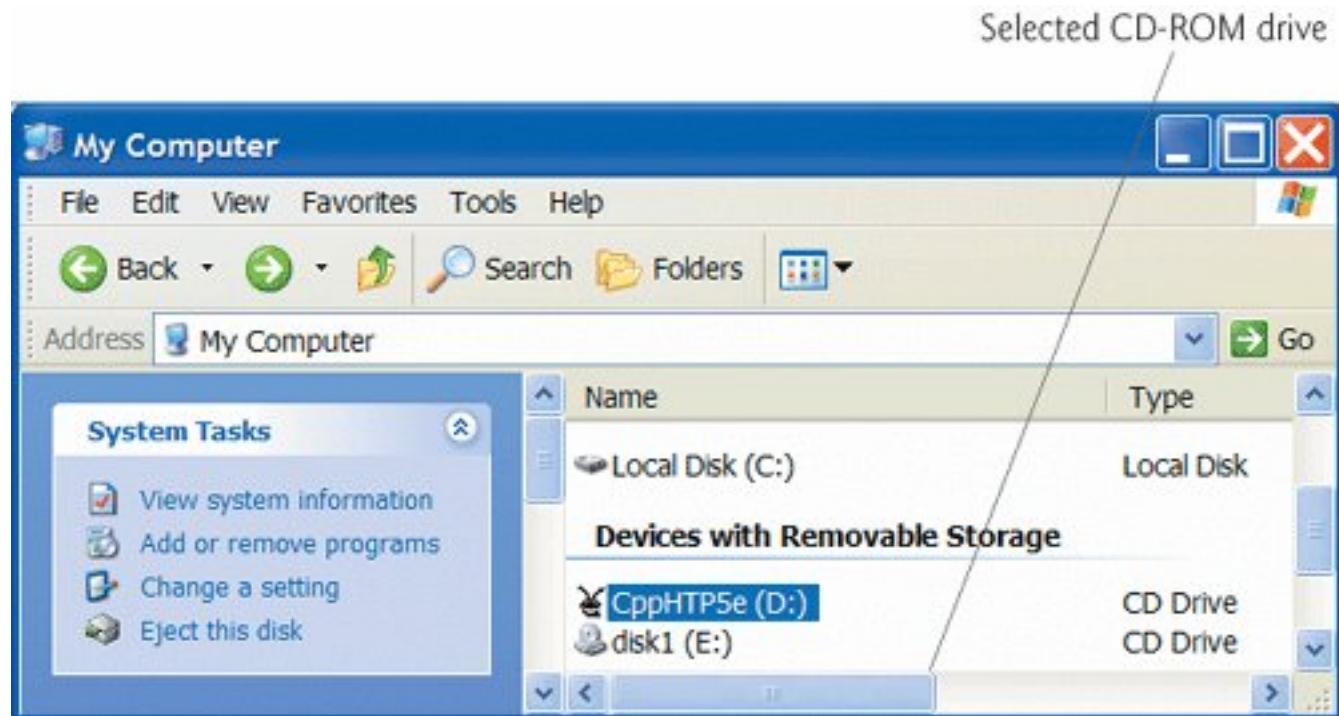
Click the **Browse CD Contents** link to access the CD's contents

- 2.** Opening the CD directory using My Computer. If the page shown in Fig. 1 does not appear, double click the My Computer icon on your desktop. In the My Computer window, double click your CD-ROM drive (Fig. 2) to load the CD (Fig. 1).

Figure 2. Locating the CD-ROM drive.

(This item is displayed on page Ixi in the print version)

[View full size image]

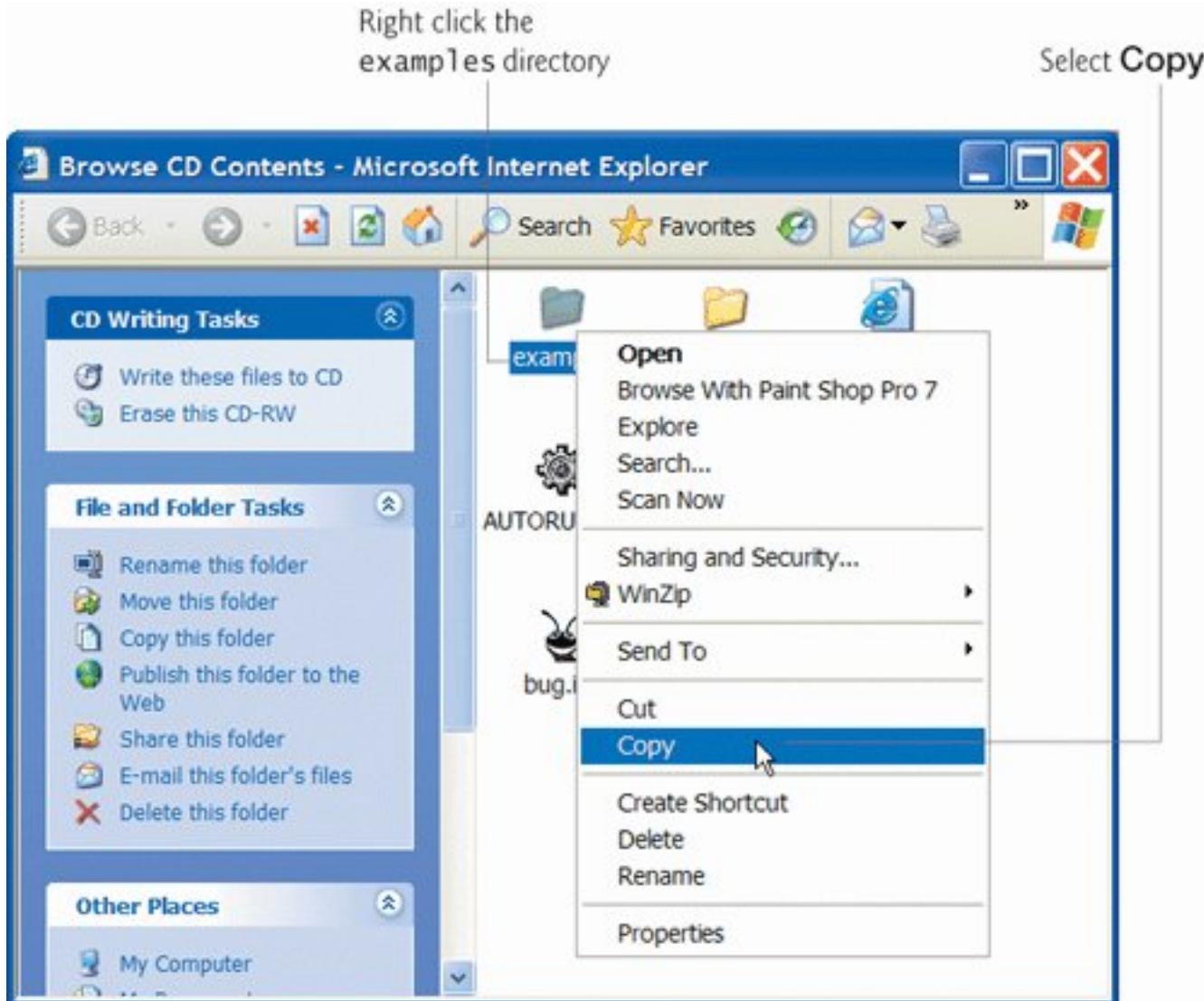


- 3.** Opening the CD-ROM directory. If the page in Fig. 1 does appear, click the Browse CD Contents link (Fig. 1) to access the CD's contents.
- 4.** Copying the examples directory. Right click the examples directory (Fig. 3), then select Copy. Next, go to My Computer and double click the C: drive. Select the Edit menu's Paste option to copy the directory and its contents from the CD to your C: drive. [Note: We save the examples to the C: drive and refer to this drive throughout the text. You may choose to save your files to a different drive based on your computer's setup, the setup in your school's lab or your personal preferences. If you are working in a computer lab, please see your instructor for more information to confirm where the examples should be saved.]

Figure 3. Copying the examples directory.

(This item is displayed on page Ixii in the print version)

[View full size image]



Right click the examples directory

Select **Copy**

The example files you copied onto your computer from the CD are read-only. Next, you will remove the read-only property so you can modify and run the examples.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page Ixiii]

You are now ready to begin your C++ studies with C++ How to Program. We hope you enjoy the book!
You can reach us easily at deitel@deitel.com. We will respond promptly.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 2]

Outline

[1.1 Introduction](#)

[1.2 What Is a Computer?](#)

[1.3 Computer Organization](#)

[1.4 Early Operating Systems](#)

[1.5 Personal, Distributed and Client/Server Computing](#)

[1.6 The Internet and the World Wide Web](#)

[1.7 Machine Languages, Assembly Languages and High-Level Languages](#)

[1.8 History of C and C++](#)

[1.9 C++ Standard Library](#)

[1.10 History of Java](#)

[1.11 FORTRAN, COBOL, Pascal and Ada](#)

[1.12 Basic, Visual Basic, Visual C++, C# and .NET](#)

[1.13 Key Software Trend: Object Technology](#)

[1.14 Typical C++ Development Environment](#)

[1.15 Notes About C++ and C++ How to Program, 5/e](#)

[1.16 Test-Driving a C++ Application](#)

[1.17 Software Engineering Case Study: Introduction to Object Technology and the UML \(Required\)](#)

1.18 Wrap-Up

1.19 Web Resources

Summary

Terminology

Self-Review Exercises

Answers to Self-Review Exercises

Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 3]

Most people are at least somewhat familiar with the exciting things computers do. Using this textbook, you will learn how to command computers to do those things. Computers (often referred to as **hardware**) are controlled by **software** (i.e., the instructions you write to command the computer to perform **actions** and make **decisions**). C++ is one of today's most popular software development languages. This text provides an introduction to programming in the version of C++ standardized in the United States through the **American National Standards Institute (ANSI)** and worldwide through the efforts of the **International Organization for Standardization (ISO)**.

Computer use is increasing in almost every field of endeavor. Computing costs have been decreasing dramatically due to rapid developments in both hardware and software technologies. Computers that might have filled large rooms and cost millions of dollars a few decades ago can now be inscribed on silicon chips smaller than a fingernail, costing a few dollars each. (Those large computers were called **mainframes** and are widely used today in business, government and industry.) Fortunately, silicon is one of the most abundant materials on earth—it's an ingredient in common sand. Silicon chip technology has made computing so economical that about a billion general-purpose computers are in use worldwide, helping people in business, industry and government, and in their personal lives.

Over the years, many programmers learned the programming methodology called structured programming. You will learn structured programming and an exciting newer methodology, object-oriented programming. Why do we teach both? Object orientation is the key programming methodology used by programmers today. You will create and work with many software objects in this text. You will discover however, that their internal structure is often built using structured-programming techniques. Also, the logic of manipulating objects is occasionally expressed with structured programming.

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 will respond promptly. 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

We hope that you will enjoy learning with C++ How to Program, Fifth Edition.

 PREV

NEXT 

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 4]

A computer consists of various devices referred to as hardware (e.g., the keyboard, screen, mouse, hard disk, memory, DVDs and processing units). The programs that run on a computer are referred to as software. Hardware costs have been declining dramatically in recent years, to the point that personal computers have become a commodity. In this book, you will learn proven methods that are reducing software development costsobject-oriented programming and (in our optional Software Engineering Case Study in [Chapters 27, 9 and 13](#)) object-oriented design.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 5]

•

Secondary storage unit. This is the long-term, high-capacity "warehousing" section of the computer. Programs or data not actively being used by the other units normally are placed on secondary storage devices, such as your hard drive, until they are again needed, possibly hours, days, months or even years later. Information in secondary storage takes much longer to access than information in primary memory, but the cost per unit of secondary storage is much less than that of primary memory. Other secondary storage devices include CDs and DVDs, which can hold hundreds of millions of characters and billions of characters, respectively.

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 5 (continued)]

1.4. Early Operating Systems

Early computers could perform only one **job** or **task** at a time. This is often called single-user **batch processing**. The computer runs a single program at a time while processing data in groups or **batches**. In these early systems, users generally submitted their jobs to a computer center on decks of punched cards and often had to wait hours or even days before printouts were returned to their desks.

Software systems called **operating systems** were developed to make using computers more convenient. Early operating systems smoothed and speeded up the transition between jobs, and hence increased the amount of work, or **throughput**, computers could process.

As computers became more powerful, it became evident that single-user batch processing was inefficient, because so much time was spent waiting for slow input/output devices to complete their tasks. It was thought that many jobs or tasks could share the resources of the computer to achieve better utilization. This is achieved by **multiprogramming**. Multiprogramming involves the simultaneous operation of many jobs that are competing to share the computer's resources. With early multiprogramming operating systems, users still submitted jobs on decks of punched cards and waited hours or days for results.

In the 1960s, several groups in industry and the universities pioneered **timesharing** operating systems. Timesharing is a special case of multiprogramming in which users access the computer through terminals, typically devices with keyboards and screens. Dozens or even hundreds of users share the computer at once. The computer actually does not run them all simultaneously. Rather, it runs a small portion of one user's job, then moves on to service the next user, perhaps providing service to each user several times per second. Thus, the users' programs appear to be running simultaneously. An advantage of timesharing is that user requests receive almost immediate responses.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 6]

Today's personal computers are as powerful as the million-dollar machines of just a few decades ago. The most powerful desktop machines called **workstations** provide individual users with enormous capabilities. Information is shared easily across computer networks, where computers called **file servers** offer a common data store that may be used by **client** computers distributed throughout the network, hence the term **client/server computing**. C++ has become widely used for writing software for operating systems, for computer networking and for distributed client/server applications. Today's popular operating systems such as UNIX, Linux, Mac OS X and Microsoft's Windows-based systems provide the kinds of capabilities discussed in this section.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 6 (continued)]

1.6. The Internet and the World Wide Web

The **Internet** global network of computers was initiated almost four decades ago with funding supplied by the U.S. Department of Defense. Originally designed to connect the main computer systems of about a dozen universities and research organizations, the Internet today is accessible by computers worldwide.

With the introduction of the **World Wide Web** which allows computer users to locate and view multimedia-based documents on almost any subject over the Internet the Internet has exploded into one of the world's premier communication mechanisms.

The Internet and the World Wide Web are surely among humankind's most important and profound creations. In the past, most computer applications ran on computers that were not connected to one another. Today's applications can be written to communicate among the world's computers. The Internet mixes computing and communications technologies. It makes our work easier. It makes information instantly and conveniently accessible worldwide. It enables individuals and local small businesses to get worldwide exposure. It is changing the way business is done. People can search for the best prices on virtually any product or service. Special-interest communities can stay in touch with one another. Researchers can be made instantly aware of the latest breakthroughs. After you master [Chapter 19](#), Web Programming, you will be able to develop Internet-based computer applications.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 7]

```
+1300042774  
+1400593419  
+1200274027
```

Machine-language programming was simply too slow, tedious and error-prone for most programmers. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations. These abbreviations formed the basis of **assembly languages**. **Translator programs** called **assemblers** were developed to convert early assembly-language programs to machine language at computer speeds. The following section of an assembly-language program also adds overtime pay to base pay and stores the result in gross pay:

```
load    basepay  
add     overpay  
store   grosspay
```

Although such code is clearer to humans, it is incomprehensible to computers until translated to machine language.

Computer usage increased rapidly with the advent of assembly languages, but programmers still had to use many instructions to accomplish even the simplest tasks. To speed the programming process, **high-level languages** were developed in which single statements could be written to accomplish substantial tasks. Translator programs called **compilers** convert high-level language programs into machine language. High-level languages allow programmers to write instructions that look almost like everyday English and contain commonly used mathematical notations. A payroll program written in a high-level language might contain a statement such as

```
grossPay = basePay + overTimePay;
```

From the programmer's standpoint, obviously, high-level languages are preferable to machine and assembly language. C, C++, Microsoft's .NET languages (e.g., Visual Basic .NET, Visual C++ .NET and C#) and Java are among the most widely used high-level programming languages.

The process of compiling a high-level language program into machine language can take a considerable amount of computer time. **Interpreter** programs were developed to execute high-level language programs directly, although much more slowly. Interpreters are popular in program development environments in which new features are being added and errors corrected. Once a program is fully developed, a compiled version can be produced to run most efficiently.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 8]

1.8. 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 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 now 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 kinds of computers (sometimes called **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.

Portability Tip 1.1



Because C is a standardized, hardware-independent, widely available language, applications written in C often can be run with little or no modification on a wide range of computer systems.

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**.

A revolution is brewing in the software community. Building software quickly, correctly and economically remains an elusive goal, and this at a time when the demand for new and more powerful software is soaring. **Objects** are essentially reusable software **components** that model items in the real world. Software developers are discovering that using a modular, object-oriented design and implementation approach can make them much more productive than they can be with previous popular programming

techniques. Object-oriented programs are easier to understand, correct and modify.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 9]

Software Engineering Observation 1.1



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

Software Engineering Observation 1.2



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

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 computerstthese tips also include general observations about how C++ achieves its high degree of portability) and **Error-Prevention Tips** (techniques for removing bugs from your programs and, more important, techniques for writing bug-free programs in the first place). Many of these are only guidelines. You will, no doubt, develop your own preferred programming style.

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

Performance Tip 1.1



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

Portability Tip 1.2



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

[◀ PREV](#)

[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 10]

Sun formally announced Java in 1995. Java generated immediate interest in the business community because of the phenomenal success of the World Wide Web. Java is now used to develop large-scale enterprise applications, to enhance the functionality of Web servers (the computers that provide the content we see in our Web browsers), to provide applications for consumer devices (such as cell phones, pagers and personal digital assistants) and for many other purposes. Current versions of C++, such as Microsoft®'s Visual C++® .NET and Borland®'s C++Builder™, have similar capabilities.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 10 (continued)]

1.11. FORTRAN, COBOL, Pascal and Ada

Hundreds of high-level languages have been developed, but only a few have achieved broad acceptance. **FORTRAN (FORmula TRANslator)** was developed by IBM Corporation in the mid-1950s to be used for scientific and engineering applications that require complex mathematical computations. FORTRAN is still widely used, especially in engineering applications.

COBOL (COmmon Business Oriented Language) was developed in the late 1950s by computer manufacturers, the U.S. government and industrial computer users. COBOL is used for commercial applications that require precise and efficient manipulation of large amounts of data. Much business software is still programmed in COBOL.

During the 1960s, many large software development efforts encountered severe difficulties. Software deliveries were typically late, costs greatly exceeded budgets and the finished products were unreliable. People began to realize that software development was a far more complex activity than they had imagined. Research in the 1960s resulted in the evolution of **structured programming**, a disciplined approach to writing programs that are clearer, easier to test and debug and easier to modify than large programs produced with previous techniques.

One of the more tangible results of this research was the development of the **Pascal** programming language by Professor Niklaus Wirth in 1971. Named after the seventeenth-century mathematician and philosopher Blaise Pascal, it was designed for teaching structured programming and rapidly became the preferred programming language in most colleges. Pascal lacks many features needed in commercial, industrial and government applications, so it has not been widely accepted in these environments.

The **Ada** programming language was developed under the sponsorship of the U.S. Department of Defense (DOD) during the 1970s and early 1980s. Hundreds of separate languages were being used to produce the DOD's massive command-and-control software systems. The DOD wanted a single language that would fill most of its needs. The Ada language was named after Lady Ada Lovelace, daughter of the poet Lord Byron. Lady Lovelace is credited with writing the world's first computer program in the early 1800s (for the Analytical Engine mechanical computing device designed by Charles Babbage). One important capability of Ada, called **multitasking**, allows programmers to specify that many activities are to occur in parallel. Java, through a technique called multithreading, also enables programmers to write programs with parallel activities. Although multithreading is not part of standard C++, it is available through various add-on class libraries.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 11]

1.12. Basic, Visual Basic, Visual C++, C# and .NET

The **BASIC** (Beginner's All-purpose Symbolic Instruction Code) programming language was developed in the mid-1960s at Dartmouth College as a means of writing simple programs. BASIC's primary purpose was to familiarize novices with programming techniques. Microsoft's Visual Basic language, introduced in the early 1990s to simplify the development of Microsoft Windows applications, has become one of the most popular programming languages in the world.

Microsoft's latest development tools are part of its corporate-wide strategy for integrating the Internet and the Web into computer applications. This strategy is implemented in Microsoft's **.NET platform**, which provides developers with the capabilities they need to create and run computer applications that can execute on computers distributed across the Internet. Microsoft's three primary programming languages are **Visual Basic .NET** (based on the original BASIC), **Visual C++ .NET** (based on C++) and **C#** (a new language based on C++ and Java that was developed expressly for the .NET platform). Developers using .NET can write software components in the language they are most familiar with and then form applications by combining those components with components written in any .NET language.

[◀ PREV](#)[NEXT ▶](#)**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 12]

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, programming languages (such as FORTRAN, COBOL, Pascal, Basic and C) were focussed 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 enhancements.

A key problem with procedural programming is that the program units do not easily mirror real-world entities effectively, so these units are not particularly reusable. It is 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 much more reusable on future projects. Using libraries of reusable componentry, such as [MFC \(Microsoft Foundation Classes\)](#), [Microsoft's .NET Framework Class Library](#) and those produced by Rogue Wave and many other software development organizations, can greatly reduce the amount of 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 1.3



Extensive class libraries of reusable software components are available over the Internet and the World Wide Web. Many of these libraries are available at no charge.

Some organizations report that the key benefit object-oriented programming gives them is not software reuse. Rather, they indicate that it tends to produce software that is more understandable, better organized and easier to maintain, modify and debug. This can be significant, because it has been estimated that 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.

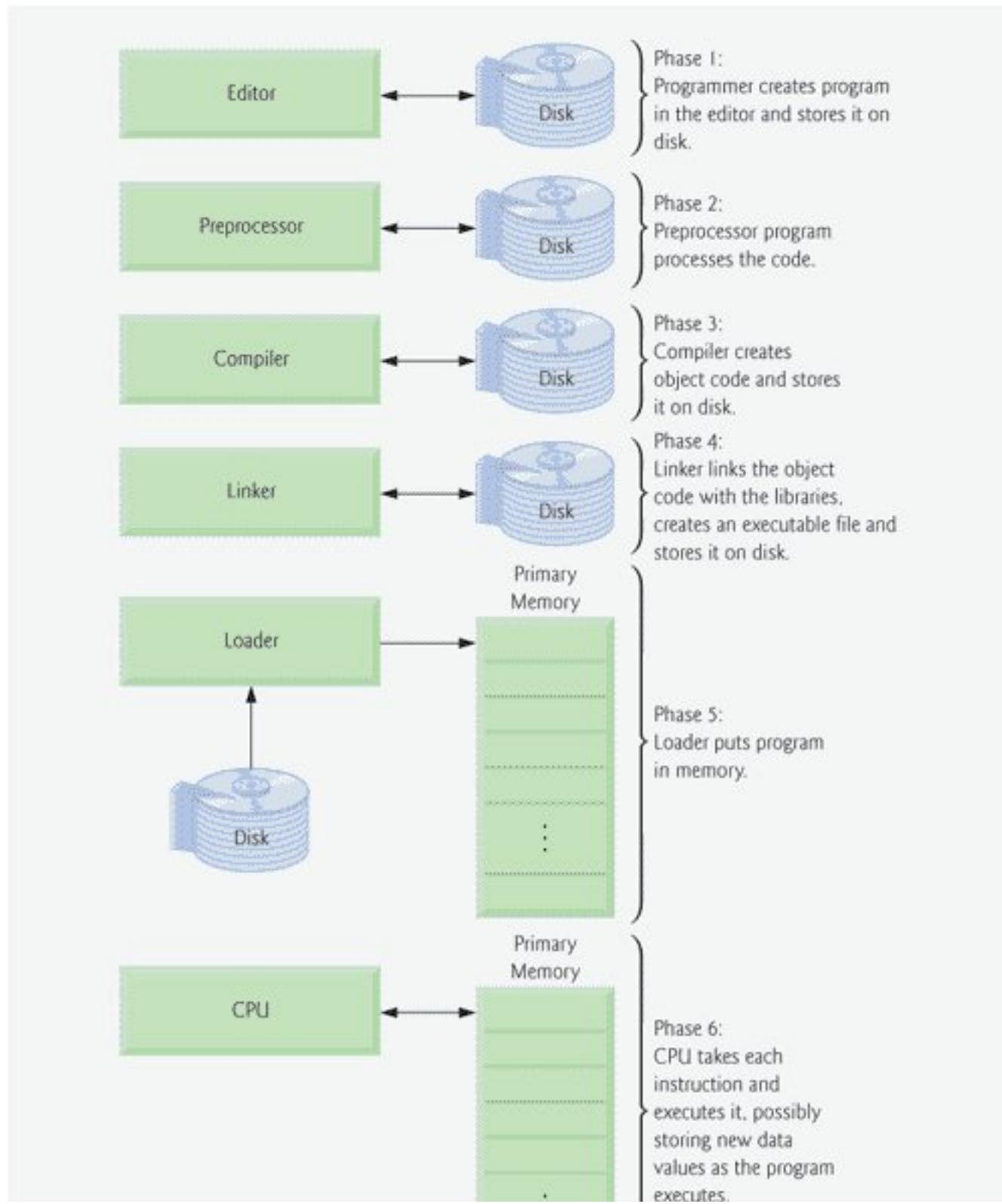
Whatever the perceived benefits of object orientation are, it is clear that object-oriented programming will be the key programming methodology for the next several decades.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 13]

Figure 1.1. Typical C++ environment.

[\[View full size image\]](#)





values as the program executes.

Phase 1: Creating a Program

Phase 1 consists of editing a file with an **editor program** (normally known simply as an **editor**). You type a C++ program (typically referred to as **source code**) using the editor, make any necessary corrections and save the program on a secondary storage device, such as your hard drive. C++ source code file names 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++ environment for more information on file-name extensions.

[Page 14]

Two editors widely used on UNIX systems are vi and emacs. C++ software packages for Microsoft Windows such as Borland C++ (www.borland.com), Metrowerks CodeWarrior (www.metrowerks.com) and Microsoft Visual C++ (www.msdn.microsoft.com/visualc/) 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. We assume the reader knows how to edit a program.

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

In phase 2, the programmer gives 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 all the preprocessor features appears in [Appendix F, Preprocessor](#). In phase 3, the compiler translates the C++ program into machine-language code (also referred to as 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

Phase 5 is called **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, under the control of its CPU, **executes** the program one instruction at a time.

Problems That May Occur at Execution Time

Programs do not always work on the first try. Each of the preceding phases can fail because of various errors that we discuss throughout the book. For example, an executing program might attempt to divide by zero (an illegal operation for whole-number arithmetic in C++). 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.

[Page 15]

Common Programming Error 1.1



Errors like 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.]

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 16]

We have included an extensive bibliography of books and papers on C++ and object-oriented programming. We also have included a C++ Resources appendix containing many Internet and Web sites relating to C++ and object-oriented programming. We have listed several Web sites in [Section 1.19](#) including links to free C++ compilers, resource sites and some fun C++ games and game programming tutorials.

Good Programming Practice 1.2



Read the manuals for the version of C++ you are using. Refer to these manuals frequently to be sure you are aware of the rich collection of C++ features and that you are using them correctly.

Good Programming Practice 1.3



Your computer and compiler are good teachers. If after reading your C++ language manual, you still are not sure how a feature of C++ works, experiment using a small "test program" and see what happens. Set your compiler options for "maximum warnings." Study each message that the compiler generates and correct the programs to eliminate the messages.

PREV

page footer

NEXT

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 17]

Running a C++ application from the Windows XP Command Prompt

1.

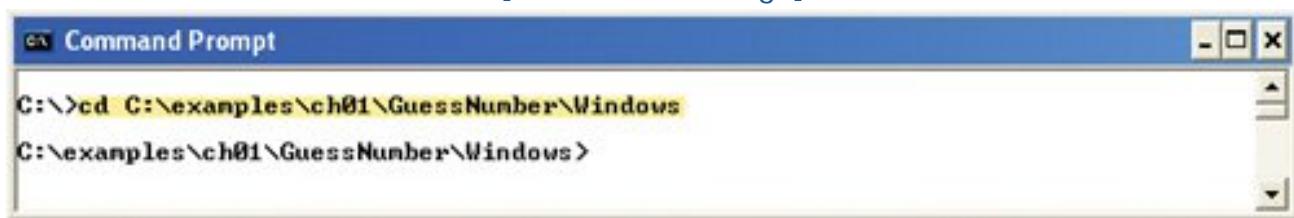
Checking your setup. Read the Before You Begin section at the beginning of this textbook to make sure that you have copied the book's examples to your hard drive correctly.

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.

Figure 1.2. Opening a Command Prompt window and changing the directory.

[\[View full size image\]](#)

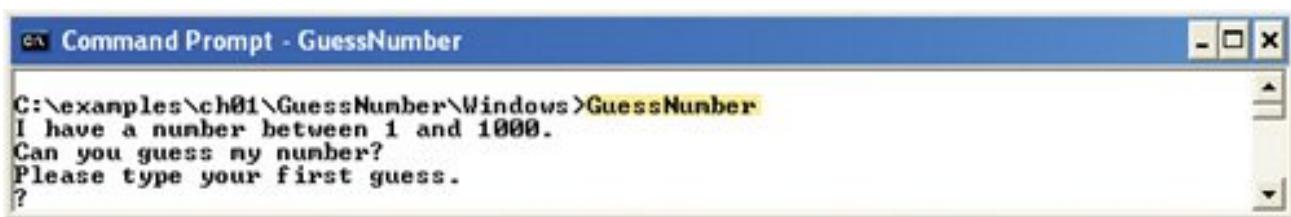


3.

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.]

Figure 1.3. Running the GuessNumber application.

[\[View full size image\]](#)



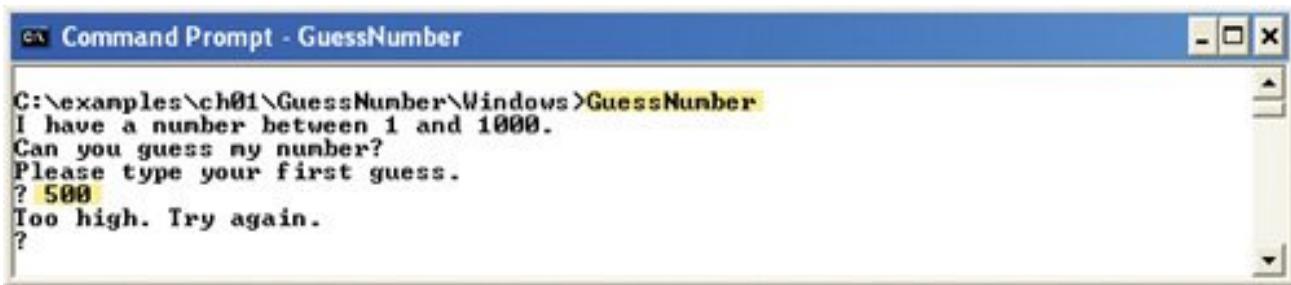
4.

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).

Figure 1.4. Entering your first guess.

(This item is displayed on page 18 in the print version)

[\[View full size image\]](#)



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 correct guess.

[Page 18]

Figure 1.5. Entering a second guess and receiving feedback.

[\[View full size image\]](#)

```
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. Once you guess the answer, the application will display "Excellent! You guessed the number!" (Fig. 1.6).

Figure 1.6. Entering additional guesses and guessing the correct number.

[View full size image]

```
Too high. Try again.
? 125
Too high. Try again.
? 62
Too high. Try again.
? 31
Too low. Try again.
? 46
Too high. Try again.
? 39
Too low. Try again.
? 42

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

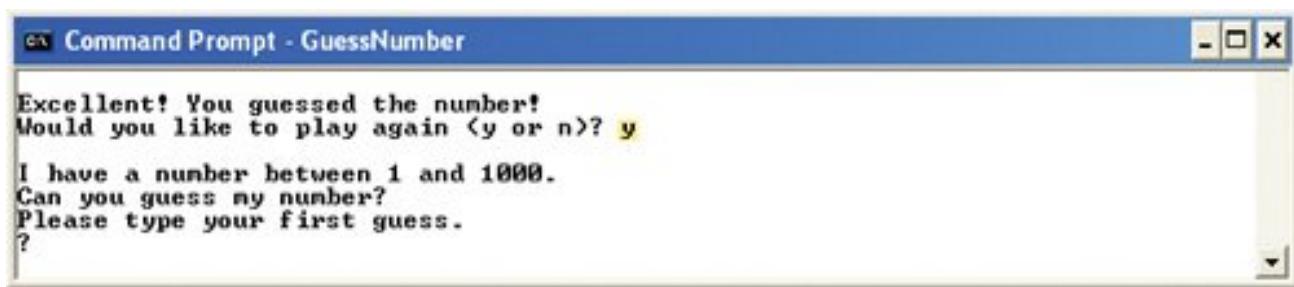
7.

Playing the game again or exiting the application. After guessing the correct number, 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 enter 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.

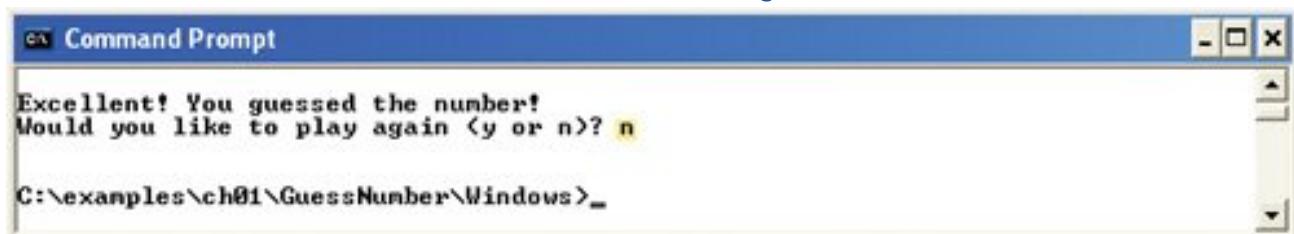
Figure 1.7. Playing the game again.

(This item is displayed on page 19 in the print version)

[View full size image]

**Figure 1.8. Exiting the game.**

(This item is displayed on page 19 in the print version)

[\[View full size image\]](#)**8.**

Close the Command Prompt window.

[\[Page 19\]](#)

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

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

1.

Locating the completed application. From a Linux shell, change to the completed GuessNumber application directory ([Fig. 1.9](#)) by typing

```
cd Examples\ch01\GuessNumber\GNU_Linux
```

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

Figure 1.9. Changing to the GuessNumber application's directory after logging in to your Linux 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, it must first be compiled by typing

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

as in Fig. 1.10. The preceding command compiles the application and produces an executable file called GuessNumber.

Figure 1.10. Compiling the GuessNumber application using the g++ command.

(This item is displayed on page 20 in the print version)

```
~/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).

[Page 20]

Figure 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.]

Figure 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.

Figure 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 the answer, the application displays "Excellent! You guessed the number!" ([Fig. 1.14](#)).

[Page 21]

Figure 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 guessing 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 enter 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.

Figure 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.  
?
```

Figure 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$
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 23]

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 processnamely, 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 objectsan 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 implementedimplementation 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 internallyas long as we know how to use the accelerator pedal, the brake pedal, the steering wheel and so on. Information hiding, as we will 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 an object-oriented design as a working software system. 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. The programmer uses 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.

[Page 24]

Classes are to objects as blueprints are to housesa 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," people in the software development community often say that the three most important factors affecting the future of software development are "reuse, reuse and reuse."

Software Engineering Observation 1.4



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 will 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)

Soon you will be writing programs in C++. How will you create the code for your programs? Perhaps, like

many beginning programmers, you will simply turn on your computer and start typing. This approach may work for small programs (like the ones we present in the early chapters of the book), but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or what if you were asked to work on a team of 1,000 software developers building the next generation of the U.S. air traffic control system? For projects so large and complex, you could not simply sit down and start writing programs.

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 **object-oriented analysis and design (OOAD)**. Experienced programmers know that analysis and design can save many hours by helping avoid an ill-planned system development approach that has to be abandoned partway through its implementation, possibly wasting considerable time, money and effort.

[Page 25]

OOAD is the generic term for the process of analyzing a problem and developing an approach for solving it. Small problems like the ones discussed in these first few chapters do not require an exhaustive OOAD process. It may be sufficient, before we begin writing C++ code, to write **pseudocode** an informal text-based means of expressing program logic. It is not actually a programming language, but we can use it as a kind of outline to guide us as we write our code. We introduce pseudocode in [Chapter 4](#).

As problems and the groups of people solving them increase in size, the methods of OOAD quickly become more appropriate than pseudocode. Ideally, a group should agree on a strictly defined process for solving its 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. Clearly, 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. In March 2003, the OMG released UML version 1.5. The UML version 2 which had been adopted and was in the process of being finalized at the time of this publication marks the first major revision since the 1997 version 1.1 standard. Many books, modeling tools and industry experts are already using the UML version 2, so we present UML version 2 terminology and notation throughout this book.

[Page 26]

What Is the UML?

The Unified Modeling Language 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, as we do throughout this book.

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.

The UML is a complex, feature-rich graphical language. In our "Software Engineering Case Study" sections on developing the software for an automated teller machine (ATM), we present a simple, concise subset of these features. We then use this subset to guide you through a first design experience with the UML, intended for novice object-oriented programmers in a first- or second-semester programming course.

This case study was carefully developed under the guidance of distinguished academic and professional reviewers. We sincerely hope you enjoy working through it. If you have the slightest question, please communicate with us at deitel@deitel.com. We will respond promptly.

Internet and Web UML Resources

For more information about the UML, refer to the following Web sites. For additional UML sites, please refer to the Internet and Web resources listed at the end of [Section 2.8](#).

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 Rationalthe successor to the Rational Software Corporation (the company that created the UML).

Recommended Readings

Many books on the UML have been published. The following recommended books provide information about object-oriented design with the UML.

- Arlow, J., and I. Neustadt. UML and the Unified Process: Practical Object-Oriented Analysis and Design. London: Pearson Education Ltd., 2002.
- Fowler, M. UML Distilled, Third Edition: A Brief Guide to the Standard Object Modeling Language. Boston: Addison-Wesley, 2004.
- Rumbaugh, J., I. Jacobson and G. Booch. The Unified Modeling Language User Guide. Reading, MA: Addison-Wesley, 1999.

For additional books on the UML, please refer to the recommended readings listed at the end of [Section 2.8](#), or visit www.amazon.com or www.bn.com. IBM Rational, formerly Rational Software Corporation, also provides a recommended-reading list for UML books at www.ibm.com/software/rational/info/technical/books.jsp.

Section 1.17 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 1.17 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.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 27 (continued)]

1.18. Wrap-Up

This chapter introduced basic hardware and software concepts, and explored C++'s role in developing distributed client/server applications. You studied the history of the Internet and the World Wide Web. 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 UMLthe industry-standard graphical language for modeling software systems.

You learned the typical steps for creating and executing a C++ application. Finally, you "test-drove" a sample C++ application similar to the types of applications you will learn to program in this book.

In the next chapter, you will create your first C++ applications. You will see several examples that demonstrate how programs display messages on the screen and obtain information from the user at the keyboard for processing. We analyze and explain each example to help you ease your way into C++ programming.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 28]

Deitel & Associates Web Sites

www.deitel.com/books/cppHTP5/index.html

The Deitel & Associates C++ How to Program, Fifth Edition site. Here you will find links to the book's examples (also included on the CD that accompanies the book) and other resources, such as our free Dive Into™ guides that help you get started with several C++ integrated development environments (IDEs).

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.

www.prenhall.com/deitel

Prentice Hall's site for Deitel publications. Here you will find detailed product information, sample chapters and Companion Web Sites containing book- and chapter-specific resources for students and instructors.

Compilers and Development Tools

www.thefreecountry.com/developercity/cccompilers.shtml

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

msdn.microsoft.com/visualc

The Microsoft Visual C++ site provides product information, overviews, supplemental materials and ordering information for the Visual C++ compiler.

www.borland.com/bcppbuilder

This is a link to the Borland C++Builder. A free command-line version is available for download.

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.

www.kai.com/C_plus_plus

This site offers the Kai C++ compiler for a 30-day free trial.

www.symbian.com/developer/development/cppdev.html

Symbian provides a C++ Developer's Pack and links to various resources, including code and development tools for C++ programmers implementing mobile applications for the Symbian operating system, which is popular on devices such as mobile phones.

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.

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

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.

<http://www.gametutorials.com/c-9-recent-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.

www.forum.nokia.com/main/0,6566,050_20,00.html

Visit this Nokia site to learn how to use C++ to program games for some Nokia wireless devices.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 30]

- The secondary storage unit is the long-term, high-capacity "warehousing" section of the computer. Programs or data not being used by the other units are normally placed on secondary storage devices (e.g., disks) until they are needed, possibly hours, days, months or even years later.
- Operating systems were developed to help make it more convenient to use computers.
- Multiprogramming involves the sharing of a computer's resources among the jobs competing for its attention, so that the jobs appear to run simultaneously.
- With distributed computing, an organization's computing is distributed over networks to the sites where the work of the organization is performed.
- Any computer can directly understand only its own machine language, which generally consist of strings of numbers that instruct computers to perform their most elementary operations.
- English-like abbreviations form the basis of assembly languages. Translator programs called assemblers convert assembly-language programs to machine language.
- Compilers translate high-level language programs into machine-language programs. High-level languages (like C++) contain English words and conventional mathematical notations.
- Interpreter programs directly execute high-level language programs, eliminating the need to compile them into machine language.
- C++ evolved from C, which evolved from two previous programming languages, BCPL and B.
- C++ is an extension of C developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. C++ enhances the C language and provides capabilities for object-oriented programming.
- Objects are reusable software components that model items in the real world. Using a modular, object-oriented design and implementation approach can make software development groups more productive than with previous programming techniques.
- C++ programs consist of pieces called classes and functions. You can program each piece 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.
- Java is used to create dynamic and interactive content for Web pages, develop enterprise applications, enhance Web server functionality, provide applications for consumer devices and more.
- FORTRAN (FORmula TRANslator) was developed by IBM Corporation in the mid-1950s for scientific and engineering applications that require complex mathematical computations.
- COBOL (COmmon Business Oriented Language) was developed in the late 1950s by a group of computer manufacturers and government and industrial computer users. COBOL is used primarily for commercial applications that require precise and efficient data manipulation.
- Ada was developed under the sponsorship of the United States Department of Defense (DOD) during the 1970s and early 1980s. Ada provides multitasking, which allows programmers to specify that many activities are to occur in parallel.
- The BASIC (Beginner's All-Purpose Symbolic Instruction Code) programming language was developed in the mid-1960s at Dartmouth College as a language for writing simple programs. BASIC's primary purpose was to familiarize novices with programming techniques.
- Microsoft's Visual Basic was introduced in the early 1990s to simplify the process of developing Microsoft Windows applications.
- Microsoft has a corporate-wide strategy for integrating the Internet and the Web into computer applications. This strategy is implemented in Microsoft's .NET platform.
- The .NET platform's three primary programming languages are Visual Basic .NET (based on the original

BASIC), Visual C++ .NET (based on C++) and C# (a new language based on C++ and Java that was developed expressly for the .NET platform).

[Page 31]

- .NET developers can write software components in their preferred language, then form applications by combining those components with components written in any .NET language.
- 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.
- C++ source code file names often end with the .cpp, .cxx, .cc or .C extensions.
- A preprocessor program executes automatically before the compiler's translation phase begins. The C++ preprocessor obeys commands called preprocessor directives, which indicate that certain manipulations are to be performed on the program before compilation.
- The object code produced by the C++ compiler typically contains "holes" due to references to functions and data defined elsewhere. A linker links the object code with the code for the missing functions to produce an executable image (with no missing pieces).
- The loader takes the executable image from disk and transfers it to memory for execution.
- Most programs in C++ input and/or output data. Data is often input from `cin` (the standard input stream) which is normally the keyboard, but `cin` can be redirected from another device. Data is often output to `cout` (the standard output stream), which is normally the computer screen, but `cout` can be redirected to another device. The `cerr` stream is used to display error messages.
- The Unified Modeling Language (UML) is a graphical language that allows people who build systems to represent their object-oriented designs in a common notation.
- Object-oriented design (OOD) models software components in terms of real-world objects. It takes advantage of class relationships, where objects of a certain class have the same characteristics. It also takes advantage of inheritance relationships, where newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. OOD encapsulates data (attributes) and functions (behavior) into objects the data and functions of an object are intimately tied together.
- Objects have the property of information hiding objects normally are not allowed to know how other objects are implemented.
- Object-oriented programming (OOP) allows programmers to implement object-oriented designs as working systems.
- C++ programmers create their own user-defined types called classes. Each class contains data (known as data members) and the set of functions (known as member functions) that manipulate that data and provide services to clients.
- Classes can have relationships with other classes. 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.
- An instance of a class is called an object.
- With object technology, programmers can build much of the software they will need by combining standardized, interchangeable parts called classes.
- The process of analyzing and designing a system from an object-oriented point of view is called object-oriented analysis and design (OOAD).

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 32]

arithmetic and logic unit (ALU)

assembler

assembly language

association

attribute of an object

BASIC (Beginner's All-Purpose Instruction Code)

batch processing

behavior of an object

Booch, Grady

C

C++

C++ Standard Library

C#

central processing unit (CPU)

class

client

client/server computing

COBOL (COmmon Business Oriented Language)

compile phase

compiler

component

computer

computer program

computer programmer

core memory

data

data member

debug

decision

design

distributed computing

dynamic content

edit phase

editor

encapsulate

executable image

execute phase

extensible

file server

FORTRAN (FORmula TRANslator)

function

hardware

hardware platform

high-level language

information hiding

inheritance

input device

input unit

input/output (I/O)

instantiate

interface

International Organization for Standardization (ISO)

Internet

interpreter

Jacobson, Ivar

Java

link phase

linker

live-code approach

load phase

loader

local area networks (LANs)

logical unit

machine dependent

machine independent

machine language

member function

memory

memory unit

method

MFC (Microsoft Foundation Classes)

Microsoft's .NET Framework Class Library

multiprocessor

multiprogramming

multitasking

multithreading

.NET platform

object

object code

Object Management Group (OMG)

object-oriented analysis and design (OOAD)

object-oriented design (OOD)

object-oriented programming (OOP)

operating system

operation

output device

output unit

personal computing

platform

portable

preprocess phase

preprocessor directives

primary memory

procedural programming

pseudocode

Rational Software Corporation

requirements document

Rumbaugh, James

runtime errors or execution-time errors

secondary storage unit

software

software reuse

source code

structured programming

structured systems analysis and design

supercomputer

task

throughput

timesharing

translation

translator program

Unified Modeling Language (UML)

user-defined type

Visual Basic .NET

Visual C++ .NET

workstation

World Wide Web

 PREV

NEXT 

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 34]

• OOD also takes advantage of _____ relationships, where new classes of objects are derived by absorbing characteristics of existing classes, then adding unique characteristics of their own.

• _____ is a graphical language that allows people who design software systems to use an industry-standard notation to represent them.

• The size, shape, color and weight of an object are considered _____ of the object.

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 34 (continued)]

Answers to Self-Review Exercises

- 1.1** a) Apple. b) IBM Personal Computer. c) programs. d) input unit, output unit, memory unit, arithmetic and logic unit, central processing unit, secondary storage unit. e) machine languages, assembly languages and high-level languages. f) compilers. g) UNIX. h) Pascal. i) multitasking.
- 1.2** a) editor. b) preprocessor. c) linker. d) loader.
- 1.3** a) information hiding. b) classes. c) associations. d) object-oriented analysis and design (OOAD). e) inheritance. f) The Unified Modeling Language (UML). g) attributes.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 35]

1.8

Distinguish between the terms fatal error and nonfatal error. Why might you prefer to experience a fatal error rather than a nonfatal error?

1.9

Give a brief answer to each of the following questions:

a.

Why does this text discuss structured programming in addition to object-oriented programming?

b.

What are the typical steps (mentioned in the text) of an object-oriented design process?

c.

What kinds of messages do people send to one another?

d.

Objects send messages to one another across well-defined interfaces. What interfaces does a car radio (object) present to its user (a person object)?

1.10

You are probably wearing on your wrist one of the world's most common types of objectsa watch. Discuss how each of the following terms and concepts applies to the notion of a watch: object, attributes, behaviors, class, inheritance (consider, for example, an alarm clock), abstraction, modeling, messages, encapsulation, interface, information hiding, data members and member functions.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 37]

Outline

[2.1 Introduction](#)

[2.2 First Program in C++: Printing a Line of Text](#)

[2.3 Modifying Our First C++ Program](#)

[2.4 Another C++ Program: Adding Integers](#)

[2.5 Memory Concepts](#)

[2.6 Arithmetic](#)

[2.7 Decision Making: Equality and Relational Operators](#)

[2.8 \(Optional\) Software Engineering Case Study: Examining the ATM Requirements Document](#)

[2.9 Wrap-Up](#)

Summary

Terminology

Self-Review Exercises

Answers to Self-Review Exercises

Exercises

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 37 (continued)]

2.1. Introduction

We now introduce C++ programming, which facilitates a disciplined approach to program design. Most of the C++ programs you will study in this book process information and display results. 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 simply display messages on the screen. The next is a program that 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 showing you how to compare two numbers, then display messages based on the comparison results. We analyze each program one line at a time to help you ease your way into C++ programming. To help you apply the skills you learn here, we provide many programming problems in the chapter's exercises.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 38]

Good Programming Practice 2.1



Every program should begin with a comment that describes the purpose of the program, author, date and time. (We are not showing the author, date and time in this book's programs because this information would be redundant.)

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 1.14](#)). 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. The program in [Fig. 2.1](#) outputs data to the screen, as we will soon see. We discuss header files in more detail in [Chapter 6](#) and explain the contents of `iostream` in [Chapter 15](#).

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 the compiler cannot recognize references to the stream components (e.g., `cout`).

Line 4 is simply a blank line. Programmers 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**. White-space characters are normally ignored by the compiler. In this chapter and several that follow, we discuss conventions for using white-space characters to enhance program readability.

Good Programming Practice 2.2



Use blank lines and space characters to enhance program readability.

[Page 39]

Line 5

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

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

Line 6

```
int main()
```

is a part of every C++ program. The parentheses after `main` indicate that `main` is a program building block called a **function**. C++ programs typically consist of one or more functions and classes (as you will learn 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 (whole number) value. A **keyword** is a word in code that is reserved by C++ for a specific use. The complete list of C++ keywords can be found in [Fig. 4.3](#). We will explain what it means for a function to "return a value" when we demonstrate how to create your own functions in [Section 3.5](#) and when we study functions in greater depth in [Chapter 6](#). For now, simply include the keyword `int` to the left of `main` in each of your programs.

The **left brace**, `{`, (line 7) must begin the **body** of every function. A corresponding **right brace**, `}`, (line 12) must end each function's body. Line 8

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

instructs the computer to **perform an action**namely, to print the **string** of characters contained between the double quotation marks. A string is sometimes called a **character string**, a **message** or a **string literal**. We refer to characters between double quotation marks simply as **strings**. 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](#), Stream Input/Output.

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 24](#), 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. 2.13](#), 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 right of the operator, the right **operand**, is inserted in the output stream. Notice that the operator points in the direction of where the data goes. The characters of the right operand normally print exactly as they appear between the double quotes. Notice, however, that the characters `\n` are not printed on the screen. 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** (i.e., the current screen-position indicator) to move to the beginning of the next line on the screen. Some other common escape sequences are listed in [Fig. 2.2](#).

[Page 40]

Figure 2.2. Escape sequences.

Escape sequence	Description
<code>\n</code>	Newline. Position the screen cursor to the beginning of the next line.
<code>\t</code>	Horizontal tab. Move the screen cursor to the next tab stop.
<code>\r</code>	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line.
<code>\a</code>	Alert. Sound the system bell.
<code>\\\</code>	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.) The **syntax** of a programming language specifies the rules for creating a proper program in that language. A **syntax error** occurs when the compiler encounters code that violates C++'s language rules (i.e., its syntax). The compiler normally issues an error message to help the programmer locate and fix the incorrect code. Syntax errors are also called **compiler errors**, **compile-time errors** or **compilation errors**, because the compiler detects them during the compilation phase. You will be unable to execute your program until you correct all the syntax errors in it. As you will see, some compilation errors are not syntax errors.

Line 10

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

is one of several means we will 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 2.3



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.

[Page 41]

Good Programming Practice 2.4



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 helps make the program easier to read.

Good Programming Practice 2.5



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.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 42]

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

```
1 // Fig. 2.4: fig02_04.cpp
2 // Printing multiple lines of text with a single statement.
3 #include <iostream> // allows program to output data to the screen
4
5 // function main begins program execution
6 int main()
7 {
8     std::cout << "Welcome\n to\n C++! \n";
9
10    return 0; // indicate that program ended successfully
11
12 } // end function main
```



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 43]

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

This makes the program less readable and prevents us from providing comments that describe each variable's purpose. If more than one name is declared in a declaration (as shown here), the names are separated by commas (,). This is referred to as a **comma-separated list**.

Good Programming Practice 2.6



Place a space after each comma (,) to make programs more readable.

Good Programming Practice 2.7



Some programmers prefer to declare each variable on a separate line. This format allows for easy insertion of a descriptive comment next to each declaration.

We will 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**, **primitive 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.

[Page 44]

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 2.8



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

Good Programming Practice 2.9



Avoid using abbreviations in identifiers. This promotes program readability.

Good Programming Practice 2.10



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

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

could have been placed immediately before line 14

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

the declaration in line 10

```
int number2; // second integer to add
```

could have been placed immediately before line 17

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

and the declaration in line 11

```
int sum; // sum of number1 and number2
```

could have been placed immediately before line 19

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

[Page 45]

Good Programming Practice 2.11



Always place a blank line between a declaration and adjacent executable statements. This makes the declarations stand out in the program and contributes to program clarity.

Good Programming Practice 2.12



If you prefer to place declarations at the beginning of a function, separate them from the executable statements in that function with one blank line to highlight where the declarations end and the executable statements begin.

Line 13

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

prints the string `Enter first integer:` (also known as a string literal or a **literal**) on the screen. This message is called a **prompt** because it directs the user to take a specific action. We like to pronounce the preceding statement as "`std::cout` gets the 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 (copies) this number (or **value**) to the variable `number1`. Any subsequent references to `number1` in this program will use this same value.

The `std::cout` and `std::cin` stream objects facilitate interaction between the user and the computer. Because this interaction resembles a dialog, it is often called **conversational computing** or **interactive computing**.

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
```

[Page 46]

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

Good Programming Practice 2.13



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

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` 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 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 users can create their own data types called classes (we introduce this capability in [Chapter 3](#) and explore it in depth in [Chapters 9 and 10](#)). Users 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](#)).

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 47]

Figure 2.6. Memory location showing the name and value of variable `number1`.



Whenever a value is placed in a memory location, the value overwrites the previous value in that location; thus, placing a new value into a memory location is said to be **destructive**.

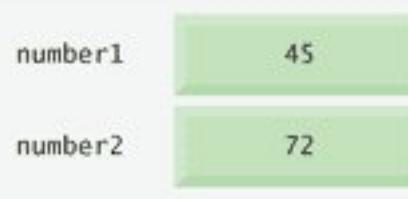
Returning to our addition program, when the statement

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

in line 17 is executed, suppose the user enters the value 72. This value is placed into location `number2`, and memory appears as in [Fig. 2.7](#). Note that these locations are not necessarily adjacent in memory.

Figure 2.7. Memory locations after storing values for `number1` and `number2`.

[[View full size image](#)]



Once the program has obtained values for `number1` and `number2`, it adds these values and places the sum into variable `sum`. The statement

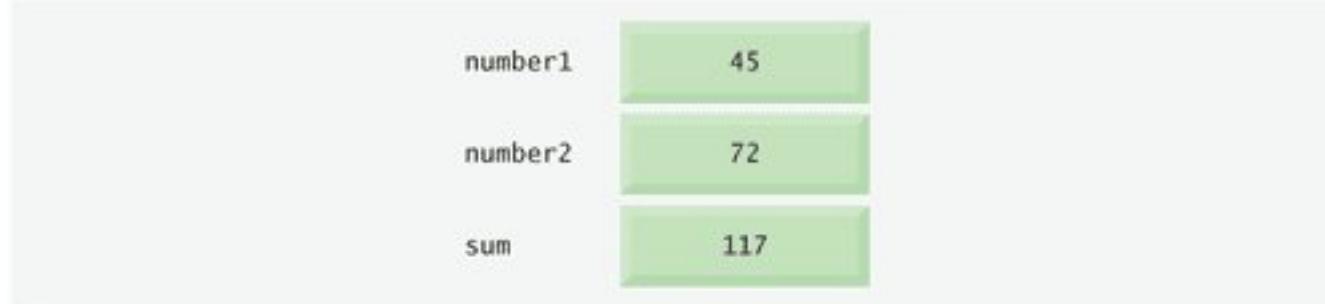
```
sum = number1 + number2; // add the numbers; store result in sum
```

that performs the addition also replaces whatever value was stored in `sum`. This occurs when the calculated sum of `number1` and `number2` is placed into location `sum` (without regard to what value may already be in `sum`; that value is lost). After `sum` is calculated, memory appears as in [Fig. 2.8](#). Note that

the values of `number1` and `number2` appear exactly as they did before they were used in the calculation of `sum`. These values were used, but not destroyed, as the computer performed the calculation. Thus, when a value is read out of a memory location, the process is **nondestructive**.

Figure 2.8. Memory locations after calculating and storing the sum of `number1` and `number2`.

[View full size image]



PREV

page footer

NEXT

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 49]

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. Thus, parentheses may be used to force the order of evaluation to occur in any sequence desired by the programmer. 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.

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 will see that some operators associate from right to left. [Figure 2.10](#) 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](#).

Figure 2.10. 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.
*	Multiplication	Evaluated second. If there are several, they are evaluated left to right.
/	Division	
%	Modulus	
+	Addition	Evaluated last. If there are several, they are evaluated left to right.
-	Subtraction	

[Page 50]

Sample Algebraic and C++ Expressions

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

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

$$\text{C++: } m = (a + b + c + d + e) / 5;$$

The parentheses are required because division has higher precedence than addition. The entire quantity $(a + b + c + d + e)$ is to be divided by 5. If the parentheses are erroneously omitted, we obtain $a + b + c + d + e / 5$, which evaluates incorrectly as

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

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

Algebra: $y = mx + b$

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

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

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

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

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



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

Evaluation of a Second-Degree Polynomial

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

$y = a * x * x + b * x + c;$



The circled numbers under the statement indicate the order in which C++ applies the operators. There is no arithmetic operator for exponentiation in C++, so we have represented x^2 as $x * x$. We will soon discuss the standard library function `pow` ("power") that performs exponentiation. Because of some subtle issues related to the data types required by `pow`, we defer a detailed explanation of `pow` until [Chapter 6](#).

Common Programming Error 2.4

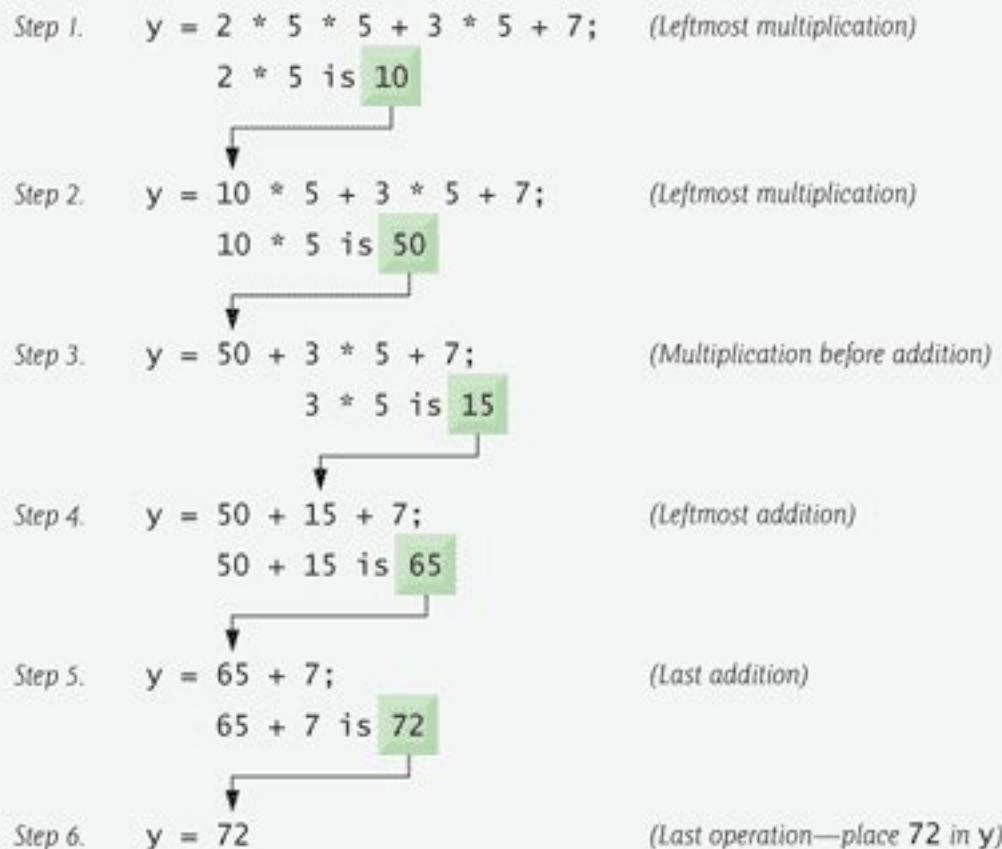


Some programming languages use operators `**` or `^` to represent exponentiation. C++ does not support these exponentiation operators; using them for exponentiation results in errors.

Suppose variables `a`, `b`, `c` and `x` in the preceding second-degree polynomial are initialized as follows: `a = 2`, `b = 3`, `c = 7` and `x = 5`. Figure 2.11 illustrates the order in which the operators are applied.

Figure 2.11. Order in which a second-degree polynomial is evaluated.

[View full size image]



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

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

Good Programming Practice 2.14



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

[◀ PREV](#)

[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 52]

Conditions in `if` statements can be formed by using the **equality operators** and **relational operators** summarized in Fig. 2.12. 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.

Figure 2.12. Equality and relational operators.

Standard algebraic equality or relational operator	C++ equality or relational operator	Sample C++ condition	Meaning of C++ condition
Relational operators			
>	>	<code>x > y</code>	<code>x</code> is greater than <code>y</code>
<	<	<code>x < y</code>	<code>x</code> is less than <code>y</code>
\geq	<code>>=</code>	<code>x >= y</code>	<code>x</code> is greater than or equal to <code>y</code>
\leq	<code><=</code>	<code>x <= y</code>	<code>x</code> is less than or equal to <code>y</code>
Equality operators			
=	<code>==</code>	<code>x == y</code>	<code>x</code> is equal to <code>y</code>
\neq	<code>!=</code>	<code>x != y</code>	<code>x</code> is not equal to <code>y</code>

Common Programming Error 2.5



A syntax error will occur if any of the operators `==`, `!=`, `>=` and `<=` appears with spaces between its pair of symbols.

Common Programming Error 2.6



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

Common Programming Error 2.7



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 2.13](#) shows the program and the input/output dialogs of three sample executions.

Figure 2.13. Equality and relational operators.

(This item is displayed on pages 53 - 54 in the print version)

```

1 // Fig. 2.13: fig02_13.cpp
2 // Comparing integers using if statements, relational operators
3 // and equality operators.
4 #include <iostream> // allows program to perform input and output
5
6 using std::cout; // program uses cout
7 using std::cin; // program uses cin
8 using std::endl; // program uses endl
9
10 // function main begins program execution
11 int main()
12 {
13     int number1; // first integer to compare
14     int number2; // second integer to compare
15
16     cout << "Enter two integers to compare: "; // prompt user for data

```

```
17     cin >> number1 >> number2; // read two integers from user
18
19     if ( number1 == number2 )
20         cout << number1 << " == " << number2 << endl;
21
22     if ( number1 != number2 )
23         cout << number1 << " != " << number2 << endl;
24
25     if ( number1 < number2 )
26         cout << number1 << " < " << number2 << endl;
27
28     if ( number1 > number2 )
29         cout << number1 << " > " << number2 << endl;
30
31     if ( number1 <= number2 )
32         cout << number1 << " <= " << number2 << endl;
33
34     if ( number1 >= number2 )
35         cout << number1 << " >= " << number2 << endl;
36
37     return 0; // indicate that program ended successfully
38
39 } // 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
```

[Page 53]

Lines 68

```
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.]

[Page 54]

Good Programming Practice 2.15



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

Lines 1314

```
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 multiple 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 at lines 1920

```
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 at 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 at lines 22, 25, 28, 31 and 34, the corresponding body statement displays an appropriate line of text.

Notice that each `if` statement in Fig. 2.13 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**).

Good Programming Practice 2.16



Indent the statement(s) in the body of an `if` statement to enhance readability.

[Page 55]

Good Programming Practice 2.17



For readability, there should be no more than one statement per line in a program.

Common Programming Error 2.8



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 would become a statement in sequence with the `if` statement and would always execute, often causing the program to produce incorrect results.

Note the use of white space in Fig. 2.13. Recall that white-space characters, such as tabs, newlines and spaces, are normally ignored by the compiler. So, statements may be split over several lines and may be spaced according to the programmer's preferences. It is a syntax error to split identifiers, strings (such as "hello") and constants (such as the number 1000) over several lines.

Common Programming Error 2.9



It is a syntax error to split an identifier by inserting white-space characters (e.g., writing `main` as `ma in`).

Good Programming Practice 2.18



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

Figure 2.14 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 will soon see, first assigns 0 to `y` then assigns the result of that assignment to `x`.

Figure 2.14. 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	
<	\leq	>	\geq	left to right	relational
\equiv	\neq			left to right	equality
=				right to left	assignment

[Page 56]

Good Programming Practice 2.19



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

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 57]

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. 2.15):

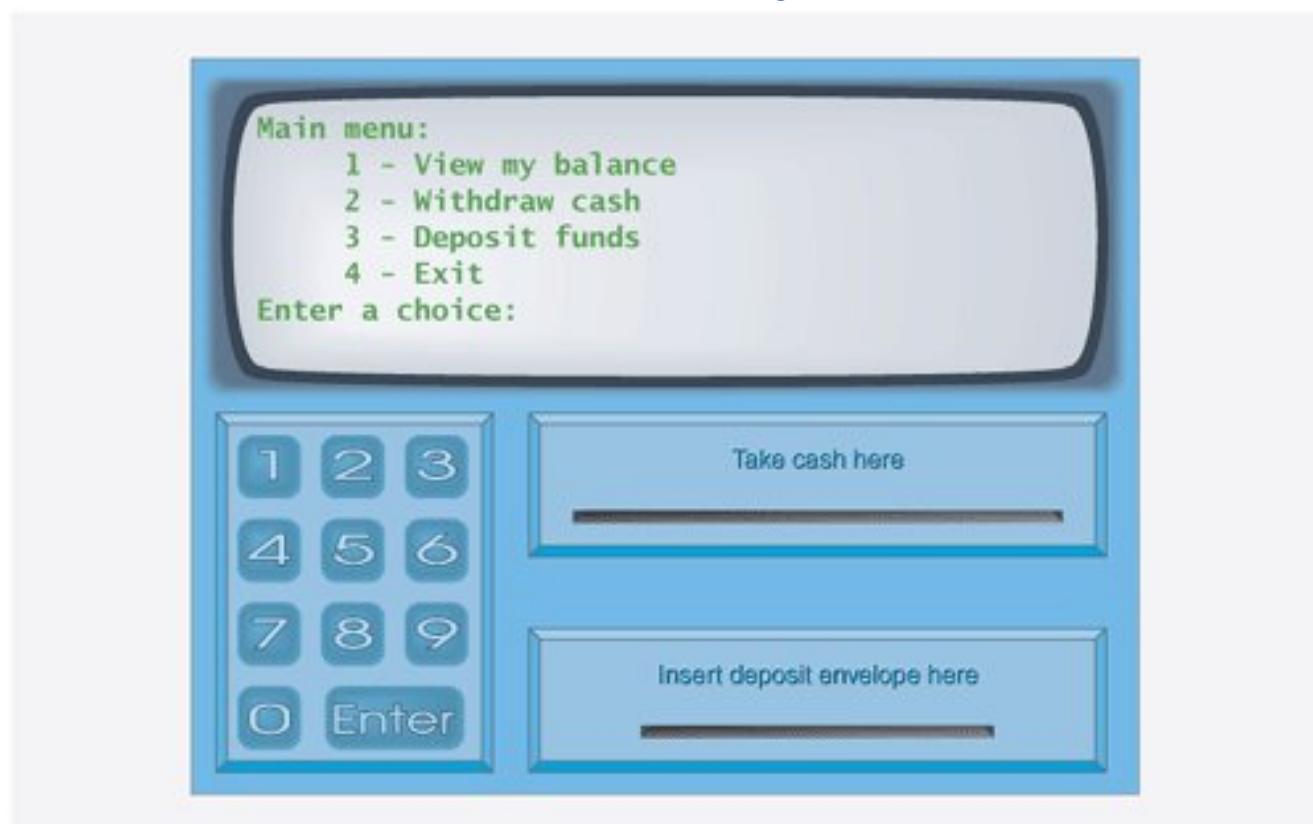
[Page 58]

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. 2.16). 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.

Figure 2.16. ATM main menu.

[View full size image]



After the ATM authenticates the user, the main menu (Fig. 2.16) 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.

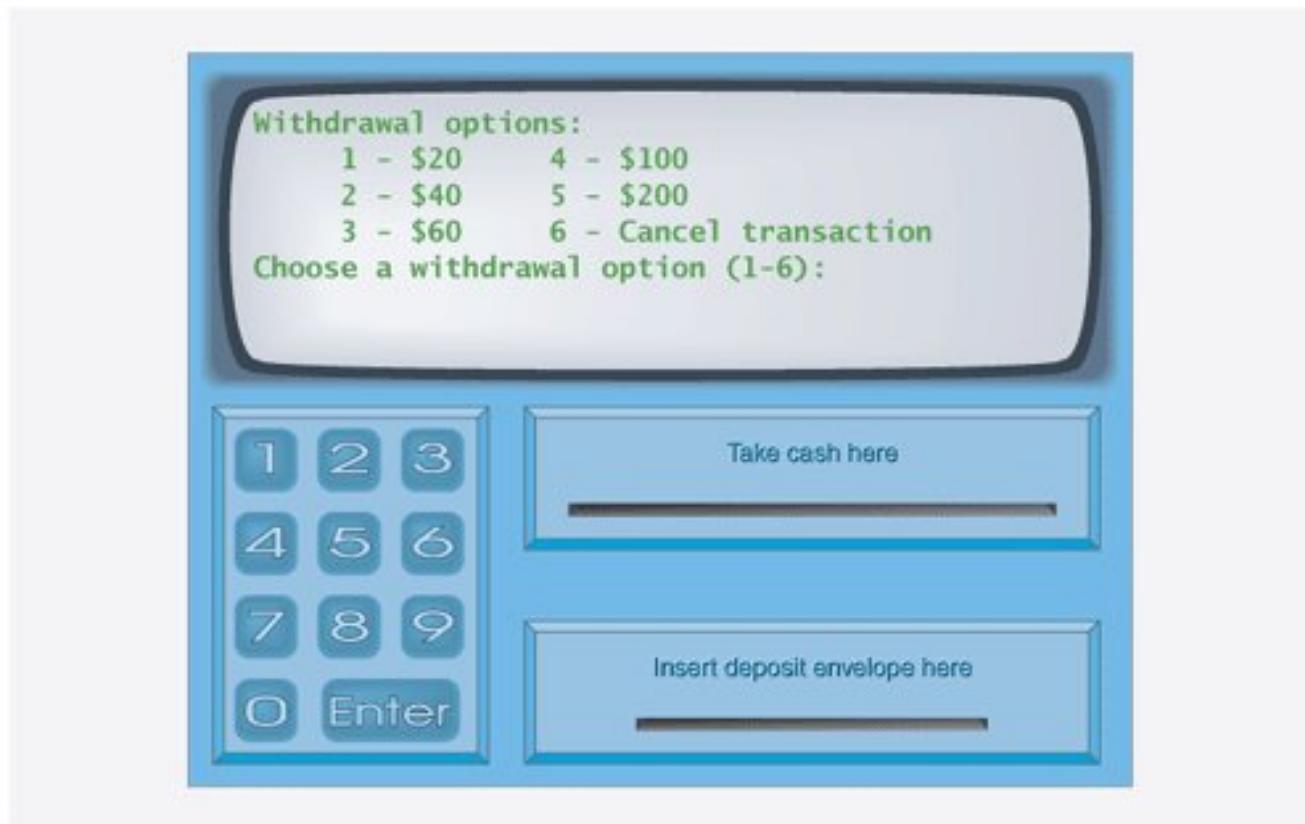
The following actions occur when the user enters 2 to make a withdrawal:

6. The screen displays a menu (shown in Fig. 2.17) 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).

[Page 59]

Figure 2.17. ATM withdrawal menu.

[\[View full size image\]](#)



7. The user enters a menu selection (16) using the keypad.
8. 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. 2.16) and waits for user input.
9. 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.

10. The ATM debits (i.e., subtracts) the withdrawal amount from the user's account balance in the bank's database.
11. The cash dispenser dispenses the desired amount of money to the user.
12. 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:

13. The screen prompts the user to enter a deposit amount or to type 0 (zero) to cancel the transaction.
14. 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., $125 \div 100 = 1.25$).]

[Page 60]

15. 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. 2.16](#)) and waits for user input.
16. The screen displays a message telling the user to insert a deposit envelope into the deposit slot.
17. 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. 2.16](#)) 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 document. 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 document 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.

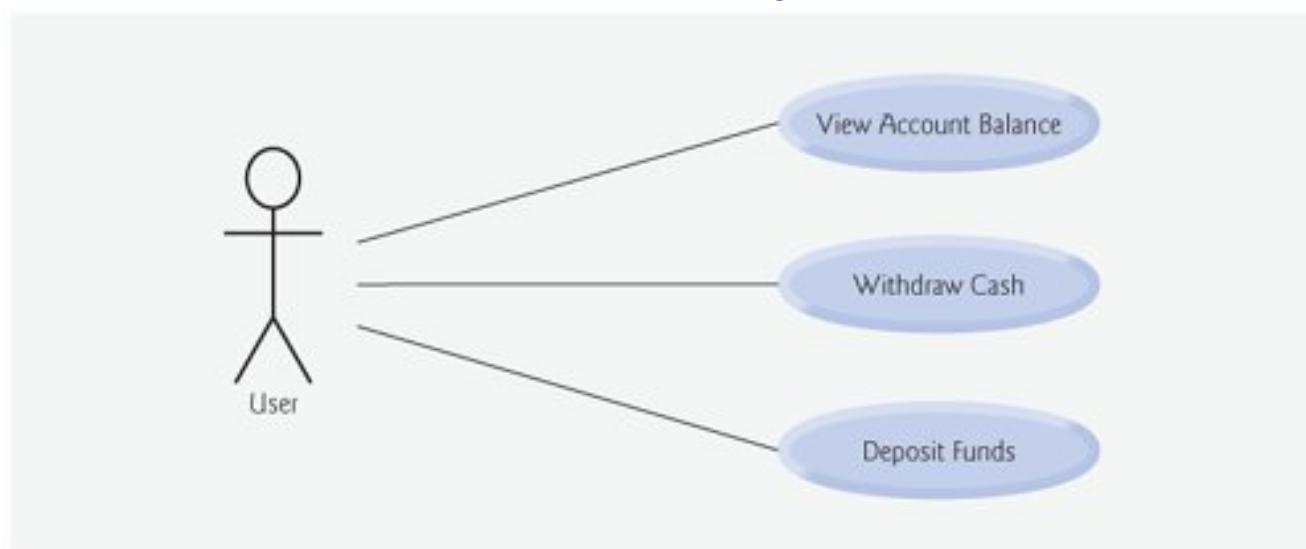
The analysis stage of the software life cycle focuses on defining the problem to be solved. When designing any system, one must certainly solve the problem right, but of equal importance, one must solve the right problem. Systems analysts collect the requirements that indicate the specific problem to solve. Our requirements document describes our ATM system in sufficient detail that you do not need to go through an extensive analysis stageit 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 use cases (Fig. 2.18).

[Page 61]

Figure 2.18. Use case diagram for the ATM system from the user's perspective.

[View full size image]



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 document; 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 document. 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 2.18](#) 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 document, which states, "ATM users should be able to view their account balance, withdraw cash and deposit funds." Therefore, 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 2.18](#) 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.

[Page 62]

Software engineers (more precisely, systems designers) must analyze the requirements document or a set of use cases and design the system before programmers implement it in a particular programming language. During the analysis stage, systems designers focus on understanding the requirements document 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. 2.15](#)), 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 H](#), UML 2: Additional Diagram Types

1.

Use case diagrams, such as the one in [Fig. 2.18](#), 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 will study in [Section 3.11](#), model the classes, or "building blocks," used in a system. Each noun or "thing" described in the requirements document 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 will study in [Section 3.11](#), 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).

[Page 63]

4.

Activity diagrams, which you will also study in [Section 5.11](#), 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 will learn 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 will learn 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 document. We accomplish this by extracting key nouns and noun phrases from the requirements document. Using these classes, we develop our first draft of the class diagram that models the structure of our ATM system.

Internet and Web Resources

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

www-306.ibm.com/software/rational/uml/

Lists frequently asked questions about the UML, provided by IBM Rational.

www.softdocwiz.com/Dictionary.htm

Hosts the Unified Modeling Language Dictionary, which lists and defines all terms used in the UML.

www-306.ibm.com/software/rational/offers/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 Rose® XDE Developer.

www.embarcadero.com/products/describe/index.html

Provides a 15-day trial license for the Embarcadero Technologies® UML modeling tool Describe.™

www.borland.com/together/index.html

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

www.ilogix.com/rhapsody/rhapsody.cfm

Provides a free 30-day license to download a trial version of I-Logix Rhapsody® a UML 2-based model-driven development environment.

argouml.tigris.org

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

www.objectsbydesign.com/books/booklist.html

Lists books on the UML and object-oriented design.

[Page 64]

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?"

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, Third Edition. Boston: Addison-Wesley, 2004.

Eriksson, H., et al. UML 2 Toolkit. New York: John Wiley, 2003.

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, Second Edition. Upper Saddle River, NJ: Prentice Hall, 2002.

Roques, P. UML in Practice: The Art of Modeling Software Systems Demonstrated Through Worked Examples and Solutions. New York: John Wiley, 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.

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. 2.18 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

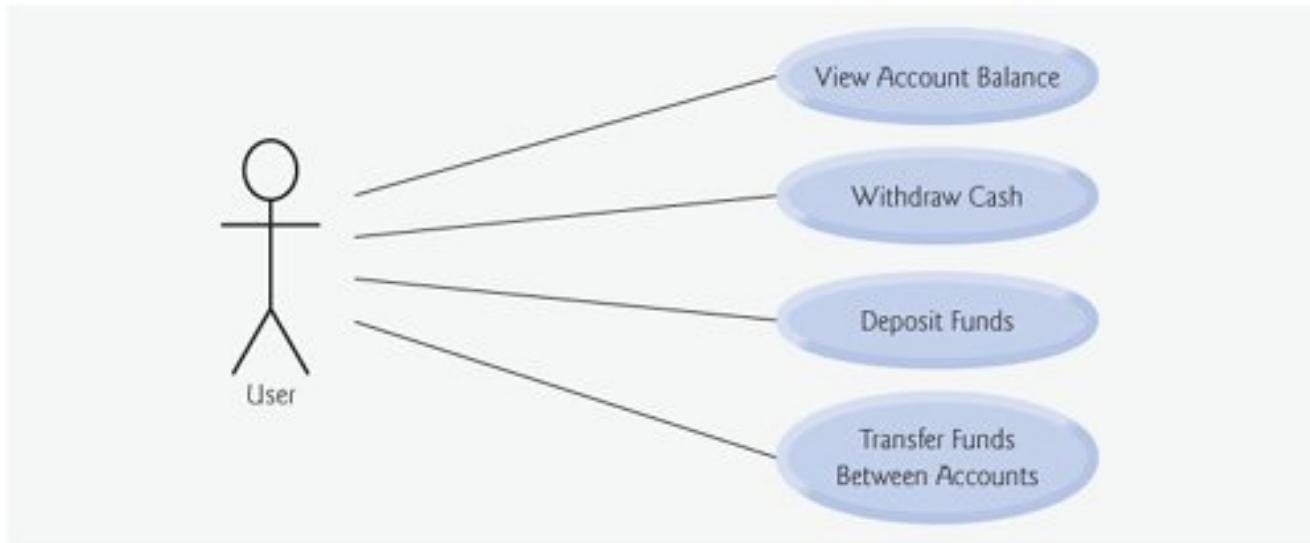
Answers to Software Engineering Case Study Self-Review Exercises

[Page 65]

- 2.1** Figure 2.19 contains a use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

Figure 2.19. Use case diagram for a modified version of our ATM system that also allows users to transfer money between accounts.

[View full size image]



2.2 b.

2.3 d.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 65 (continued)]

2.9. Wrap-Up

You learned many important 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 will 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 will learn how to implement your own classes and use objects of those classes in applications.

[◀ PREV](#)[NEXT ▶](#)

[page footer](#)

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 66]

- The line `#include <iostream>` tells the C++ preprocessor to include the contents of the input/output stream header file in the program. This file contains information necessary to compile programs that use `std::cin` and `std::cout` and operators `<<` and `>>`.
- Programmers use white space (i.e., blank lines, space characters and tab characters) to make programs easier to read. White-space characters are ignored by the compiler.
- C++ programs begin executing at the function `main`, even if `main` does not appear first in the program.
- The keyword `int` to the left of `main` indicates that `main` "returns" an integer value.
- A left brace, `{`, must begin the body of every function. A corresponding right brace, `}`, must end each function's body.
- A string in double quotes is sometimes referred to as a character string, message or string literal. White-space characters in strings are not ignored by the compiler.
- Every statement must end with a semicolon (also known as the statement terminator).
- Output and input in C++ are accomplished with streams of characters.
- The output stream object `std::cout` normally connected to the screen is used to output data. Multiple data items can be output by concatenating stream insertion (`<<`) operators.
- The input stream object `std::cin` normally connected to the keyboard is used to input data. Multiple data items can be input by concatenating stream extraction (`>>`) operators.
- The `std::cout` and `std::cin` stream objects facilitate interaction between the user and the computer. Because this interaction resembles a dialog, it is often called conversational computing or interactive computing.
- The notation `std::cout` specifies that we are using a name, in this case `cout`, that belongs to "namespace" `std`.
- When a backslash (i.e., an escape character) 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 (i.e., the current screen-position indicator) to move to the beginning of the next line on the screen.
- A message that directs the user to take a specific action is known as a prompt.
- C++ keyword `return` is one of several means to exit a function.
- All variables in a C++ program must be declared before they can be used.
- A variable name in C++ is any valid identifier that is not a keyword. An identifier is a series of characters consisting of letters, digits and underscores (`_`). Identifiers cannot start with a digit. C++ identifiers can be any length; however, some systems and/or C++ implementations may impose some restrictions on the length of identifiers.
- C++ is case sensitive.
- Most calculations are performed in assignment statements.
- A variable is a location in the computer's memory where a value can be stored for use by a program.
- Variables of type `int` hold integer values, i.e., whole numbers such as 7, 11, 0, 31914.
- Every variable stored in the computer's memory has a name, a value, a type and a size.
- Whenever a new value is placed in a memory location, the process is destructive; i.e., the new value replaces the previous value in that location. The previous value is lost.
- When a value is read from memory, the process is nondestructive; i.e., a copy of the value is read,

leaving the original value undisturbed in the memory location.

- The `std::endl` stream manipulator outputs a newline, then "flushes the output buffer."

[Page 67]

- C++ evaluates arithmetic expressions in a precise sequence determined by the rules of operator precedence and associativity.
- Parentheses may be used to force the order of evaluation to occur in any sequence desired by the programmer.
- Integer division (i.e., both the numerator and the denominator are integers) yields an integer quotient. Any fractional part in integer division is truncated no rounding occurs.
- The modulus operator, `%`, yields the remainder after integer division. The modulus operator can be used only with integer operands.
- The `if` statement allows a program to make a decision when a certain condition is met. The format for an `if` statement is

```
if ( condition )
    statement;
```

If the condition is true, the statement in the body of the `if` is executed. If the condition is not met, i.e., the condition is false, the body statement is skipped.

- Conditions in `if` statements are commonly formed by using equality operators and relational operators. The result of using these operators is always the value true or false.
- The declaration

```
using std::cout;
```

is a `using` declaration that eliminates the need to repeat the `std::` prefix. Once we include this `using` declaration, we can write `cout` instead of `std::cout` in the remainder of a program.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 68]

nested parentheses

newline character (\n)

nondestructive read

nonfatal logic error

operand

operator

operator associativity

parentheses ()

perform an action

precedence

preprocessor directive

prompt

redundant parentheses

relational operators

< "is less than"

<= "is less than or equal to"

> "is greater than"

>= "is greater than or equal to"

return statement

rules of operator precedence

self-documenting program

semicolon (;) statement terminator

standard input stream object (cin)

standard output stream object (cout)

statement

statement terminator (;

stream

stream insertion operator (<<)

stream extraction operator (>>)

stream manipulator

string

string literal

syntax error

using declaration

variable

white space

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 69]

- If the variable `number` is not equal to 7, print "The variable number is not equal to 7".
- Print the message "This is a C++ program" on one line.
- Print the message "This is a C++ program" on two lines. End the first line with C++.
- Print the message "This is a C++ program" with each word on a separate line.
- Print the message "This is a C++ program" with each word separated from the next by a tab.

2.4

Write a statement (or comment) to accomplish each of the following (assume that using declarations have been used):

a.

State that a program calculates the product of three integers.

b.

Declare the variables `x`, `y`, `z` and `result` to be of type `int` (in separate statements).

c.

Prompt the user to enter three integers.

Read three integers from the keyboard and store them in the variables `x`, `y` and `z`.

e.

Compute the product of the three integers contained in variables `x`, `y` and `z`, and assign the result to the variable `result`.

f.

Print "The product is " followed by the value of the variable `result`.

g.

Return a value from `main` indicating that the program terminated successfully.

2.5

Using the statements you wrote in [Exercise 2.4](#), write a complete program that calculates and displays the product of three integers. Add comments to the code where appropriate. [Note: You will need to write the necessary `using` declarations.]

2.6

Identify and correct the errors in each of the following statements (assume that the statement `using std::cout;` is used):

a.

```
if ( c < 7 );
    cout << "c is less than 7\n";
```

b.

```
if ( c => 7 )
    cout << "c is equal to or greater than 7\n";
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 70]

2.4

a.

```
// Calculate the product of three integers
```

b.

```
int x;  
int y;  
int z;  
int result;
```

c.

```
cout << "Enter three integers: ";
```

d.

```
cin >> x >> y >> z;
```

e.

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

f.

```
cout << "The product is " << result << endl;
```

g.

```
return 0;
```

2.5

(See program below)

```
1 // Calculate the product of three integers
```

```

2 #include <iostream> // allows program to perform input and output
3
4 using std::cout; // program uses cout
5 using std::cin; // program uses cin
6 using std::endl; // program uses endl
7
8 // function main begins program execution
9 int main()
10 {
11     int x; // first integer to multiply
12     int y; // second integer to multiply
13     int z; // third integer to multiply
14     int result; // the product of the three integers
15
16     cout << "Enter three integers: "; // prompt user for data
17     cin >> x >> y >> z; // read three integers from user
18     result = x * y * z; // multiply the three integers; store result
19     cout << "The product is " << result << endl; // print result; end line
20
21     return 0; // indicate program executed successfully
22 } // end function main

```

2.6

a.

Error: Semicolon after the right parenthesis of the condition in the `if` statement.

Correction: Remove the semicolon after the right parenthesis. [Note: The result of this error is that the output statement will be executed whether or not the condition in the `if` statement is true.] The semicolon after the right parenthesis is a null (or empty) statementa statement that does nothing. We will learn more about the null statement in the next chapter.

b.

Error: The relational operator `=>`.

Correction: Change `=>` to `>=`, and you may want to change "equal to or greater than" to "greater than or equal to" as well.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 71]

•

The object used to print information on the screen is _____.

•

A C++ statement that makes a decision is _____.

•

Most calculations are normally performed by _____ statements.

•

The _____ object inputs values from the keyboard.

2.9

Write a single C++ statement or line that accomplishes each of the following:

a.

Print the message "Enter two numbers".

b.

Assign the product of variables b and c to variable a.

c.

State that a program performs a payroll calculation (i.e., use text that helps to document a program).

d.

Input three integer values from the keyboard into integer variables a, b and c.

2.10

State which of the following are true and which are false. If false, explain your answers.

a.

C++ operators are evaluated from left to right.

b.

The following are all valid variable names: _under_bar_, m928134, t5, j7, her_sales, his_account_total, a, b, c, z, z2.

c.

The statement `cout << "a = 5;" ;` is a typical example of an assignment statement.

d.

A valid C++ arithmetic expression with no parentheses is evaluated from left to right.

e.

The following are all invalid variable names: 3g, 87, 67h2, h22, 2h.

2.11

Fill in the blanks in each of the following:

a.

What arithmetic operations are on the same level of precedence as multiplication? _____.

b.

When parentheses are nested, which set of parentheses is evaluated first in an arithmetic expression? _____.

c.

A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a _____.

2.12

What, if anything, prints when each of the following C++ statements is performed? If nothing prints, then answer "nothing." Assume $x = 2$ and $y = 3$.

a.

```
cout << x;
```

b.

```
cout << x + x;
```

c.

```
cout << "x=";
```

d.

```
cout << "x = " << x;
```

e.

```
cout << x + y << " = " << y + x;
```

f.

```
z = x + y;
```

g.

```
cin >> x >> y;
```

h.

```
// cout << "x + y = " << x + y;
```

i.

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

2.13

Which of the following C++ statements contain variables whose values are replaced?

a.

```
cin >> b >> c >> d >> e >> f;
```

b.

```
p = i + j + k + 7;
```

c.

```
cout << "variables whose values are replaced";
```

d.

```
cout << "a = 5";
```

2.14

Given the algebraic equation $y = ax^3 + 7$, which of the following, if any, are correct C++ statements for this equation?

a.

```
y = a * x * x * x + 7;
```

b.

```
y = a * x * x * ( x + 7 );
```

c.

```
y = ( a * x ) * x * ( x + 7 );
```

d.

```
y = (a * x) * x * x + 7;
```

e.

```
y = a * ( x * x * x ) + 7;
```

f.

```
y = a * x * ( x * x + 7 );
```

[Page 72]

2.15

State the order of evaluation of the operators in each of the following C++ statements and show the value of *x* after each statement is performed.

a.

```
x = 7 + 3 * 6 / 2 - 1;
```

b.

```
x = 2 % 2 + 2 * 2 - 2 / 2;
```

c.

```
x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );
```

2.16

Write a program that asks the user to enter two numbers, obtains the two numbers from the user and prints the sum, product, difference, and quotient of the two numbers.

2.17

Write a program that prints the numbers 1 to 4 on the same line with each pair of adjacent numbers separated by one space. Do this several ways:

a.

Using one statement with one stream insertion operator.

b.

Using one statement with four stream insertion operators.

c.

Using four statements.

2.18

Write a program that asks the user to enter two integers, obtains the numbers from the user, then prints the larger number followed by the words "is larger." If the numbers are equal, print the message "These numbers are equal."

2.19

Write a program that inputs three integers from the keyboard and prints the sum, average, product, smallest and largest of these numbers. The screen dialog should appear as follows:

```
Input three different integers: 13 27 14
Sum is 54
Average is 18
Product is 4914
Smallest is 13
Largest is 27
```

2.20

Write a program that reads in the radius of a circle as an integer and prints the circle's diameter, circumference and area. Use the constant value 3.14159 for π . Do all calculations in output statements. [Note: In this chapter, we have discussed only integer constants and variables. In [Chapter 4](#) we discuss floating-point numbers, i.e., values that can have decimal points.]

2.21

Write a program that prints a box, an oval, an arrow and a diamond as follows:

2.22

What does the following code print?

```
cout << "*\n**\n***\n****\n*****" << endl;
```

2.23

Write a program that reads in five integers and determines and prints the largest and the smallest integers in the group. Use only the programming techniques you learned in this chapter.

2.24

Write a program that reads an integer and determines and prints whether it is odd or even. [Hint: Use the modulus operator. An even number is a multiple of two. Any multiple of two leaves a remainder of zero when divided by 2.]

[Page 73]

2.25

Write a program that reads in two integers and determines and prints if the first is a multiple of the second. [Hint: Use the modulus operator.]

2.26

Display the following checkerboard pattern with eight output statements, then display the same pattern using as few statements as possible.

```
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
```

2.27

Here is a peek ahead. In this chapter you learned about integers and the type `int`. C++ can also represent uppercase letters, lowercase letters and a considerable variety of special symbols. C++ uses small integers internally to represent each different character. The set of characters a computer uses and the corresponding integer representations for those characters is called that computer's [character set](#). You can print a character by enclosing that character in single quotes, as with

```
cout << 'A'; // print an uppercase A
```

You can print the integer equivalent of a character using `static_cast` as follows:

```
cout << static_cast< int >( 'A' ); // print 'A' as an integer
```

This is called a [cast](#) operation (we formally introduce casts in [Chapter 4](#)). When the preceding statement executes, it prints the value 65 (on systems that use the [ASCII character set](#)). Write a program that prints the integer equivalent of a character typed at the keyboard. Test your program several times using uppercase letters, lowercase letters, digits and special characters (like \$).

2.28

Write a program that inputs a five-digit integer, separates the integer into its individual digits and prints the digits separated from one another by three spaces each. [Hint: Use the integer division and modulus operators.] For example, if the user types in 42339, the program should print:

4	2	3	3	9
---	---	---	---	---

2.29

Using only the techniques you learned in this chapter, write a program that calculates the squares and cubes of the integers from 0 to 10 and uses tabs to print the following neatly formatted table of values:

integer	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

 [PREV](#)[NEXT](#) **page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 75]

Outline

[3.1](#) Introduction

[3.2](#) Classes, Objects, Member Functions and Data Members

[3.3](#) Overview of the Chapter Examples

[3.4](#) Defining a Class with a Member Function

[3.5](#) Defining a Member Function with a Parameter

[3.6](#) Data Members, set Functions and get Functions

[3.7](#) Initializing Objects with Constructors

[3.8](#) Placing a Class in a Separate File for Reusability

[3.9](#) Separating Interface from Implementation

[3.10](#) Validating Data with set Functions

[3.11](#) (Optional) Software Engineering Case Study: Identifying the Classes in the ATM Requirements Document

[3.12](#) Wrap-Up

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 75 (continued)]

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 will begin writing programs that employ the basic concepts of object-oriented programming that we introduced in [Section 1.17](#). 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.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 76]

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.

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 1.17](#) 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.

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.

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**.

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.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 77]

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 then 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 createdthe 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.

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 112. 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.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

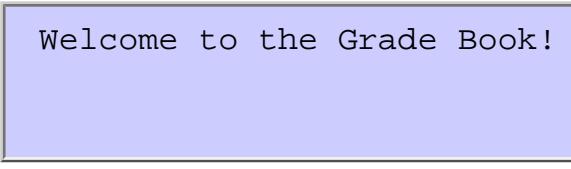
[Page 78]

Figure 3.1. Defining class GradeBook with a member function, creating a GradeBook object and calling its member 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 2025) can create an object of class `GradeBook`, we must tell the compiler what member functions and data members belong to the class. This is known as **defining a class**. The `GradeBook` **class definition** (lines 917) contains a member function called `displayMessage` (lines 1316) 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 2223 in detail.

The class definition begins at 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.

[Page 79]

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 the function `main` is always called automatically when you execute a program. Most functions do not get called automatically. As you will 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 called an **access specifier**. Lines 1316 define member function `displayMessage`. This member function appears after access specifier `public`: to indicate that the function is "available to the public" that is, it can be called by other functions in the program 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 will omit the colon as we did in this sentence. [Section 3.6](#) introduces a second access specifier, `private` (again, we omit the colon in our discussions, but include it in our programs).

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 perform a task but will not return (i.e., give back) 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 will 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 will 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.

[Page 80]

Testing Class GradeBook

Next, we'd like to use class `GradeBook` in a program. As you learned in Chapter 2, function `main` begins the execution of every program. Lines 2025 of Fig. 3.1 contain the `main` function that will control our program's execution.

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 will learn 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 917. When we declare variables of type `int`, as we did in [Chapter 2](#), the compiler knows what `int` is—it's a fundamental type. When we write line 22, however, the compiler does not automatically know what type `GradeBook` is—it's a [user-defined type](#). Thus, we must tell the compiler what `GradeBook` is by including the class definition, as we did in lines 917. If we omitted these lines, the compiler would issue an error message (such as "'`GradeBook`' : undeclared identifier" in Microsoft Visual C++ .NET or "'`GradeBook`' : undeclared" in GNU C++). Each new class you create becomes a new type that can be used to create objects. Programmers 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 1316) 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 at line 24, which indicates that `main` performed its tasks successfully. This is the end of `main`, so the program terminates.

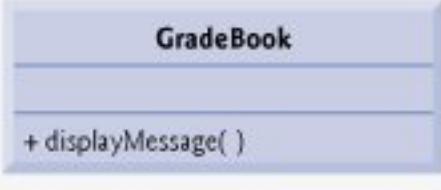
UML Class Diagram for Class `GradeBook`

Recall from [Section 1.17](#) that the UML is a graphical language used by programmers to represent their object-oriented systems in a standardized manner. In the UML, each class is modeled in a class diagram as a rectangle with three compartments. [Figure 3.2](#) presents a [UML class diagram](#) for class `GradeBook` of [Fig. 3.1](#). The top compartment contains the name of the class, centered horizontally and in boldface type. The middle compartment contains the class's attributes, which correspond to data members in C++. In [Fig. 3.2](#) the middle compartment is empty, because the version of class `GradeBook` in [Fig. 3.1](#) does not have any attributes. ([Section 3.6](#) presents a version of the `GradeBook` class that does have 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. The 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++). We frequently use UML class diagrams to summarize class attributes and operations.

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

(This item is displayed on page 80 in the print version)

[View full size image]



GradeBook

+ displayMessage()

 PREV
page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 82]

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.

[Page 83]

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 should 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 2021 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 learn 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. 6.46.5](#)). 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 match the types of the corresponding parameters in the function header. (As you will learn in subsequent chapters, an argument's type and its corresponding parameter's type need not always be identical, but they must be "consistent.") In our example, the one `string` argument in the function call (i.e., `nameOfCourse`) exactly matches the one `string` parameter in the member-function definition (i.e., `courseName`).

Common Programming Error 3.4



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

[Page 84]

Common Programming Error 3.5



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

Good Programming Practice 3.1



To avoid ambiguity, do not use the same names for the arguments passed to a function and the corresponding parameters in the function definition.

Good Programming Practice 3.2



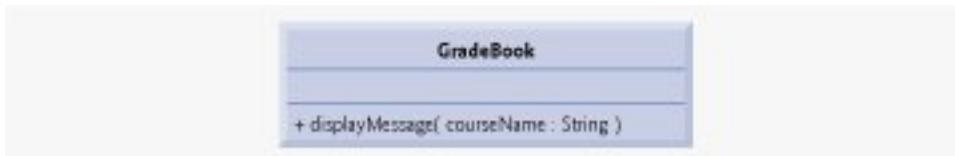
Choosing meaningful function names and meaningful parameter names makes programs more readable and helps avoid excessive use of comments.

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 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 1822) 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.

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

[View full size image]



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 85]

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 membermember function `getCourseName` obtains a `GradeBook`'s course name from that data member. Member function `displayMessage`which now specifies no parametersstill displays a welcome message that includes the course name. However, as you will see, the function now obtains the course name by calling another function in the same class`getCourseName`.

Figure 3.5. Defining and testing class `GradeBook` with a data member and set and get functions.

(This item is displayed on pages 85 - 86 in the print version)

```

1 // Fig. 3.5: fig03_05.cpp
2 // Define class GradeBook that contains a courseName data member
3 // and member functions to set and get its value;
4 // Create and manipulate a GradeBook object with these functions.
5 #include <iostream>
6 using std::cout;
7 using std::cin;
8 using std::endl;
9
10 #include <string> // program uses C++ standard string class
11 using std::string;
12 using std::getline;
13
14 // GradeBook class definition
15 class GradeBook
16 {
17 public:
18     // function that sets the course name

```

```
19 void setCourseName( string name )
20 {
21     courseName = name; // store the course name in the object
22 } // end function setCourseName
23
24 // function that gets the course name
25 string getCourseName()
26 {
27     return courseName; // return the object's courseName
28 } // end function getCourseName
29
30 // function that displays a welcome message
31 void displayMessage()
32 {
33     // this statement calls getCourseName to get the
34     // name of the course this GradeBook represents
35     cout << "Welcome to the grade book for\n" << getCourseName() << "!"
36     << endl;
37 } // end function displayMessage
38 private:
39     string courseName; // course name for this GradeBook
40 }; // end class GradeBook
41
42 // function main begins program execution
43 int main()
44 {
45     string nameOfCourse; // string of characters to store the course name
46     GradeBook myGradeBook; // create a GradeBook object named myGradeBook
47
48     // display initial value of courseName
49     cout << "Initial course name is: " << myGradeBook.getCourseName()
50     << endl;
51
52     // prompt for, input and set course name
53     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!

[Page 86]

Good Programming Practice 3.3



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

A typical instructor teaches more than one course, 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 1540) but outside the bodies of the class's member-function definitions (lines 1922, 2528 and 3137), line 39 is a declaration for a data member. Every instance (i.e., object) of class `GradeBook` contains one copy of each of the class's data members. For example, if there are two `GradeBook` objects, each object has its own copy of `courseName` (one per object), as we'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, `GradeBook`) can manipulate any data members that appear in the class definition (in this case, `courseName`).

[Page 87]

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 will 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 will see in [Chapter 10](#)), 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 3.4



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 3.5



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) an object of class `GradeBook`, 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).

[Page 88]

Software Engineering Observation 3.2



We will learn in [Chapter 10](#), Classes: A Deeper Look, Part 2, 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 1922) 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 will 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 2528) 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

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

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

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

returns 114 from function `maximum` and initializes variable `biggest` with 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 at 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`.

[Page 89]

Member Function `displayMessage`

Member function `displayMessage` (lines 3137) 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 3536 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 4360) creates one object of class `GradeBook` and uses each of its member functions. Line 46 creates a `GradeBook` object named `myGradeBook`. Lines 4950 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 1922). 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 will see in [Chapter 10](#), Classes: A Deeper Look, Part 2). 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 4360) 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 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 3.6



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



It is important to write programs that are understandable and easy to maintain. Change is the rule rather than the exception. Programmers should anticipate that their code will be modified.

Software Engineering Observation 3.4



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.

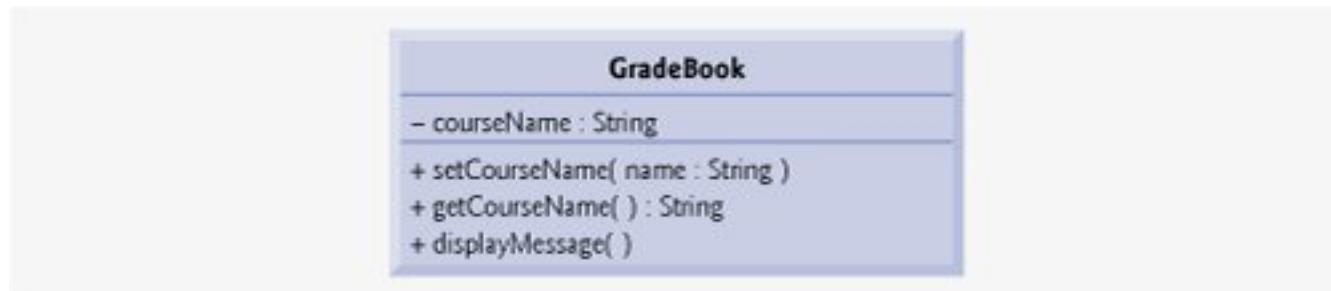
[Page 91]

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 class GradeBook's data member `courseName` as an attribute in the middle compartment of the class. 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.

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

[View full size image]



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 92]

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 1720) 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.

Figure 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.

(This item is displayed on pages 92 - 93 in the print version)

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

```

30     {
31         return courseName; // return object's courseName
32     } // end function getCourseName
33
34     // display a welcome message to the GradeBook user
35     void displayMessage()
36     {
37         // call getCourseName to get the courseName
38         cout << "Welcome to the grade book for\n" << getCourseName( )
39             << "!" << endl;
40     } // end function displayMessage
41 private:
42     string courseName; // course name for this GradeBook
43 }; // end class GradeBook
44
45 // function main begins program execution
46 int main()
47 {
48     // create two GradeBook objects
49     GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
50     GradeBook gradeBook2( "CS102 Data Structures in C++" );
51
52     // display initial value of courseName for each GradeBook
53     cout << "gradeBook1 created for course: " << gradeBook1.getCourseName( )
54         << "\ngradeBook2 created for course: " << gradeBook2.getCourseName( )
55         << endl;
56     return 0; // indicate successful termination
57 } // end main

```

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

[Page 93]

Defining a Constructor

Lines 1720 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 4950). 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 2326) 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.

[Page 94]

Testing Class `GradeBook`

Lines 4657 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 1720) 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 5354 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.

The programmer explicitly defines a constructor that takes no arguments. Such a default constructor

will perform the initialization specified by the programmer and will call the default constructor for each data member that is an object of another class.

If the programmer defines 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 3.5



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.

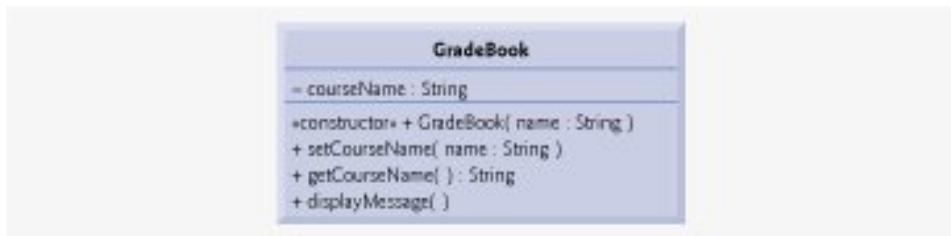
Adding the Constructor to Class `GradeBook`'s UML Class Diagram

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.

[Page 95]

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

[\[View full size image\]](#)



PREV

NEXT

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 96]

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 1141) and lines 38, 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) at lines 1021. To help you prepare for the larger programs you will 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 will soon learn how a source-code file with `main` can use the class definition found in a header file to create objects of a class.

Figure 3.9. GradeBook class definition.

(This item is displayed on pages 96 - 97 in the print version)

```

1 // Fig. 3.9: GradeBook.h
2 // GradeBook class definition in a separate file from main.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string> // class GradeBook uses C++ standard string class
8 using std::string;
9
10 // GradeBook class definition
11 class GradeBook
12 {
13 public:
14     // constructor initializes courseName with string supplied as argument
15     GradeBook( string name )
16     {
17         setCourseName( name ); // call set function to initialize courseName
18     } // end GradeBook constructor
19
20     // function to set the course name
21     void setCourseName( string name )
22     {
23         courseName = name; // store the course name in the object
24     } // end function setCourseName
25
26     // function to get the course name
27     string getCourseName()

```

```

28     {
29         return courseName; // return object's courseName
30     } // end function getCourseName
31
32     // display a welcome message to the GradeBook user
33     void displayMessage( )
34     {
35         // call getCourseName to get the courseName
36         cout << "Welcome to the grade book for\n" << getCourseName( )
37             << "!" << endl;
38     } // end function displayMessage
39 private:
40     string courseName; // course name for this GradeBook
41 }; // end class GradeBook

```

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++ .NET will produce the linker error message:

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

[Page 97]

Running GNU C++ on Linux 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.

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

```

1 // Fig. 3.10: fig03_10.cpp
2 // Including class GradeBook from file GradeBook.h for use in main.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "GradeBook.h" // include definition of class GradeBook
8
9 // function main begins program execution
10 int main()
11 {
12     // create two GradeBook objects
13     GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
14     GradeBook gradeBook2( "CS102 Data Structures in C++" );
15
16     // display initial value of courseName for each GradeBook
17     cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
18         << endl;
19         << endl;
20     return 0; // indicate successful termination
21 } // end main

```

```

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

```

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.

[Page 98]

To create `GradeBook` objects `gradeBook1` and `gradeBook2` in lines 1314 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 typically 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 1314 and 1718 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

3.

the clients do not know how the class's member functions are implemented.

 PREV

page footer

NEXT 

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 100]

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 918). 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 1215) 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. 3.12) 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.

[Page 101]

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

(This item is displayed on page 100 in the print version)

```

1 // Fig. 3.11: GradeBook.h
2 // GradeBook class definition. This file presents GradeBook's public
3 // interface without revealing the implementations of GradeBook's member
4 // functions, which are defined in GradeBook.cpp.
5 #include <string> // class GradeBook uses C++ standard string class
6 using std::string;
7
8 // GradeBook class definition
9 class GradeBook
10 {
11 public:
12     GradeBook( string ); // constructor that initializes courseName
13     void setCourseName( string ); // function that sets the course name
14     string getCourseName(); // function that gets the course name
15     void displayMessage(); // function that displays a welcome message
16 private:
17     string courseName; // course name for this GradeBook
18 } // end class GradeBook

```

Common Programming Error 3.8



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

Good Programming Practice 3.7



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

Source-code file GradeBook.cpp (Fig. 3.12) defines class GradeBook's member functions, which were declared in lines 1215 of Fig. 3.11. The member-function definitions appear in lines 1134 and are nearly identical to the member-function definitions in lines 1538 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, 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 GradeBookthe 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 functionthe 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

[Page 102]

1.

the first line of each member function (lines 11, 17, 23 and 29) matches its prototype in the GradeBook.h filefor example, the compiler ensures that `getCourseName` accepts no parameters and returns a `string`.

2.

each member function knows about the class's data members and other member functionsfor 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).

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

```

1 // Fig. 3.12: GradeBook.cpp
2 // GradeBook member-function definitions. This file contains
3 // implementations of the member functions prototyped in GradeBook.h.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "GradeBook.h" // include definition of class GradeBook
9
10 // constructor initializes courseName with string supplied as argument
11 GradeBook::GradeBook( string name )
12 {
13     setCourseName( name ); // call set function to initialize courseName
14 } // end GradeBook constructor
15
16 // function to set the course name
17 void GradeBook::setCourseName( string name )
18 {
19     courseName = name; // store the course name in the object
20 } // end function setCourseName
21
22 // function to get the course name
23 string GradeBook::getCourseName( )
24 {
25     return courseName; // return object's courseName
26 } // end function getCourseName
27
28 // display a welcome message to the GradeBook user
29 void GradeBook::displayMessage( )
30 {
31     // call getCourseName to get the courseName
32     cout << "Welcome to the grade book for\n" << getCourseName( )
33         << "!" << endl;
34 } // end function displayMessage

```

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.

Figure 3.13. GradeBook class demonstration after separating its interface from its implementation.

```
1 // Fig. 3.13: fig03_13.cpp
2 // GradeBook class demonstration after separating
3 // its interface from its implementation.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "GradeBook.h" // include definition of class GradeBook
9
10 // function main begins program execution
11 int main()
12 {
13     // create two GradeBook objects
14     GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
15     GradeBook gradeBook2( "CS102 Data Structures in C++" );
16
17     // display initial value of courseName for each GradeBook
18     cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
19         << endl
20         << "gradeBook2 created for course: " << gradeBook2.getCourseName()
21         << endl;
22     return 0; // indicate successful termination
23 } // end main
```

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

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.

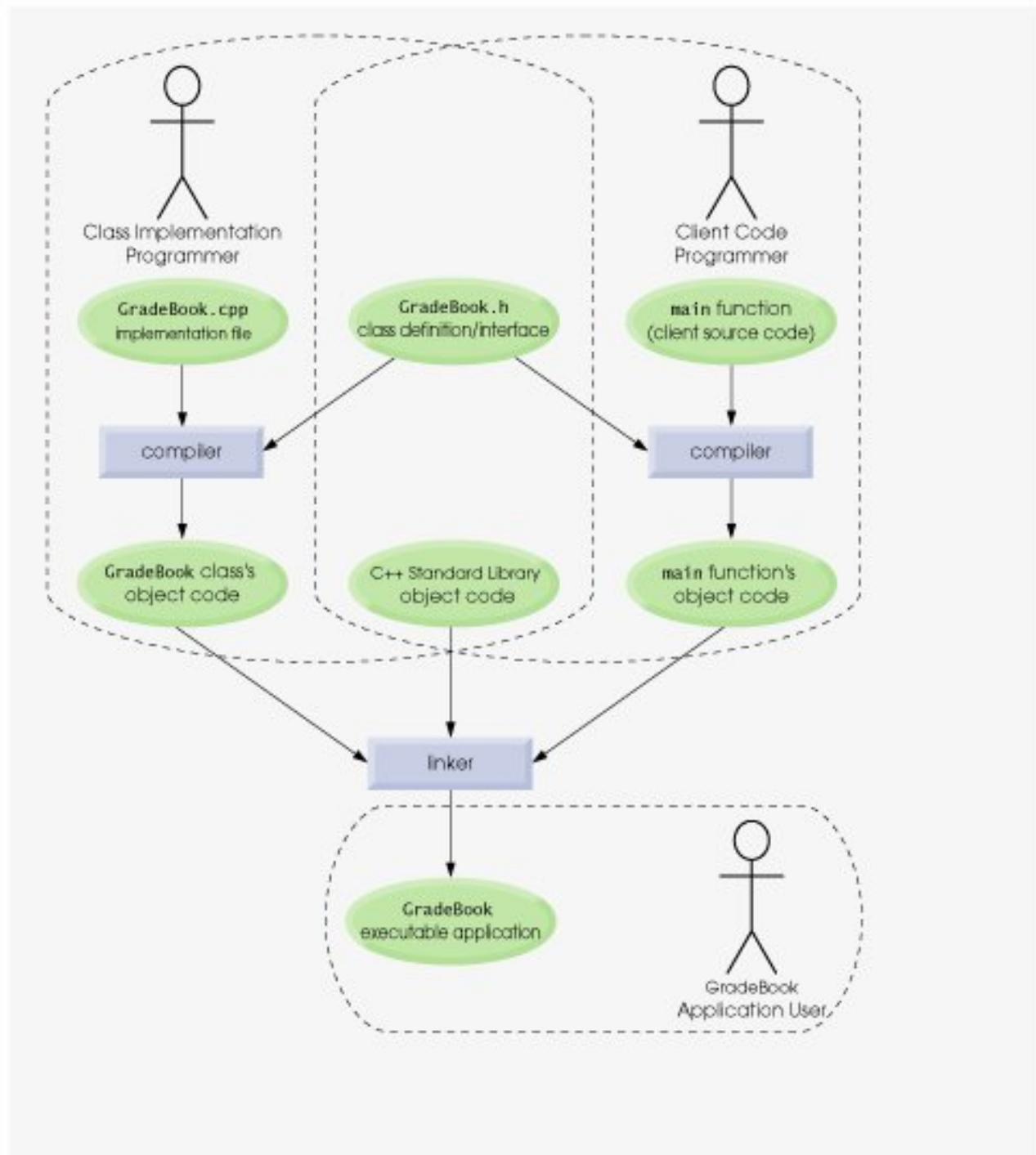
The Compilation and Linking Process

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 class's client code. 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.]

Figure 3.14. Compilation and linking process that produces an executable application.

(This item is displayed on page 104 in the print version)

[View full size image]



A class-implementation programmer responsible for creating a reusable GradeBook class creates the header file `GradeBook.h` and source-code file `GradeBook.cpp` that `#includes` the header file, then compiles the source-code file to create `GradeBook`'s object code. To hide the implementation details of `GradeBook`'s member functions, 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`'s source-code file, so the client remains unaware of how `GradeBook`'s member functions are implemented.

[Page 104]

[Page 105]

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 or study the Dive-Into™ publications that we provide for various C++ compilers at www.deitel.com/books/cpphtp5.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 107]

Figure 3.15. GradeBook class definition.

(This item is displayed on page 106 in the print version)

```
1 // Fig. 3.15: GradeBook.h
2 // GradeBook class definition presents the public interface of
3 // the class. Member-function definitions appear in GradeBook.cpp.
4 #include <string> // program uses C++ standard string class
5 using std::string;
6
7 // GradeBook class definition
8 class GradeBook
9 {
10 public:
11     GradeBook( string ); // constructor that initializes a GradeBook object
12     void setCourseName( string ); // function that sets the course name
13     string getCourseName(); // function that gets the course name
14     void displayMessage(); // function that displays a welcome message
15 private:
16     string courseName; // course name for this GradeBook
17 };
```

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

(This item is displayed on pages 106 - 107 in the print version)

```

1 // Fig. 3.16: GradeBook.cpp
2 // Implementations of the GradeBook member-function definitions.
3 // The setCourseName function performs validation.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "GradeBook.h" // include definition of class GradeBook
9
10 // constructor initializes courseName with string supplied as argument
11 GradeBook::GradeBook( string name )
12 {
13     setCourseName( name ); // validate and store courseName
14 } // end GradeBook constructor
15
16 // function that sets the course name;
17 // ensures that the course name has at most 25 characters
18 void GradeBook::setCourseName( string name )
19 {
20     if ( name.length() <= 25 ) // if name has 25 or fewer characters
21         courseName = name; // store the course name in the object
22
23     if ( name.length() > 25 ) // if name has more than 25 characters
24     {
25         // set courseName to first 25 characters of parameter name
26         courseName = name.substr( 0, 25 ); // start at 0, length of 25
27
28         cout << "Name \" " << name << "\" exceeds maximum length (25).\n"
29             << "Limiting courseName to first 25 characters.\n" << endl;
30     } // end if
31 } // end function setCourseName
32
33 // function to get the course name
34 string GradeBook::getCourseName( )
35 {
36     return courseName; // return object's courseName
37 } // end function getCourseName
38
39 // display a welcome message to the GradeBook user
40 void GradeBook::displayMessage( )
41 {
42     // call getCourseName to get the courseName
43     cout << "Welcome to the grade book for\n" << getCourseName( )
44         << "!" << endl;
45 } // end function displayMessage

```

The `if` statement in lines 2330 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 2829 display a warning message.

Note that the `if` statement in lines 2330 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 will learn more about placing multiple statements in the body of a control statement in [Chapter 4](#).

[Page 108]

Note that the `cout` statement in lines 2829 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 "`\n` 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 2829 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. 3.153.16](#)) featuring validation. Line 14 creates a `GradeBook` object named `gradeBook1`. Recall that the `GradeBook` constructor calls

member function `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 red in line 14). Notice that the output in Fig. 3.17 contains the warning message output by lines 2829 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.

[Page 109]

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

(This item is displayed on pages 108 - 109 in the print version)

```

1 // Fig. 3.17: fig03_17.cpp
2 // Create and manipulate a GradeBook object; illustrate validation.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "GradeBook.h" // include definition of class GradeBook
8
9 // function main begins program execution
10 int main()
11 {
12     // create two GradeBook objects;
13     // initial course name of gradeBook1 is too long
14     GradeBook gradeBook1( "CS101 Introduction to Programming in C++" );
15     GradeBook gradeBook2( "CS102 C++ Data Structures" );
16
17     // display each GradeBook's courseName
18     cout << "gradeBook1's initial course name is: "
19         << gradeBook1.getCourseName()
20         << "\ngradeBook2's initial course name is: "
21         << gradeBook2.getCourseName() << endl;
22
23     // modify myGradeBook's courseName (with a valid-length string)
24     gradeBook1.setCourseName( "CS101 C++ Programming" );
25
26     // display each GradeBook's courseName

```

```

27     cout << "\ngradeBook1's course name is: "
28     << gradeBook1.getCourseName( )
29     << "\ngradeBook2's course name is: "
30     << gradeBook2.getCourseName() << endl;
31     return 0; // indicate successful termination
32 } // 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 1821 of Fig. 3.17 display the truncated course name for gradeBook1 (we highlight this in red 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 2730 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, etc.

Software Engineering Observation 3.6



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—the programmer must provide appropriate validity checking and report the errors.

[Page 110]

Software Engineering Observation 3.7



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. 3.153.17 simple at this early point in the book, `setCourseName` in Fig. 3.16 just prints an appropriate message on the screen.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 111]

Figure 3.18. Nouns and noun phrases in the requirements document.

Nouns and noun phrases in the requirements document		
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	

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 document 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

[Page 112]

- withdrawal
- deposit

The elements of this list are likely to be classes we will 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 as UML conventions as we will 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 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 4.13](#) and [Section 5.11](#).) 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.

Figure 3.19. Representing a class in the UML using a class diagram.

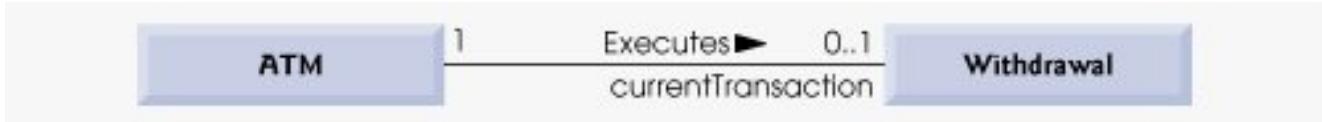


Class diagrams also show the relationships between the classes of the system. [Figure 3.20](#) shows how our classes `ATM` and `withdrawal` relate to one another. For the moment, we choose to model only this subset of classes for simplicity. We present a more complete class diagram later in this section. 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 will place information in these compartments in [Section 4.13](#) and [Section 6.22](#)

Figure 3.20. Class diagram showing an association among classes.

[View full size image]



In [Fig. 3.20](#), the solid line that connects the two classes represents an [association](#) 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.

[Page 113]

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 arrow-head 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 document tells us that the ATM is composed of a screen, a keypad, a cash dispenser and a deposit slot.

Figure 3.21. Multiplicity types.

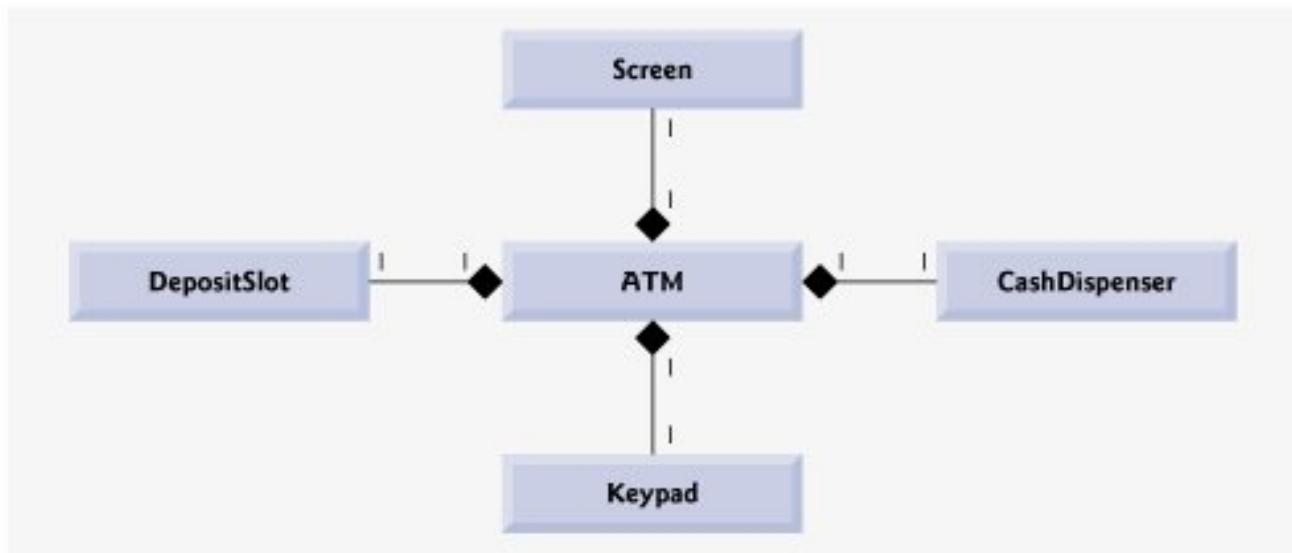
Symbol	Meaning
0	None
1	One
m	An integer value
0..1	Zero or one
m, n	m or n
m..n	At least m, but not more than n
*	Any nonnegative integer (zero or more)
0..*	Zero or more (identical to *)
1..*	One or more

[Page 114]

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 will see in the "Software Engineering Case Study" section in Chapter 13 that the "is-a" relationship defines inheritance.)

Figure 3.22. Class diagram showing composition relationships.

[\[View full size image\]](#)



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 document. [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.]

Figure 3.23. Class diagram for the ATM system model.

[View full size image]

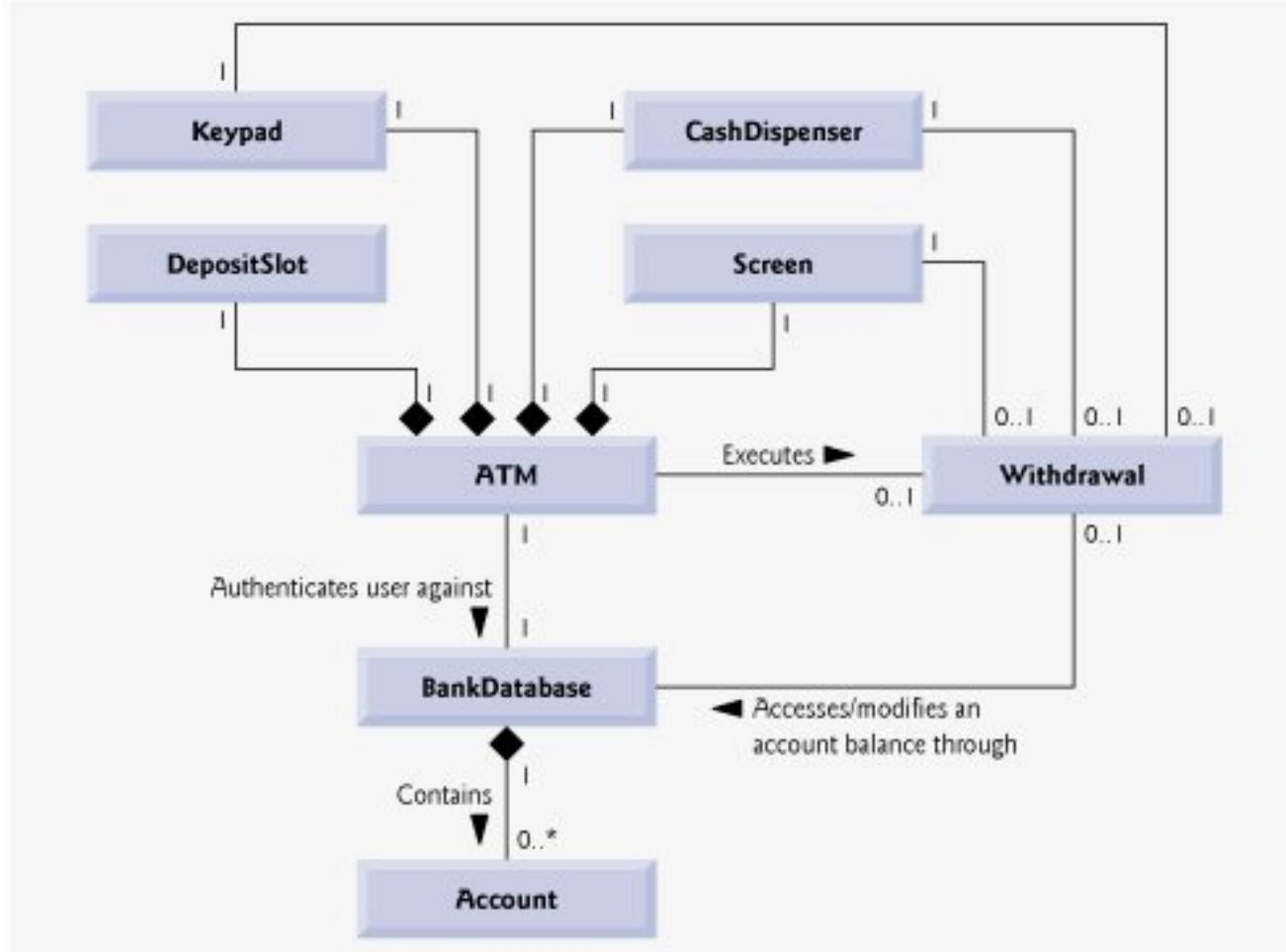


Figure 3.23 presents a graphical model of the structure of the ATM system. This class diagram includes classes `BankDatabase` and `Account`, and several associations that were not present in either [Fig. 3.20](#) or [Fig. 3.22](#). The class diagram shows that class `ATM` has a [one-to-one relationship](#) 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 accountsone 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` stores many `Accounts`. Similarly, class `Account` has a [many-to-one relationship](#) with class

BankDatabase there can be many Accounts stored 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.]

[Page 116]

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 document, 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 4.13, we determine the attributes for each of these classes, and in Section 5.11, 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 stand-alone, 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`.

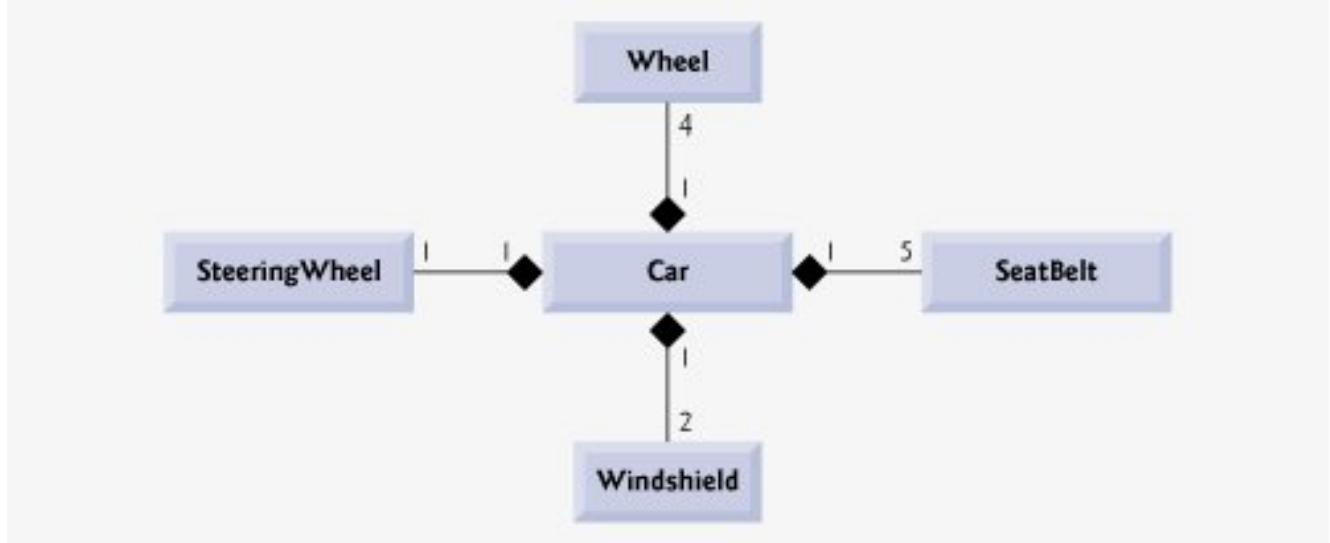
Answers to Software Engineering Case Study Self-Review Exercises

- 3.1** [Note: Student answers may vary.] [Figure 3.24](#) presents a class diagram that shows some of the composition relationships of a class **Car**.

Figure 3.24. Class diagram showing composition relationships of a class Car.

(This item is displayed on page 117 in the print version)

[View full size image]



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

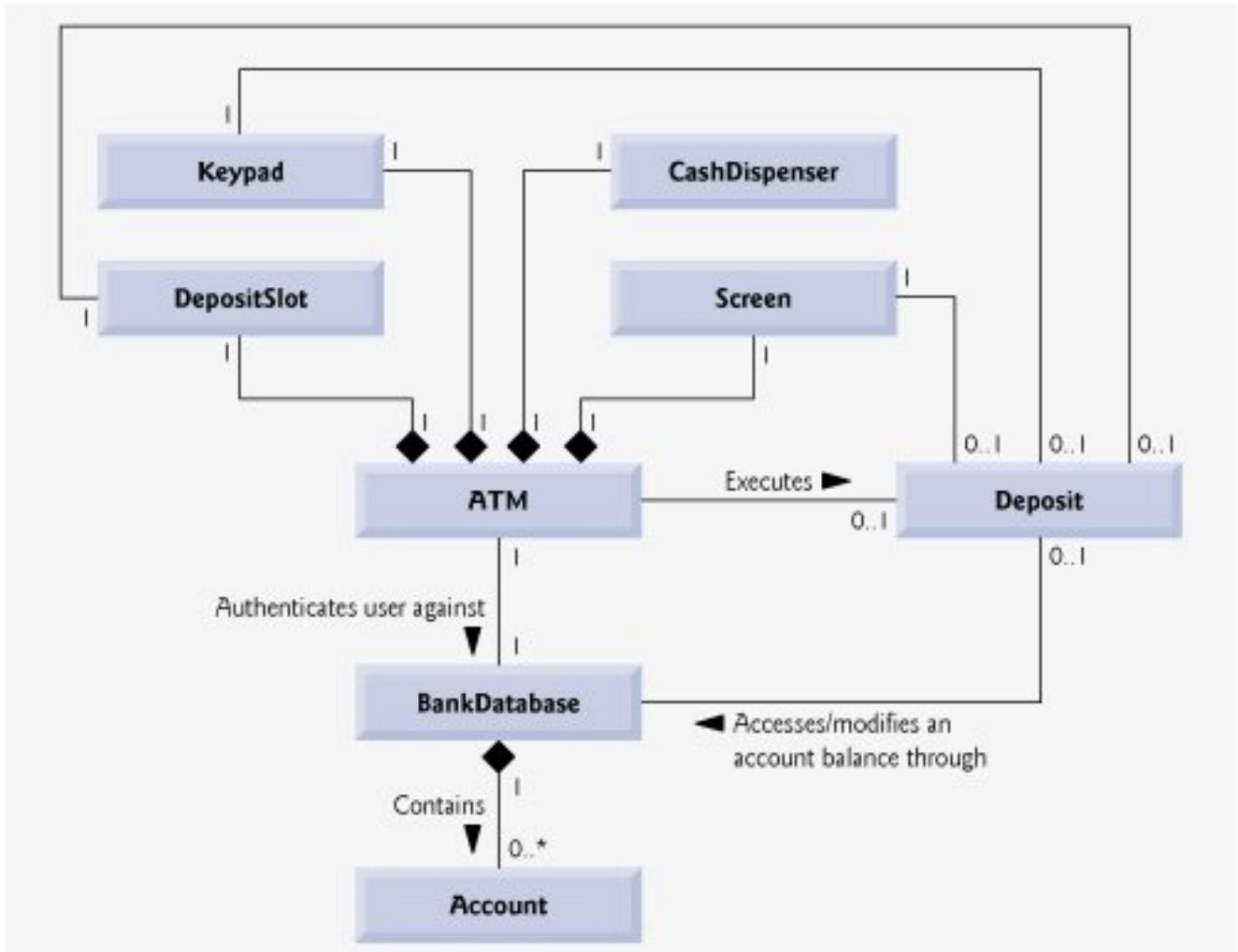
[Page 117]

- 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**.

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

[View full size image]



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 118]

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. In particular, 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 it 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 the 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 also 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.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 119]

- Each new class you create becomes a new type in C++ that can be used to declare variables and create objects. This is one reason why C++ is known as an extensible language.
- A member function can require one or more parameters that represent additional data it needs to perform its task. A function call supplies arguments for each of the function's parameters.
- A member function is called by following the object name with a dot operator (.), the function name and a set of parentheses containing the function's arguments.
- A variable of C++ Standard Library class `string` represents a string of characters. This class is defined in header file `<string>`, and the name `string` belongs to namespace `std`.
- Function `getline` (from header `<string>`) reads characters from its first argument until a newline character is encountered, then places the characters (not including the newline) in the `string` variable specified as its second argument. The newline character is discarded.
- A parameter list may contain any number of parameters, including none at all (represented by empty parentheses) to indicate that a function does not require any parameters.
- The number of arguments in a function call must match the number 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.
- Variables declared in a function's body are local variables and can be used only from the point of their declaration in the function to the immediately following closing right brace (`}`). When a function terminates, the values of its local variables are lost.
- 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.
- Data members normally are `private`. Variables or functions declared `private` are accessible only to member functions of the class in which they are declared.
- When a program creates (instantiates) an object of a class, its `private` data members are encapsulated (hidden) in the object and can be accessed only by member functions of the object's class.
- 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.
- By default, the initial value of a `string` is the empty string i.e., a string that does not contain any characters. Nothing appears on the screen when an empty string is displayed.
- Classes often provide `public` member functions to allow clients of the class to set or get `private` data members. The names of these member functions normally begin with `set` or `get`.
- Providing `public` `set` and `get` functions allows clients of a class to indirectly access the hidden data. 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.
- 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. If the class's data representation is changed, member functions that access the data only via the `set` and `get` functions will not require modification only the bodies of the `set` and `get` functions that directly manipulate the data member will need to change.
- A `public` `set` function should carefully scrutinize any attempt to modify the value of a data member to ensure that the new value is appropriate for that data item.
- Each class you declare should provide a constructor 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.

- A difference between constructors and functions is that constructors cannot return values, so they cannot specify a return type (not even `void`). Normally, constructors are declared `public`.

[Page 120]

- C++ requires a constructor call at the time each object is created, which helps ensure that every object is initialized before it is used in a program.
- A constructor that takes no arguments is a default constructor. In any class that does not include a constructor, the compiler provides a default constructor. The class programmer can also define a default constructor explicitly. If the programmer defines a constructor for a class, C++ will not create a default constructor.
- Class definitions, when packaged properly, can be reused by programmers worldwide.
- It is customary to define a class in a header file that has a `.h` filename extension.
- If the class's implementation changes, the class's clients should not be required to change.
- Interfaces define and standardize the ways in which things such as people and systems interact.
- The interface of a class describes the `public` member functions (also known as `public services`) that are made available to the class's clients. The interface describes what services clients can use and how to request those services, but does not specify how the class carries out the services.
- A fundamental principle of good software engineering is to separate interface from implementation. This makes programs easier to modify. Changes in the class's implementation do not affect the client as long as the class's interface originally provided to the client remains unchanged.
- A function prototype contains a function's name, its return type and the number, types and order of the parameters the function expects to receive.
- Once a class is defined and its member functions are declared (via function prototypes), the member functions should be defined in a separate source-code file
- For each member function defined outside of its corresponding class definition, the function name must be preceded by the class name and the binary scope resolution operator (`::`).
- Class `string`'s `length` member function returns the number of characters in a `string` object.
- Class `string`'s member function `substr` (short for "substring") returns a new `string` object created by copying part of an existing `string` object. The function's first argument specifies the starting position in the original `string` from which characters are copied. Its second argument specifies the number of characters to copy.
- In the UML, each class is modeled in a class diagram as a rectangle with three compartments. The top compartment contains the class name, centered horizontally in boldface. The middle compartment contains the class's attributes (data members in C++). The bottom compartment contains the class's operations (member functions and constructors in C++).
- The UML models operations by listing the operation name followed by a set of parentheses. A plus sign (+) preceding the operation name indicates a `public` operation in the UML (i.e., a `public` member function in C++).
- The UML models a parameter of an operation by listing the parameter name, followed by a colon and the parameter type between the parentheses following the operation name.
- The UML has its own data types. Not all the UML data types have the same names as the corresponding C++ types. The UML type `String` corresponds to the C++ type `string`.
- The UML represents data members as attributes by listing the attribute name, followed by a colon and the attribute type. Private attributes are preceded by a minus sign (-) in the UML.
- The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name.

- UML class diagrams do not specify return types for operations that do not return values.
- The UML models constructors as operations in a class diagram's third compartment. To distinguish a constructor from a class's operations, the UML places the word "constructor" between guillemets (« and ») before the constructor's name.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 121]

Terminology

access specifier

accessor

argument

attribute (UML)

binary scope resolution operator (::)

body of a class definition

calling function (caller)

camel case

class definition

class diagram (UML)

class-implementation programmer

client-code programmer

client of an object or class

compartment in a class diagram (UML)

consistent state

constructor

data hiding

data member

default constructor

default precision

defining a class

dot operator (.)

empty string

extensible language

function call

function header

function prototype

get function

getline function of <string> library

guillemets, « and » (UML)

header file

implementation of a class

instance of a class

interface of a class

invoke a member function

length member function of class string

local variable

member function

member-function call

message (send to an object)

minus (-) sign (UML)

mutator

object code

operation (UML)

operation parameter (UML)

parameter

parameter list

plus (+) sign (UML)

precision

private access specifier

public access specifier

public services of a class

return type

separate interface from implementation

set function

software engineering

source-code file

string class

<string> header file

substr member function of class string

UML class diagram

validation

validity checking

void return type

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 122]

• When a member function is defined outside the class definition, the function header must include the class name and the _____, followed by the function name to "tie" the member function to the class definition.

•

The source-code file and any other files that use a class can include the class's header file via an _____ preprocessor directive.

3.2

State whether each of the following is true or false. If false, explain why.

a.

By convention, function names begin with a capital letter and all subsequent words in the name begin with a capital letter.

b.

Empty parentheses following a function name in a function prototype indicate that the function does not require any parameters to perform its task.

c.

Data members or member functions declared with access specifier `private` are accessible to member functions of the class in which they are declared.

d.

Variables declared in the body of a particular member function are known as data members and can be used in all member functions of the class.

e.

Every function's body is delimited by left and right braces (`{` and `}`).

f.

Any source-code file that contains `int main()` can be used to execute a program.

g.

The types of arguments in a function call must match the types of the corresponding parameters in the function prototype's parameter list.

3.3

What is the difference between a local variable and a data member?

3.4

Explain the purpose of a function parameter. What is the difference between a parameter and an argument?

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 122 (continued)]

Answers to Self-Review Exercises

- 3.1** a) object. b) class. c) .h d) type, name. e) data member. f) access specifier. g) void. h) getline. i) binary scope resolution operator (: :). j) #include.
- 3.2** a) False. By convention, function names begin with a lowercase letter and all subsequent words in the name begin with a capital letter. b) True. c) True. d) False. Such variables are called local variables and can be used only in the member function in which they are declared. e) True. f) True. g) True.
- 3.3** A local variable is declared in the body of a function and can be used only from the point at which it is declared to the immediately following closing brace. A data member is declared in a class definition, but not in the body of any of the class's member functions. Every object (instance) of a class has a separate copy of the class's data members. Also, data members are accessible to all member functions of the class.
- 3.4** A parameter represents additional information that a function requires to perform its task. Each parameter required by a function is specified in the function header. An argument is the value supplied in the function call. When the function is called, the argument value is passed into the function parameter so that the function can perform its task.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 123]

3.10

Explain why a class might provide a set function and a get function for a data member.

3.11

(Modifying Class GradeBook) Modify class GradeBook ([Figs. 3.113.12](#)) as follows:

a.

Include a second `string` data member that represents the course instructor's name.

b.

Provide a set function to change the instructor's name and a get function to retrieve it.

c.

Modify the constructor to specify two parameters one for the course name and one for the instructor's name.

d.

Modify member function `displayMessage` such that it first outputs the welcome message and course name, then outputs "This course is presented by: " followed by the instructor's name.

Use your modified class in a test program that demonstrates the class's new capabilities.

3.12

(Account Class) Create a class called `Account` that a bank might use to represent customers' bank accounts. Your class should include one data member of type `int` to represent the account balance. [Note: In subsequent chapters, we'll use numbers that contain decimal points (e.g., 2.75) called floating-point values to represent dollar amounts.] Your class should provide a constructor that receives an initial balance and uses it to initialize the data member. The constructor should validate the initial balance to ensure that it is greater than or equal to 0. If not, the balance should be set to 0 and the constructor should display an error message, indicating that the initial balance was invalid. The class should provide three member functions. Member function `credit` should add an amount to the current balance. Member

function `debit` should withdraw money from the `Account` and should ensure that the debit amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the function should print a message indicating "Debit amount exceeded account balance." Member function `getBalance` should return the current balance. Create a program that creates two `Account` objects and tests the member functions of class `Account`.

3.13

(Invoice Class) Create a class called `Invoice` that a hardware store might use to represent an invoice for an item sold at the store. An `Invoice` should include four pieces of information as data members: a part number (type `string`), a part description (type `string`), a quantity of the item being purchased (type `int`) and a price per item (type `int`). [Note: In subsequent chapters, we'll use numbers that contain decimal points (e.g., 2.75) called floating-point values to represent dollar amounts.] Your class should have a constructor that initializes the four data members. Provide a set and a get function for each data member. In addition, provide a member function named `getInvoiceAmount` that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as an `int` value. If the quantity is not positive, it should be set to 0. If the price per item is not positive, it should be set to 0. Write a test program that demonstrates class `Invoice`'s capabilities.

3.14

(Employee Class) Create a class called `Employee` that includes three pieces of information as data members: a first name (type `string`), a last name (type `string`) and a monthly salary (type `int`). [Note: In subsequent chapters, we'll use numbers that contain decimal points (e.g., 2.75) called floating-point values to represent dollar amounts.] Your class should have a constructor that initializes the three data members. Provide a set and a get function for each data member. If the monthly salary is not positive, set it to 0. Write a test program that demonstrates class `Employee`'s capabilities. Create two `Employee` objects and display each object's yearly salary. Then give each `Employee` a 10 percent raise and display each `Employee`'s yearly salary again.

3.15

(Date Class) Create a class called `Date` that includes three pieces of information as data members: a month (type `int`), a day (type `int`) and a year (type `int`). Your class should have a constructor with three parameters that uses the parameters to initialize the three data members. For the purpose of this exercise, assume that the values provided for the year and day are correct, but ensure that the month value is in the range 1-12; if it is not, set the month to 1. Provide a set and a get function for each data member. Provide a member function `displayDate` that displays the month, day and year separated by forward slashes (/). Write a test program that demonstrates class `Date`'s capabilities.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 125]

Outline

[4.1 Introduction](#)

[4.2 Algorithms](#)

[4.3 Pseudocode](#)

[4.4 Control Structures](#)

[4.5 if Selection Statement](#)

[4.6 if...else Double-Selection Statement](#)

[4.7 while Repetition Statement](#)

[4.8 Formulating Algorithms: Counter-Controlled Repetition](#)

[4.9 Formulating Algorithms: Sentinel-Controlled Repetition](#)

[4.10 Formulating Algorithms: Nested Control Statements](#)

[4.11 Assignment Operators](#)

[4.12 Increment and Decrement Operators](#)

[4.13 \(Optional\) Software Engineering Case Study: Identifying Class Attributes in the ATM System](#)

[4.14 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

Answers to Self-Review Exercises

Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 125 (continued)]

4.1. Introduction

Before writing a program to solve a problem, we must have a thorough understanding of the problem and a carefully planned approach to solving it. When writing a program, we must also understand the types of building blocks that are available and employ proven program construction techniques. In this chapter and in [Chapter 5](#), Control Statements: Part 2, we discuss these issues in presenting of the theory and principles of structured programming. The concepts presented here are crucial to building effective classes and manipulating objects.

In this chapter, we introduce C++'s `if`, `if...else` and `while` statements, three of the building blocks that allow programmers 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.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 126]

Consider the "rise-and-shine algorithm" followed by one junior executive for getting out of bed and going to work: (1) Get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast, (6) carpool to work. This routine gets the executive to work well prepared to make critical decisions. Suppose that the same steps are performed in a slightly different order: (1) Get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work. In this case, our junior executive shows up for work soaking wet. Specifying the order in which statements (actions) execute in a computer program is called **program control**. This chapter investigates program control using C++'s **control statements**.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 127]

There are a few important aspects of the pseudocode in [Fig. 4.1](#). Notice that the pseudocode corresponds to code only in function `main`. This occurs because pseudocode is normally used for algorithms, not complete programs. In this case, the pseudocode is used to represent the algorithm. The function in which this code is placed is not important to the algorithm itself. For the same reason, line 23 of [Fig. 2.5](#) (the `return` statement) is not included in the pseudocode; this `return` statement is placed at the end of every `main` function and is not important to the algorithm. Finally, lines 911 of [Fig. 2.5](#) are not included in the pseudocode because these variable declarations are not executable statements.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

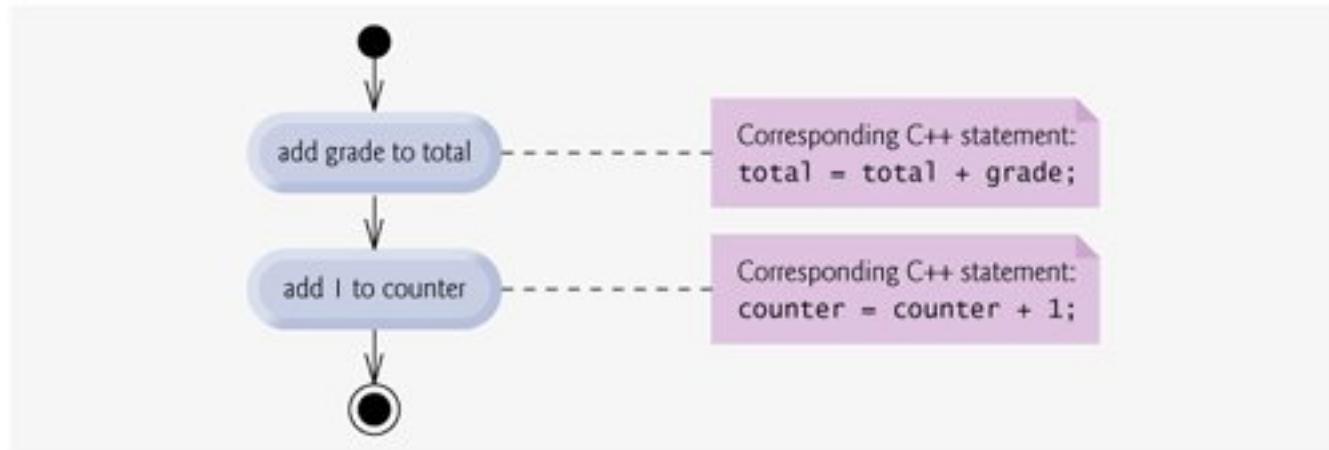
[Page 128]

Sequence Structure in C++

The sequence structure is built into C++. Unless directed otherwise, the computer executes C++ statements 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. 4.2 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 will soon see, anywhere a single action may be placed, we may place several actions in sequence.

Figure 4.2. Sequence-structure activity diagram.

[\[View full size image\]](#)



In this figure, the two statements involve adding a grade to a total variable and adding the value 1 to a counter variable. Such statements might appear in a program that takes the average of several student grades. To calculate an average, the total of the grades being averaged is divided by the number of grades. A counter variable would be used to keep track of the number of values being averaged. You will see similar statements in the program of Section 4.8.

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. 4.2. 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.

Like pseudocode, activity diagrams help programmers develop and represent algorithms, although many

programmers prefer pseudocode. Activity diagrams clearly show how control structures operate.

Consider the sequence-structure activity diagram of Fig. 4.2. 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. 4.2 first adds grade to total, then adds 1 to counter.

[Page 129]

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 4.2 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 4.2 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 17, 9, 10, 12 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 will 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 (also called **looping statements** or **loops**) that enable

programs to perform statements repeatedly as long as a condition (called the **loop-continuation 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 4.3 contains a complete list of C++ keywords.

[Page 130]

Figure 4.3. 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...else` and `switch`) and repetition statements (three types `while`, `for` and `do...while`). Each C++ program combines as many of these control statements as is appropriate for the algorithm the program implements. As with the sequence statement of Fig. 4.2, 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** make it easy to build programs—the control statements are attached to one another by connecting the exit point of one to the entry point of the next. This is similar to the way a child stacks building blocks, so we call this **control-statement stacking**. We will learn shortly that there is only one other way to connect control statements called **control-statement nesting**, in which one control statement is contained inside another. Thus, algorithms in C++ programs are constructed from only three kinds of control statements, combined in only two ways. This is the essence of simplicity.



Any C++ program we will 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).

PREV

page footer

NEXT

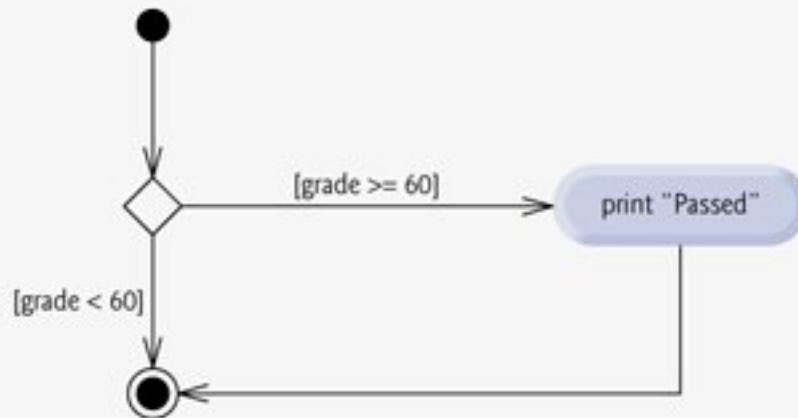
The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 132]

Figure 4.4. if single-selection statement activity diagram.

[\[View full size image\]](#)



We learned in [Chapter 1](#) that decisions can be based on conditions containing relational or equality operators. Actually, 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.

Note that the `if` statement is a single-entry/single-exit statement. We will see that the activity diagrams for the remaining control statements also contain initial states, transition arrows, action states that indicate actions to perform, decision symbols (with associated guard conditions) that indicate decisions to be made and final states. This is consistent with the **action/decision model of programming** we have been emphasizing.

We can envision seven bins, each containing only empty UML activity diagrams of one of the seven types of control statements. The programmer's task, then, is assembling a program from the activity diagrams of as many of each type of control statement as the algorithm demands, combining the activity diagrams in only two possible ways (stacking or nesting), then filling in the action states and decisions with action

expressions and guard conditions in a manner appropriate to form a structured implementation for the algorithm. We will discuss the variety of ways in which actions and decisions may be written.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 133]

prints "Passed" if the student's grade is greater than or equal to 60, but prints "Failed" if the student's grade is less than 60. In either case, after printing occurs, the next pseudocode statement in sequence is "performed."

The preceding pseudocode If...Else statement can be written in C++ as

```
if ( grade >= 60 )
    cout << "Passed";
else
    cout << "Failed";
```

Note that the body of the `else` is also indented. Whatever indentation convention you choose should be applied consistently throughout your programs. It is difficult to read programs that do not obey uniform spacing conventions.

Good Programming Practice 4.2



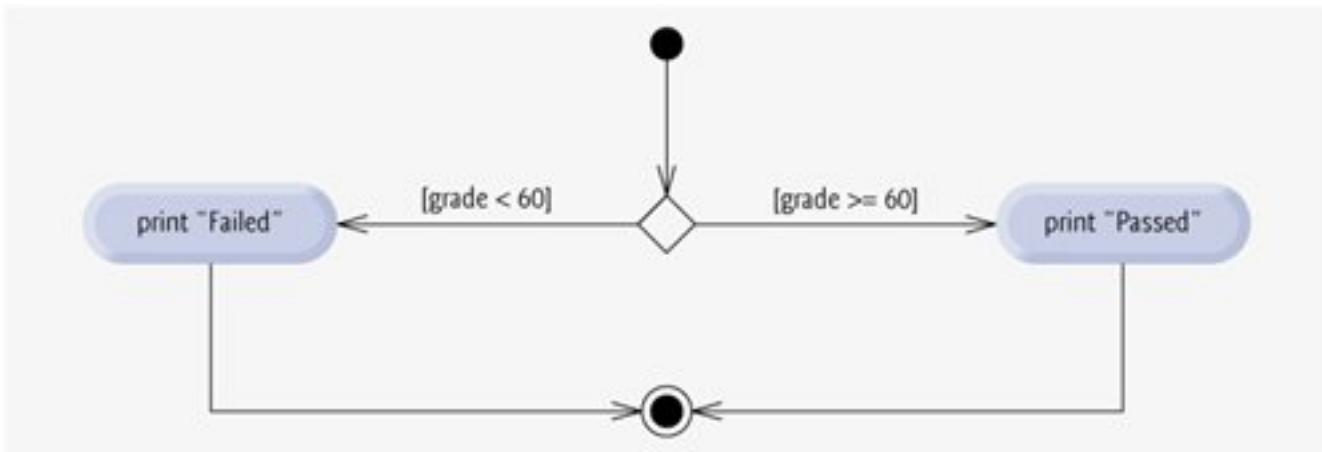
Indent both body statements of an `if...else` statement.

Good Programming Practice 4.3



If there are several levels of indentation, each level should be indented the same additional amount of space.

[Figure 4.5](#) 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. We continue to emphasize this action/decision model of computing. Imagine again a deep bin of empty UML activity diagrams of double-selection statements as many as the programmer might need to stack and nest with the activity diagrams of other control statements to form a structured implementation of an algorithm. The programmer fills in the action states and decision symbols with action expressions and guard conditions appropriate to the algorithm.

Figure 4.5. if...else double-selection statement activity diagram.[\[View full size image\]](#)

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 output statement

[Page 134]

```
cout << ( grade >= 60 ? "Passed" : "Failed" );
```

contains a conditional expression, `grade >= 60 ? "Passed" : "Failed"`, that evaluates to the string "Passed" if the condition `grade >= 60` is `True`, but evaluates to the string "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 will 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 program locations 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 pseudocode 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 student's grade is greater than or equal to 90
    Print "A"
Else
    If student's grade is greater than or equal to 80
        Print "B"
    Else
        If student's grade is greater than or equal to 70
            Print "C"
        Else
            If student's grade is greater than or equal to 60
                Print "D"
            Else
                Print "F"
```

This pseudocode can be written in C++ as

[Page 135]

```
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 `cout` statement after the first test will execute. After that `cout` 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. Such indentation often leaves little room on a line, forcing lines 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 and exit earlier than testing infrequently occurring cases 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";
```

[Page 136]

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 it was originally intended, we must 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. [Exercise 4.23](#) and [Exercise 4.24](#) further investigate the dangling-else problem.

Blocks

The if selection statement normally 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";
}
```

[Page 137]

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 is less than 60. This is an example of a logic error.

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.

Good Programming Practice 4.4



Always putting the braces in an `if...else` statement (or any control statement) helps prevent their accidental omission, especially when adding statements to an `if` or `else` clause at a later time. To avoid omitting one or both of the braces, some programmers prefer to type the beginning and ending braces of blocks even before typing the individual statements within the braces.

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).



[page footer](#)



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 138]

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` statement multiplies `product` by 3, so `product` takes on the values 9, 27, 81 and 243 successively. When `product` becomes 243, the `while` statement 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

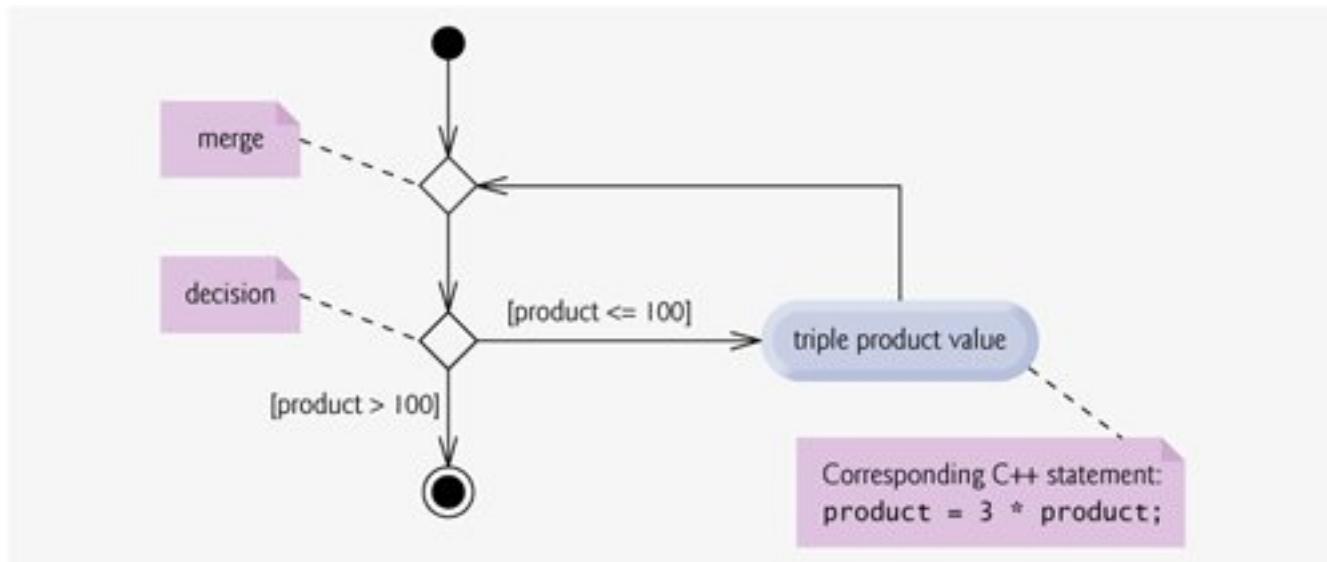


Not providing, in the body of a `while` statement, an action that eventually causes the condition in the `while` to become `false` normally results in a logic error called an infinite loop, in which the repetition statement never terminates. This can make a program appear to "hang" or "freeze" if the loop body does not contain statements that interact with the user.

The UML activity diagram of Fig. 4.6 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.

Figure 4.6. while repetition statement UML activity diagram.

(This item is displayed on page 139 in the print version)

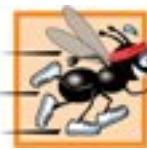
[\[View full size image\]](#)

The diagram of Fig. 4.6 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.

Imagine a deep bin of empty UML `while` repetition statement activity diagrams as many as the programmer might need to stack and nest with the activity diagrams of other control statements to form a structured implementation of an algorithm. The programmer fills in the action states and decision symbols with action expressions and guard conditions appropriate to the algorithm.

[\[Page 139\]](#)

Performance Tip 4.3



Many of the performance tips we mention in this text result in only small improvements, so the reader might be tempted to ignore them. However, a small performance improvement for code that executes many times in a loop can result in substantial overall performance improvement.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 140]

Software Engineering Observation 4.3



Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working C++ program from the algorithm is normally straightforward.

Note the references in the pseudocode algorithm of Fig. 4.7 to a total and a counter. A **total** is a variable used to accumulate the sum of several values. A **counter** is a variable used to countin this case, the grade counter indicates which of the 10 grades is about to be entered by the user. Variables used to store totals are normally initialized to zero before being used in a program; otherwise, the sum would include the previous value stored in the total's memory location.

[Page 142]

Enhancing GradeBook Validation

Before we discuss the implementation of the class average algorithm, 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 if 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 if 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 have modified our code, replacing the two `if` statements with one `if...else` statement (lines 2128 of Fig. 4.9).

Implementing Counter-Controlled Repetition in Class GradeBook

Class GradeBook (Fig. 4.8Fig. 4.9) contains a constructor (declared in line 11 of Fig. 4.8 and defined in lines 1215 of Fig. 4.9) that assigns a value to the class's instance variable `courseName` (declared in line 17 of Fig. 4.8). Lines 1929, 3235 and 3842 of Fig. 4.9 define member functions `setCourseName`, `getCourseName` and `displayMessage`, respectively. Lines 4571 define member function `determineClassAverage`, which implements the class average algorithm described by the pseudocode in Fig. 4.7.

Lines 4750 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 a data structure known as 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.

[Page 143]

Good Programming Practice 4.5



Separate declarations from other statements in functions with a blank line for readability.

Lines 5354 initialize `total` to 0 and `gradeCounter` to 1. Note that variables `total` and `gradeCounter` are initialized before they are used in a calculation. Counter variables normally are initialized to zero or one, depending on their use (we will present examples showing each possibility). An uninitialized variable contains a **"garbage" value** (also called an **undefined value**) the value last stored in the memory location reserved for that variable. 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.

[Page 144]

Common Programming Error 4.6



Not initializing counters and totals can lead to logic errors.

Error-Prevention Tip 4.2



Initialize each counter and total, either in its declaration or in an assignment statement. Totals are normally initialized to 0. Counters are normally initialized to 0 or 1, depending on how they are used (we will show examples of when to use 0 and when to use 1).

Good Programming Practice 4.6



Declare each variable on a separate line with its own comment to make programs more readable.

Line 57 indicates that the `while` statement should continue looping (also called **iterating**) 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 5863).

Line 59 displays the prompt "Enter grade: ". This line corresponds to the pseudocode statement "Prompt the user to enter the next grade." Line 60 reads the grade entered by the user and assigns it to variable `grade`. This line corresponds to the pseudocode statement "Input the next grade." Recall that variable `grade` was not initialized earlier in the program, because the program obtains the value for `grade` from the user during each iteration of the loop. 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. 4.10).

Figure 4.10. Class average problem using counter-controlled repetition: Creating an object of class GradeBook (Fig. 4.8Fig. 4.9) and invoking its `determineClassAverage` member function.

(This item is displayed on page 143 in the print version)

```

1 // Fig. 4.10: fig04_10.cpp
2 // Create GradeBook object and invoke its determineClassAverage function.
3 #include "GradeBook.h" // include definition of class GradeBook
4
5 int main()
6 {
7     // create GradeBook object myGradeBook and
8     // pass course name to constructor
9     GradeBook myGradeBook( "CS101 C++ Programming" );
10
11    myGradeBook.displayMessage(); // display welcome message
12    myGradeBook.determineClassAverage(); // find average of 10 grades
13    return 0; // indicate successful termination
14 } // 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

```

Demonstrating Class GradeBook

Figure 4.10 contains this application's main function, which creates an object of class GradeBook and demonstrates its capabilities. Line 9 of Fig. 4.10 creates a new GradeBook object called myGradeBook. The string in line 9 is passed to the GradeBook constructor (lines 1215 of Fig. 4.9). Line 11 of Fig. 4.10 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.

function performs the algorithm shown in the pseudocode of Fig. 4.7.

Notes on Integer Division and Truncation

The averaging calculation performed by member function `determineClassAverage` in response to the function call at line 12 in Fig. 4.10 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. 4.9) is the integer 84, because `total` and 10 are both integers. Dividing two integers results in integer division and the fractional part of the calculation is lost (i.e., **truncated**). We will see how to obtain a result that includes a decimal point from the averaging calculation in the next section.

[Page 145]

Common Programming Error 4.7



Assuming that integer division rounds (rather than truncates) can lead to incorrect results. For example, $7 \div 4$, which yields 1.75 in conventional arithmetic, truncates to 1 in integer arithmetic, rather than rounding to 2.

In Fig. 4.9, if line 66 used `gradeCounter` rather than 10 for the calculation, the output for this program would display an incorrect value, 76. This occurs because in the final iteration of the `while` statement, `gradeCounter` was incremented to the value 11 in line 62.

Common Programming Error 4.8



Using a loop's counter-control variable in a calculation after the loop often causes a common logic error called an **off-by-one-error**. In a counter-controlled loop that counts up by one each time through the loop, the loop terminates when the counter's value is one higher than its last legitimate value (i.e., 11 in the case of counting from 1 to 10).

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 146]

Common Programming Error 4.9



Choosing a sentinel value that is also a legitimate data value is a logic error.

Developing the Pseudocode Algorithm with Top-Down, Stepwise Refinement: The Top and First Refinement

We approach the class average program with a technique called **top-down, stepwise refinement**, a technique that is essential to the development of well-structured programs. We begin with a pseudocode representation of the **top**a single statement that conveys the overall function of the program:

Determine the class average for the quiz

The top is, in effect, a complete representation of a program. Unfortunately, the top (as in this case) rarely conveys sufficient detail from which to write a program. So we now begin the refinement process. We divide the top into a series of smaller tasks and list these in the order in which they need to be performed. This results in the following **first refinement**.

Initialize variables

Input, sum and count the quiz grades

Calculate and print the total of all student grades and the class average

This refinement uses only the sequence structurethe steps listed should execute in order, one after the other.

Software Engineering Observation 4.4



Each refinement, as well as the top itself, is a complete specification of the algorithm; only the level of detail varies.

Software Engineering Observation 4.5



Many programs can be divided logically into three phases: an initialization phase that initializes the program variables; a processing phase that inputs data values and adjusts program variables (such as counters and totals) accordingly; and a termination phase that calculates and outputs the final results.

Proceeding to the Second Refinement

The preceding Software Engineering Observation is often all you need for the first refinement in the top-down process. To proceed to the next level of refinement, i.e., the **second refinement**, we commit to specific variables. In this example, we need a running total of the numbers, a count of how many numbers have been processed, a variable to receive the value of each grade as it is input by the user and a variable to hold the calculated average.

The pseudocode statement

Initialize variables

can be refined as follows:

**Initialize total to zero
Initialize counter to zero**

Only the variables total and counter need to be initialized before they are used. The variables average and grade (for the calculated average and the user input, respectively) need not be initialized, because their values will be replaced as they are calculated or input.

[Page 147]

The pseudocode statement

Input, sum and count the quiz grades

requires a repetition statement (i.e., a loop) that successively inputs each grade. We do not know in advance how many grades are to be processed, so we will use sentinel-controlled repetition. The user enters legitimate grades one at a time. After entering the last legitimate grade, the user enters the sentinel value. The program tests for the sentinel value after each grade is input and terminates the loop when the user enters the sentinel value. The second refinement of the preceding pseudocode statement is then

Prompt the user to enter the first grade

```

Input the first grade (possibly the sentinel)

While the user has not yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Prompt the user to enter the next grade
    Input the next grade (possibly the sentinel)

```

In pseudocode, we do not use braces around the statements that form the body of the While structure. We simply indent the statements under the While to show that they belong to the While. Again, pseudocode is only an informal program development aid.

The pseudocode statement

Calculate and print the total of all student grades and the class average

can be refined as follows:

```

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the total of the grades for all students in the class
    Print the class average
else
    Print "No grades were entered"

```

We are careful here to test for the possibility of division by zero normally a **fatal logic error** that, if undetected, would cause the program to fail (often called "**bombing**" or "**crashing**"). The complete second refinement of the pseudocode for the class average problem is shown in Fig. 4.11.

Figure 4.11. Class average problem pseudocode algorithm with sentinel-controlled repetition.

(This item is displayed on page 148 in the print version)

```

1 Initialize total to zero
2 Initialize counter to zero
3
4 Prompt the user to enter the first grade
5 Input the first grade (possibly the sentinel)
6
7 While the user has not yet entered the sentinel
8   Add this grade into the running total
9   Add one to the grade counter
10  Prompt the user to enter the next grade
11  Input the next grade (possibly the sentinel)
12
13 If the counter is not equal to zero
14   Set the average to the total divided by the counter
15   Print the total of the grades for all students in the class
16   Print the class average
17 else
18   Print "No grades were entered"

```

Common Programming Error 4.10



An attempt to divide by zero normally causes a fatal runtime error.

Error-Prevention Tip 4.3



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.

In Fig. 4.7 and Fig. 4.11, we include some blank lines and indentation in the pseudocode to make it more readable. The blank lines separate the pseudocode algorithms into their various phases, and the indentation emphasizes the control statement bodies.

developed after only two levels of refinement. Sometimes more levels are necessary.

Software Engineering Observation 4.6



Terminate the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for you to be able to convert the pseudocode to C++. Normally, implementing the C++ program is then straightforward.

Software Engineering Observation 4.7



Many experienced programmers write programs without ever using program development tools like pseudocode. These programmers feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs. Although this method might work for simple and familiar problems, it can lead to serious difficulties in large, complex projects.

Implementing Sentinel-Controlled Repetition in Class GradeBook

Figures 4.12 and 4.13 show the C++ class `GradeBook` containing member function `determineClassAverage` that implements the pseudocode algorithm of Fig. 4.11 (this class is demonstrated in Fig. 4.14). Although each grade entered is an integer, the averaging calculation is likely to produce a number with a decimal point in other words, a real number or **floating-point number** (e.g., 7.33, 0.0975 or 1000.12345). 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 in memory, including `float` and `double`. The primary difference between these types is that, compared to `float` variables, `double` variables can 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.

[Page 149]

Figure 4.12. Class average problem using sentinel-controlled repetition: GradeBook header file.

```

1 // Fig. 4.12: GradeBook.h
2 // Definition of class GradeBook that determines a class average.
3 // Member functions are defined in GradeBook.cpp
4 #include <string> // program uses C++ standard string class
5 using std::string;
6
7 // GradeBook class definition
8 class GradeBook
9 {
10 public:
11     GradeBook( string ); // constructor initializes course name
12     void setCourseName( string ); // function to set the course name
13     string getCourseName(); // function to retrieve the course name
14     void displayMessage(); // display a welcome message
15     void determineClassAverage(); // averages grades entered by the user
16 private:
17     string courseName; // course name for this GradeBook
18 } // end class GradeBook

```

Figure 4.13. Class average problem using sentinel-controlled repetition: GradeBook source code file.

(This item is displayed on pages 149 - 151 in the print version)

```

1 // Fig. 4.13: GradeBook.cpp
2 // Member-function definitions for class GradeBook that solves the
3 // class average program with sentinel-controlled repetition.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed; // ensures that decimal point is displayed
9
10 #include <iomanip> // parameterized stream manipulators
11 using std::setprecision; // sets numeric output precision
12
13 // include definition of class GradeBook from GradeBook.h
14 #include "GradeBook.h"
15
16 // constructor initializes courseName with string supplied as argument
17 GradeBook::GradeBook( string name )
18 {
19     setCourseName( name ); // validate and store courseName
20 } // end GradeBook constructor
21

```

```

22 // function to set the course name;
23 // ensures that the course name has at most 25 characters
24 void GradeBook::setCourseName( string name )
25 {
26     if ( name.length() <= 25 ) // if name has 25 or fewer characters
27         courseName = name; // store the course name in the object
28     else // if name is longer than 25 characters
29     { // set courseName to first 25 characters of parameter name
30         courseName = name.substr( 0, 25 ); // select first 25 characters
31         cout << "Name \" " << name << "\" exceeds maximum length (25).\n"
32             << "Limiting courseName to first 25 characters.\n" << endl;
33     } // end if...else
34 } // end function setCourseName
35
36 // function to retrieve the course name
37 string GradeBook::getCourseName()
38 {
39     return courseName;
40 } // end function getCourseName
41
42 // display a welcome message to the GradeBook user
43 void GradeBook::displayMessage()
44 {
45     cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
46         << endl;
47 } // end function displayMessage
48
49 // determine class average based on 10 grades entered by user
50 void GradeBook::determineClassAverage()
51 {
52     int total; // sum of grades entered by user
53     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      // prompt for input and read next grade from user
72      cout << "Enter grade or -1 to quit: ";
73      cin >> grade; // input grade or sentinel value
74  } // end while
75
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

```

Figure 4.14. Class average problem using sentinel-controlled repetition: Creating an object of class GradeBook (Fig. 4.12Fig. 4.13) and invoking its determineClassAverage member function.

(This item is displayed on pages 151 - 152 in the print version)

```

1 // Fig. 4.14: fig04_14.cpp
2 // Create GradeBook object and invoke its determineClassAverage function.
3
4 // include definition of class GradeBook from GradeBook.h
5 #include "GradeBook.h"
6
7 int main()
8 {
9     // create GradeBook object myGradeBook and
10    // pass course name to constructor
11    GradeBook myGradeBook( "CS101 C++ Programming" );
12
13    myGradeBook.displayMessage(); // display welcome message
14    myGradeBook.determineClassAverage(); // find average of 10 grades
15    return 0; // indicate successful termination
16 } // 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
```

In this example, we see that control statements can be stacked on top of one another (in sequence) just as a child stacks building blocks. The `while` statement (lines 6775 of Fig. 4.13) is immediately followed by an `if...else` statement (lines 7890) in sequence. Much of the code in this program is identical to the code in Fig. 4.9, so we concentrate on the new features and issues.

[Page 152]

Line 55 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 4.7



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.

Program Logic for Sentinel-Controlled Repetition vs. Counter-Controlled Repetition

Compare the program logic for sentinel-controlled repetition in this application with that for counter-controlled repetition in Fig. 4.9. In counter-controlled repetition, each iteration of the `while` statement

(lines 5763 of Fig. 4.9) reads a value from the user, for the specified number of iterations. In sentinel-controlled repetition, the program reads the first value (lines 6364 of Fig. 4.13) before reaching the while. This value determines whether the program's flow of control should enter the body of the while. If the condition of the while is false, the user entered the sentinel value, so the body of the while does not execute (i.e., no grades were entered). If, on the other hand, the condition is true, the body begins execution, and the loop adds the grade value to the total (line 69). Then lines 7374 in the loop's body input the next value from the user. Next, program control reaches the closing right brace () of the body at line 75, so execution continues with the test of the while's condition (line 67). The condition uses the most recent grade input by the user to determine whether the loop's body should execute again. Note that the value of variable grade is always input from the user immediately before the program tests the while condition. This allows the program to determine whether the value just input is the sentinel value before the program processes that value (i.e., adds it to the total and increments gradeCounter). If the sentinel value is input, the loop terminates, and the program does not add 1 to the total.

[Page 153]

After the loop terminates, the if...else statement at lines 7890 executes. The condition at line 78 determines whether any grades were entered. If none were, the else part (lines 8990) of the if...else statement executes and displays the message "No grades were entered" and the member function returns control to the calling function.

Notice the block in the while loop in Fig. 4.13. Without the braces, the last three statements in the body of the loop would fall outside the loop, causing the computer to interpret this code incorrectly, as follows:

```
// loop until sentinel value read from user
while ( grade != -1 )
    total = total + grade; // add grade to total
    gradeCounter = gradeCounter + 1; // increment counter

// prompt for input and read next grade from user
cout << "Enter grade or -1 to quit: ";
cin >> grade;
```

This would cause an infinite loop in the program if the user did not input 1 for the first grade (at line 64).

Common Programming Error 4.11



Omitting the braces that delimit a block can lead to logic errors, such as infinite loops. To prevent this problem, some programmers enclose the body of every control statement in braces, even if the body contains only a single statement.

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 today. Variables of type `double` represent **double-precision floating-point numbers**. These require twice as much memory as `float` variables and provide 15 significant digits on most 32-bit systems today approximately double the precision of `float` variables. For the range of values required by most programs, variables of type `float` 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`.

Floating-point numbers often arise as a result of division. In conventional arithmetic, when we divide 10 by 3, the result is 3.333333..., with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so clearly the stored floating-point value can be only an approximation.

Common Programming Error 4.12



Using floating-point numbers in a manner that assumes they are represented exactly (e.g., using them in comparisons for equality) can lead to incorrect results. Floating-point numbers are represented only approximately by most computers.

[Page 154]

Although floating-point numbers are not always 100% precise, they have numerous applications. For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits. When we read the temperature on a thermometer as 98.6, it may actually be 98.5999473210643. Calling this number simply 98.6 is fine for most applications involving body temperatures. Due to the imprecise nature of floating-point numbers, type `double` is preferred over type `float`, because `double` variables can represent floating-point numbers more accurately. For this reason, we use type `double` throughout the book.

Converting Between Fundamental Types Explicitly and Implicitly

The variable `average` is declared to be of type `double` (line 55 of [Fig. 4.13](#)) 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 4.13



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.

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 the programmer 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. 4.22](#)).

[Page 155]

Formatting for Floating-Point Numbers

The formatting capabilities in [Fig. 4.13](#) 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 names from the `<iomanip>` header file that are 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 will 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 and 10, multiplied by a power of 10. For instance, the value 3,100 would be displayed in scientific notation as 3.1×10^3 . 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. 4.13](#) 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. 4.14](#) total 257, which yields the average 85.66666.... 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.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 157]

Here, too, even though we have a complete representation of the entire program, further refinement is necessary. We now commit to specific variables. Counters are needed to record the passes and failures, a counter will be used to control the looping process and a variable is needed to store the user input. The last variable is not initialized, because its value is read from the user during each iteration of the loop.

The pseudocode statement

Initialize variables

can be refined as follows:

Initialize passes to zero

Initialize failures to zero

Initialize student counter to one

Notice that only the counters are initialized at the start of the algorithm.

The pseudocode statement

Input the 10 exam results, and count passes and failures

requires a loop that successively inputs the result of each exam. Here it is known in advance that there are precisely 10 exam results, so counter-controlled looping is appropriate. Inside the loop (i.e., **nested** within the loop), an **if...else** statement will determine whether each exam result is a pass or a failure and will increment the appropriate counter. The refinement of the preceding pseudocode statement is then

While student counter is less than or equal to 10

Prompt the user to enter the next exam result

Input the next exam result

If the student passed

Add one to passes

Else

Add one to failures

Add one to student counter

We use blank lines to isolate the If...Else control structure, which improves readability.

The pseudocode statement

Print a summary of the exam results and decide whether tuition should be raised

can be refined as follows:

Print the number of passes

Print the number of failures

If more than eight students passed

 Print "Raise tuition"

The complete second refinement appears in Fig. 4.15. Notice that blank lines are also used to set off the While structure for program readability. This pseudocode is now sufficiently refined for conversion to C++.

Figure 4.15. Pseudocode for examination-results problem.

(This item is displayed on page 158 in the print version)

```

1 Initialize passes to zero
2 Initialize failures to zero
3 Initialize student counter to one
4
5 While student counter is less than or equal to 10
6     Prompt the user to enter the next exam result
7     Input the next exam result
8
9     If the student passed
10        Add one to passes
11    Else
12        Add one to failures
13
14    Add one to student counter
15
16 Print the number of passes
17 Print the number of failures
18

```

19 If more than eight students passed
20 Print "Raise tuition"

Conversion to Class Analysis

The C++ class that implements the pseudocode algorithm is shown in Fig. 4.16Fig. 4.17, and two sample executions appear in Fig. 4.18.

[Page 158]

Figure 4.16. Examination-results problem: Analysis header file.

```
1 // Fig. 4.16: Analysis.h
2 // Definition of class Analysis that analyzes examination results.
3 // Member function is defined in Analysis.cpp
4
5 // Analysis class definition
6 class Analysis
7 {
8 public:
9     void processExamResults(); // process 10 students' examination results
10 };
```

Figure 4.17. Examination-results problem: Nested control statements in Analysis source code file.

(This item is displayed on pages 158 - 159 in the print version)

```

1 // Fig. 4.17: Analysis.cpp
2 // Member-function definitions for class Analysis that
3 // analyzes examination results.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 // include definition of class Analysis from Analysis.h
10 #include "Analysis.h"
11
12 // process the examination results of 10 students
13 void Analysis::processExamResults()
14 {
15     // initializing variables in declarations
16     int passes = 0; // number of passes
17     int failures = 0; // number of failures
18     int studentCounter = 1; // student counter
19     int result; // one exam result (1 = pass, 2 = fail)
20
21     // process 10 students using counter-controlled loop
22     while ( studentCounter <= 10 )
23     {
24         // prompt user for input and obtain value from user
25         cout << "Enter result (1 = pass, 2 = fail): ";
26         cin >> result; // input result
27
28         // if...else nested in while
29         if ( result == 1 )           // if result is 1,
30             passes = passes + 1;    // increment passes;
31         else                      // else result is not 1, so
32             failures = failures + 1; // increment failures
33
34         // increment studentCounter so loop eventually terminates
35         studentCounter = studentCounter + 1;
36     } // end while
37
38     // termination phase; display number of passes and failures
39     cout << "Passed " << passes << "\nFailed " << failures << endl;
40
41     // determine whether more than eight students passed
42     if ( passes > 8 )
43         cout << "Raise tuition " << endl;
44 } // end function processExamResults

```

Figure 4.18. Test program for class Analysis.

(This item is displayed on pages 159 - 160 in the print version)

```
1 // Fig. 4.18: fig04_18.cpp
2 // Test program for class Analysis.
3 #include "Analysis.h" // include definition of class Analysis
4
5 int main()
6 {
7     Analysis application; // create Analysis object
8     application.processExamResults(); // call function to process results
9     return 0; // indicate successful termination
10 } // 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
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

```

[Page 160]

Lines 1618 of Fig. 4.17 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 2236) loops 10 times. During each iteration, the loop inputs and processes one exam result. Notice that the `if...else` statement (lines 2932) 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 at 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 at lines 4243 determines whether more than eight students passed the exam and, if so, outputs the message "Raise Tuition".

Demonstrating Class Analysis

Figure 4.18 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 4.18 shows the input and output from two sample executions of the program. At the end of the first sample execution, the condition at line 42 of member function `processExamResults` in Fig. 4.17 is true more than eight students passed the exam, so the program outputs a message indicating that the tuition should be raised.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)
[NEXT ▶](#)

[Page 161 (continued)]

4.11. 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 4.19](#) shows the arithmetic assignment operators, sample expressions using these operators and explanations.

Figure 4.19. Arithmetic assignment operators.

Assignment operator	Sample expression	Explanation	Assigns
	Assume: <code>int c = 3, d = 5, e = 4, f = 6, g = 12;</code>		

<code>+ =</code>	<code>c += 7</code>	<code>c = c + 7</code>	10 to c
<code>- =</code>	<code>d -= 4</code>	<code>d = d - 4</code>	1 to d
<code>* =</code>	<code>e *= 5</code>	<code>e = e * 5</code>	20 to e
<code>/ =</code>	<code>f /= 3</code>	<code>f = f / 3</code>	2 to f
<code>% =</code>	<code>g %= 9</code>	<code>g = g % 9</code>	3 to g

 PREVNEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 162]

Figure 4.20. Increment and decrement operators.

Operator	Called	Sample expression	Explanation
<code>++</code>	preincrement	<code>++a</code>	Increment <code>a</code> by 1, then use the new value of <code>a</code> in the expression in which <code>a</code> resides.
<code>++</code>	postincrement	<code>a++</code>	Use the current value of <code>a</code> in the expression in which <code>a</code> resides, then increment <code>a</code> by 1.
<code>--</code>	predecrement	<code>--b</code>	Decrement <code>b</code> by 1, then use the new value of <code>b</code> in the expression in which <code>b</code> resides.
<code>--</code>	postdecrement	<code>b--</code>	Use the current value of <code>b</code> in the expression in which <code>b</code> resides, then decrement <code>b</code> 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, and 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, and then the variable's value is incremented (decremented) by 1.

Good Programming Practice 4.8



Unlike binary operators, the unary increment and decrement operators should be placed next to their operands, with no intervening spaces.

[Figure 4.21](#) 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 will use a mechanical example contained entirely within the `main` function of a single source code file.

Figure 4.21. Preincrementing and postincrementing.

(This item is displayed on page 163 in the print version)

```
1 // Fig. 4.21: fig04_21.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.

[Page 163]

Line 20 resets `c`'s value to 5, and line 21 outputs `c`'s 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. 4.17

```

passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;

```

can be written more concisely with assignment operators as

```

passes += 1;
failures += 1;
studentCounter += 1;

```

[Page 164]

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 4.14



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 4.22 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 (++) and predecrement (--), plus (+) and minus (-), and the assignment operators =, +=, -=, *=, /= and %= associate from right to left. All other operators in the operator precedence chart of Fig. 4.22 associate from left to right. The third column names the various groups of operators.

Figure 4.22. Operator precedence for the operators encountered so far in the text.

Operators		Associativity	Type
()		left to right	parentheses
++	--	static_cast< type >()	unary (postfix)

<code>++</code>	<code>--</code>	<code>+</code>	<code>-</code>			right to left	unary (prefix)
<code>*</code>	<code>/</code>	<code>%</code>				left to right	multiplicative
<code>+</code>	<code>-</code>					left to right	additive
<code><<</code>	<code>>></code>					left to right	insertion/extraction
<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>			left to right	relational
<code>==</code>	<code>!=</code>					left to right	equality
<code>? :</code>						right to left	conditional
<code>=</code>	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	right to left	assignment

 PREVNEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 166]

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]

[Page 166]

Figure 4.23 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 5.11), 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 document also suggest some differences in the attributes required by each transaction class. The requirements document 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 document 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 document 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.]

[Page 167]

Class `CashDispenser` has one attribute. The requirements document 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 documents 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 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. 4.24 lists some of the attributes for the classes in our systemthe descriptive words and phrases in Fig. 4.23 helped us identify these attributes. For simplicity, Fig. 4.24 does not show the associations among classeswe showed these in Fig. 3.23. This is a common practice of systems designers when designs are being developed. Recall from Section 5.11 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.

Figure 4.24. Classes with attributes.

(This item is displayed on page 168 in the print version)

[\[View full size image\]](#)



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. 4.24](#) we discuss the reasoning behind this decision shortly. [Note: [Figure 4.24](#) lists UML data types for the attributes. When we implement the system, we will 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. This number will be determined at execution time based on the account number entered by the current ATM user.

[Page 168]

[Figure 4.24](#) 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 phases of design or when we implement these classes in C++. This is perfectly normal for the iterative process of software engineering.

Software Engineering Observation 4.8



At 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. 4.24](#) 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. 4.24](#) (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 will be required to create an attribute of class BankDatabase to hold zero or more Account objects. Similarly, we will 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. 4.24](#), 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.

[Page 169]

The class diagram in [Fig. 4.24](#) provides a solid basis for the structure of our model, but the diagram is not complete. In [Section 5.11](#), 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 will 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 document.

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. 4.24:

```
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.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 170]

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.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 171]

- A diamond in an activity diagram also represents the merge symbol, which joins two flows of activity into one. 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.
- Top-down, stepwise refinement is a process for refining pseudocode by maintaining a complete representation of the program during each refinement.
- There are three types of control structuressequence, selection and repetition.
- The sequence structure is built into C++ by default, statements execute in the order they appear.
- A selection structure chooses among alternative courses of action.
- The `if` single-selection statement either performs (selects) an action if a condition is true, or skips the action if the condition is false.
- The `if...else` double-selection statement performs (selects) an action if a condition is true and performs a different action if the condition is false.
- To include several statements in an `if`'s body (or the body of an `else` for an `if...else` statement), enclose the statements in braces (`{` and `}`). A set of statements contained within a pair of braces is called a block. A block can be placed anywhere in a program that a single statement can be placed.
- A null statement, indicating that no action is to be taken, is indicated by a semicolon (`:`).
- A repetition statement specifies that an action is to be repeated while some condition remains true.
- A value that contains a fractional part is referred to as a floating-point number and is represented approximately by data types such as `float` and `double`.
- Counter-controlled repetition is used when the number of repetitions is known before a loop begins executing, i.e., when there is definite repetition.
- The unary cast operator `static_cast` can be used to create a temporary floating-point copy of its operand.
- Unary operators take only one operand; binary operators take two.
- The parameterized stream manipulator `setprecision` indicates the number of digits of precision that should be displayed to the right of the decimal point.
- The stream manipulator `fixed` indicates that floating-point values should be output in so-called fixed-point format, as opposed to scientific notation.
- Sentinel-controlled repetition is used when the number of repetitions is not known before a loop begins executing, i.e., when there is indefinite repetition.
- A nested control statement appears in the body of another control statement.
- C++ provides the arithmetic assignment operators `+=`, `-=`, `*=`, `/=` and `%=` for abbreviating assignment expressions.
- The increment operator, `++`, and the decrement operator, `--`, increment or decrement a variable by 1, respectively. If the operator is prefixed to the variable, the variable is incremented or decremented by 1 first, and then its new value is used in the expression in which it appears. If the operator is postfix to the variable, the variable is first used in the expression in which it appears, and then the variable's value is incremented or decremented by 1.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 173]

second refinement

selection statement

sentinel-controlled repetition

sentinel value

sequence statement

sequence-statement activity diagram

sequential execution

`setprecision` stream manipulator

`showpoint` stream manipulator

signal value

single-entry/single-exit control statement

single-selection `if` statement

single-precision floating-point number

small circle symbol

solid circle symbol

stream manipulator

structured programming

ternary operator

top

top-down, stepwise refinement

total

transfer of control

transition

transition arrow symbol

truncate

unary cast operator

unary minus (-) operator

unary operator

unary plus (+) operator

undefined value

while repetition statement

workflow of a portion of a software system

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 174]

4.7

Write single C++ statements that do the following:

a.

Input integer variable `x` with `cin` and `>>`.

b.

Input integer variable `y` with `cin` and `>>`.

c.

Set integer variable `i` to 1.

d.

Set integer variable `power` to 1.

e.

Multiply variable `power` by `x` and assign the result to `power`.

f.

Postincrement variable `i` by 1.

g.

Determine whether `i` is less than or equal to `y`.

h.

Output integer variable `power` with `cout` and `<<`.

4.8

Write a C++ program that uses the statements in Exercise 4.7 to calculate x raised to the y power. The program should have a `while` repetition statement.

4.9

Identify and correct the errors in each of the following:

a.

```
while ( c <= 5 )
{
    product *= c;
    c++;
}
```

b.

```
cin << value;
```

c.

```
if ( gender == 1 )
    cout << "Woman" << endl;
else;
    cout << "Man" << endl;
```

4.10

What is wrong with the following `while` repetition statement?

```
while ( z >= 0 )
    sum += z;
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 175]

```
1 // Exercise 4.5 Solution: ex04_05.cpp 2 // Calculate the sum of the integers from 1 to 10. 3 #include
<iostream> 4 using std::cout; 5 using std::endl; 6 7 int main() 8 { 9 int sum; // stores sum of integers 1
to 10 10 int x; // counter 11 12 x = 1; // count from 1 13 sum = 0; // initialize sum 14 15 while ( x <=
10 ) // loop 10 times 16 { 17 sum += x; // add x to sum 18 x++; // increment x 19 } // end while 20 21
cout << "The sum is: " << sum << endl; 22 return 0; // indicate successful termination 23 } // end main
```

The sum is: 55

4.6

a.

```
product = 25, x = 6;
```

b.

```
quotient = 0, x = 6;
```

```
1 // Exercise 4.6 Solution: ex04_06.cpp
2 // Calculate the value of product and quotient.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
{
9     int x = 5;
10    int product = 5;
11    int quotient = 5;
12
13    // part a
14    product *= x++; // part a statement
15    cout << "Value of product after calculation: " << product << endl;
16    cout << "Value of x after calculation: " << x << endl << endl;
17
18    // part b
19    x = 5; // reset value of x
```

```
20     quotient /= ++x; // part b statement
21     cout << "Value of quotient after calculation: " << quotient << endl;
22     cout << "Value of x after calculation: " << x << endl << endl;
23     return 0; // indicate successful termination
24 } // end main
```

[Page 176]

```
Value of product after calculation: 25
Value of x after calculation: 6
```

```
Value of quotient after calculation: 0
Value of x after calculation: 6
```

4.7

a.

```
cin >> x;
```

b.

```
cin >> y;
```

c.

```
i = 1;
```

d.

```
power = 1;
```

e.

```
power *= x;
or
power = power * x;
```

f.

```
i++;
```

g.

```
if (i <= y )
```

h.

```
cout << power << endl;
```

4.8

See the following code:

```

1 // Exercise 4.8 Solution: ex04_08.cpp
2 // Raise x to the y power.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int main()
9 {
10    int x; // base
11    int y; // exponent
12    int i; // counts from 1 to y
13    int power; // used to calculate x raised to power y
14
15    i = 1; // initialize i to begin counting from 1
16    power = 1; // initialize power
17
18    cout << "Enter base as an integer: "; // prompt for base
19    cin >> x; // input base
20
21    cout << "Enter exponent as an integer: "; // prompt for exponent
22    cin >> y; // input exponent
23
24    // count from 1 to y and multiply power by x each time
25    while ( i <= y )
26    {
27        power *= x;
28        i++;
29    } // end while
30
31    cout << power << endl; // display result

```

```
32     return 0; // indicate successful termination
33 } // end main
```

[Page 177]

```
Enter base as an integer: 2
Enter exponent as an integer: 3
8
```

4.9

a.

Error: Missing the closing right brace of the `while` body.

Correction: Add closing right brace after the statement `c++;`.

b.

Error: Used stream insertion instead of stream extraction.

Correction: Change `<<` to `>>`.

c.

Error: Semicolon after `else` results in a logic error. The second output statement will always be executed.

Correction: Remove the semicolon after `else`.

4.10

The value of the variable `z` is never changed in the `while` statement. Therefore, if the loopcontinuation condition (`z >= 0`) is initially `TRUE`, an infinite loop is created. To prevent the infinite loop, `z` must be decremented so that it eventually becomes less than 0.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 178]

```
6 7 int main() 8 { 9 int y; // declare y 10 int x = 1; // initialize x 11 int total = 0; // initialize total 12 13
while ( x <= 10 ) // loop 10 times 14 { 15 y = x * x; // perform calculation 16 cout << y << endl; //
output result 17 total += y; // add y to total 18 x++; // increment counter x 19 } // end while 20 21 cout
<< "Total is " << total << endl; // display result 22 return 0; // indicate successful termination 23 } //
end main
```

For [Exercise 4.13](#) to [Exercise 4.16](#), perform each of these steps:

- a.** Read the problem statement.
- b.** Formulate the algorithm using pseudocode and top-down, stepwise refinement.
- c.** Write a C++ program.
- d.** Test, debug and execute the C++ program.

- 4.13** Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several tankfuls of gasoline by recording miles driven and gallons used for each tankful. Develop a C++ program that uses a while statement to input the miles driven and gallons used for each tankful. The program should calculate and display the miles per gallon obtained for each tankful and print the combined miles per gallon obtained for all tankfuls up to this point.

```
Enter the miles used (-1 to quit): 287
Enter gallons: 13
MPG this tankful: 22.076923
Total MPG: 22.076923

Enter the miles used (-1 to quit): 200
Enter gallons: 10
MPG this tankful: 20.000000
Total MPG: 21.173913

Enter the miles used (-1 to quit): 120
Enter gallons: 5
MPG this tankful: 24.000000
Total MPG: 21.678571
```

```
Enter miles (-1 to quit): -1
```

- 4.14** Develop a C++ program that will determine whether a department-store customer has exceeded the credit limit on a charge account. For each customer, the following facts are available:

a.

Account number (an integer)

[Page 179]

b.

Balance at the beginning of the month

c.

Total of all items charged by this customer this month

d.

Total of all credits applied to this customer's account this month

e.

Allowed credit limit

The program should use a `while` statement to input each of these facts, calculate the new balance (= beginning balance + charges credits) and determine whether the new balance exceeds the customer's credit limit. For those customers whose credit limit is exceeded, the program should display the customer's account number, credit limit, new balance and the message "Credit Limit Exceeded."

```
Enter account number (-1 to end): 100
Enter beginning balance: 5394.78
Enter total charges: 1000.00
Enter total credits: 500.00
Enter credit limit: 5500.00
New balance is 5894.78
Account: 100
Credit limit: 5500.00
Balance:      5894.78
Credit Limit Exceeded.

Enter Account Number (or -1 to quit): 200
Enter beginning balance: 1000.00
Enter total charges: 123.45
Enter total credits: 321.00
Enter credit limit: 1500.00
New balance is 802.45

Enter Account Number (or -1 to quit): 300
Enter beginning balance: 500.00
Enter total charges: 274.73
Enter total credits: 100.00
Enter credit limit: 800.00
New balance is 674.73

Enter Account Number (or -1 to quit): -1
```

- 4.15** One large chemical company pays its salespeople on a commission basis. The salespeople each receive \$200 per week plus 9 percent of their gross sales for that week. For example, a salesperson who sells \$5000 worth of chemicals in a week receives \$200 plus 9 percent of \$5000, or a total of \$650. Develop a C++ program that uses a while statement to input each salesperson's gross sales for last week and calculates and displays that salesperson's earnings. Process one salesperson's figures at a time.

```
Enter sales in dollars (-1 to end): 5000.00
Salary is: $650.00

Enter sales in dollars (-1 to end): 6000.00
Salary is: $740.00

Enter sales in dollars (-1 to end): 7000.00
Salary is: $830.00

Enter sales in dollars (-1 to end): -1
```

[Page 180]

- 4.16** Develop a C++ program that uses a while statement to determine the gross pay for each of several employees. The company pays "straight time" for the first 40 hours worked by each employee and pays "time-and-a-half" for all hours worked in excess of 40 hours. You are given a list of the employees of the company, the number of hours each employee worked last week and the hourly rate of each employee. Your program should input this information for each employee and should determine and display the employee's gross pay.

```
Enter hours worked (-1 to end): 39
Enter hourly rate of the worker ($00.00): 10.00
Salary is $390.00

Enter hours worked (-1 to end): 40
Enter hourly rate of the worker ($00.00): 10.00
Salary is $400.00

Enter hours worked (-1 to end): 41
Enter hourly rate of the worker ($00.00): 10.00
Salary is $415.00

Enter hours worked (-1 to end): -1
```

- 4.17** The process of finding the largest number (i.e., the maximum of a group of numbers) is used frequently in computer applications. For example, a program that determines the winner of a sales contest inputs the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write a pseudocode program, then a C++ program that uses a `while` statement to determine and print the largest number of 10 numbers input by the user. Your program should use three variables, as follows:

counter: A counter to count to 10 (i.e., to keep track of how many numbers have been input and to determine when all 10 numbers have been processed).

number: The current number input to the program.

largest: The largest number found so far.

- 4.18** Write a C++ program that uses a `while` statement and the tab escape sequence `\t` to print the following table of values:

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

- 4.19** Using an approach similar to that in [Exercise 4.17](#), find the two largest values among the 10 numbers. [Note: You must input each number only once.]

- 4.20** The examination-results program of [Fig. 4.16](#)[Fig. 4.18](#) assumes that any value input by the user that is not a 1 must be a 2. Modify the application to validate its inputs. On any input, if the value entered is other than 1 or 2, keep looping until the user enters a correct value.

4.21 What does the following program print?

```

1 // Exercise 4.21: ex04_21.cpp
2 // What does this program print?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int count = 1; // initialize count
10
11    while ( count <= 10 ) // loop 10 times
12    {
13        // output line of text
14        cout << ( count % 2 ? "*****" : "+++++++" ) << endl;
15        count++; // increment count
16    } // end while
17
18    return 0; // indicate successful termination
19 } // end main

```

4.22 What does the following program print?

```

1 // Exercise 4.22: ex04_22.cpp
2 // What does this program print?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int row = 10; // initialize row
10    int column; // declare column
11
12    while ( row >= 1 ) // loop until row < 1
13    {
14        column = 1; // set column to 1 as iteration begins
15
16        while ( column <= 10 ) // loop 10 times
17        {
18            cout << ( row % 2 ? "<" : ">" ); // output
19            column++; // increment column
20        } // end inner while
21

```

```

22         row--; // decrement row
23         cout << endl; // begin new output line
24     } // end outer while
25
26     return 0; // indicate successful termination
27 } // end main

```

[Page 182]

- 4.23** (Dangling-Else Problem) State the output for each of the following when x is 9 and y is 11 and when x is 11 and y is 9. Note that the compiler ignores the indentation in a C++ program. The C++ compiler always associates an `else` with the previous `if` unless told to do otherwise by the placement of braces `{ }`. On first glance, the programmer may not be sure which `if` and `else` match, so this is referred to as the "dangling-else" problem. We eliminated the indentation from the following code to make the problem more challenging. [Hint: Apply indentation conventions you have learned.]

a.

```

if ( x < 10 )
if ( y > 10 )
cout << "*****" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;

```

b.

```

if ( x < 10 )
{
if ( y > 10 )
cout << "*****" << endl;
}
else
{
cout << "#####" << endl;
cout << "$$$$$" << endl;
}

```

- 4.24** (Another Dangling-Else Problem) Modify the following code to produce the output shown. Use proper indentation techniques. You must not make any changes other than inserting braces. The compiler ignores indentation in a C++ program. We eliminated the indentation from the following code to make the problem more challenging. [Note: It is possible that no modification is necessary.]

```
if ( y == 8 )
if ( x == 5 )
cout << "@@@@@@" << endl;
else
cout << "#####" << endl;
cout << "$$$$$$" << endl;
cout << "&&&&&" << endl;
```

a.

Assuming $x = 5$ and $y = 8$, the following output is produced.



@@@@
\$\$\$\$\$
&&&&&

b.

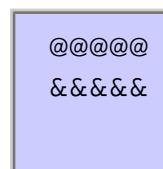
Assuming $x = 5$ and $y = 8$, the following output is produced.



@@@@
@

c.

Assuming $x = 5$ and $y = 8$, the following output is produced.



@@@@
&&&&&

d.

Assuming $x = 5$ and $y = 7$, the following output is produced. [Note: The last three output statements after the `else` are all part of a block.]

```
#####  
$$$$$  
&&&&&
```

- 4.25** Write a program that reads in the size of the side of a square and then prints a hollow square of that size out of asterisks and blanks. Your program should work for squares of all side sizes between 1 and 20. For example, if your program reads a size of 5, it should print

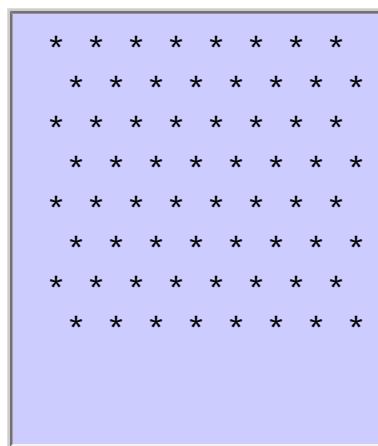
```
*****  
*   *  
*   *  
*   *  
*****
```

- 4.26** A palindrome is a number or a text phrase that reads the same backwards as forwards. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a program that reads in a five-digit integer and determines whether it is a palindrome. [Hint: Use the division and modulus operators to separate the number into its individual digits.]

- 4.27** Input an integer containing only 0s and 1s (i.e., a "binary" integer) and print its decimal equivalent. Use the modulus and division operators to pick off the "binary" number's digits one at a time from right to left. Much as in the decimal number system, where the rightmost digit has a positional value of 1, the next digit left has a positional value of 10, then 100, then 1000, and so on, in the binary number system the rightmost digit has a positional value of 1, the next digit left has a positional value of 2, then 4, then 8, and so on. Thus the decimal number 234 can be interpreted as $2 * 100 + 3 * 10 + 4 * 1$. The decimal equivalent of binary 1101 is $1 * 1 + 0 * 2 + 1 * 4 + 1 * 8$ or $1 + 0 + 4 + 8$, or 13. [Note: The reader not familiar with binary numbers might wish to refer to [Appendix D](#).]

- 4.28** Write a program that displays the checkerboard pattern shown below. Your program must use only three output statements, one of each of the following forms:

```
cout << " * ";
cout << ' ';
cout << endl;
```



- 4.29** Write a program that prints the powers of the integer 2, namely 2, 4, 8, 16, 32, 64, etc. Your `while` loop should not terminate (i.e., you should create an infinite loop). To do this, simply use the keyword `true` as the expression for the `while` statement. What happens when you run this program?

[Page 184]

- 4.30** Write a program that reads the radius of a circle (as a `double` value) and computes and prints the diameter, the circumference and the area. Use the value 3.14159 for π .

- 4.31** What is wrong with the following statement? Provide the correct statement to accomplish what the programmer was probably trying to do.

```
cout << ++( x + y );
```

- 4.32** Write a program that reads three nonzero double values and determines and prints whether they could represent the sides of a triangle.

- 4.33** Write a program that reads three nonzero integers and determines and prints whether they could be the sides of a right triangle.

- 4.34** (Cryptography) A company wants to transmit data over the telephone, but is concerned that its phones could be tapped. All of the data are transmitted as four-digit integers. The company has asked you to write a program that encrypts the data so that it can be transmitted more securely. Your program should read a four-digit integer and encrypt it as follows: Replace each digit by (the sum of that digit plus 7) modulus 10. Then, swap the first digit with the third, swap the second digit with the fourth and print the encrypted integer. Write a separate program that inputs an encrypted fourdigit integer and decrypts it to form the original number.

- 4.35** The factorial of a nonnegative integer n is written $n!$ (pronounced "n factorial") and is defined as follows:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1 \text{ (for values of } n \text{ greater than to 1)}$$

and

$$n! = 1 \text{ (for } n = 0 \text{ or } n = 1\text{).}$$

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is 120. Use while statements in each of the following:

a.

Write a program that reads a nonnegative integer and computes and prints its factorial.

b.

Write a program that estimates the value of the mathematical constant e by using the formula:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

Prompt the user for the desired accuracy of e (i.e., the number of terms in the summation).

c.

Write a program that computes the value of e^x by using the formula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Prompt the user for the desired accuracy of e (i.e., the number of terms in the summation).

- 4.36** [Note: This exercise corresponds to [Section 4.13](#), a portion of our software engineering case study.] Describe in 200 words or fewer what an automobile is and does. List the nouns and verbs separately. In the text, we stated that each noun might correspond to an object that will need to be built to implement a system, in this case a car. Pick five of the objects you listed, and, for each, list several attributes and several behaviors. Describe briefly how these objects interact with one another and other objects in your description. You have just performed several of the key steps in a typical object-oriented design.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 186]

Outline

[5.1 Introduction](#)

[5.2 Essentials of Counter-Controlled Repetition](#)

[5.3 for Repetition Statement](#)

[5.4 Examples Using the for Statement](#)

[5.5 do...while Repetition Statement](#)

[5.6 switch Multiple-Selection Statement](#)

[5.7 break and continue Statements](#)

[5.8 Logical Operators](#)

[5.9 Confusing Equality \(==\) and Assignment \(=\) Operators](#)

[5.10 Structured Programming Summary](#)

[5.11 \(Optional\) Software Engineering Case Study: Identifying Objects' States and Activities in the ATM System](#)

[5.12 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 186 (continued)]

5.1. Introduction

Chapter 4 began our introduction to the types of building blocks that are available for problem solving. We used those building blocks to employ proven program construction techniques. In this chapter, we continue our presentation of the theory and principles of structured programming by introducing C++'s remaining control statements. The control statements we study here and in Chapter 4 will help us in building and manipulating objects. We continue our early emphasis on object-oriented programming that began with a discussion of basic concepts in Chapter 1 and extensive object-oriented code examples and exercises in Chapters 34.

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 devote a portion of the chapter to expanding the `GradeBook` class presented in Chapters 34. 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 programmers 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. Finally, we summarize C++'s control statements and the proven problem-solving techniques presented in this chapter and Chapter 4.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 187]

- the **loop-continuation condition** that tests for the **final value** of the control variable (i.e., whether looping should continue)

•

- 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 at 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 will use the term "declaration" except when the distinction is important.

Figure 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).

[Page 188]

[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.

Good Programming Practice 5.1



Put a blank line before and after each control statement to make it stand out in the program.

Good Programming Practice 5.2



Too many levels of nesting can make a program difficult to understand. As a rule, try to avoid using more than three levels of indentation.

Good Programming Practice 5.3



Vertical spacing above and below control statements and indentation of the bodies of control statements within the control statement headers give programs a two-dimensional appearance that greatly improves readability.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

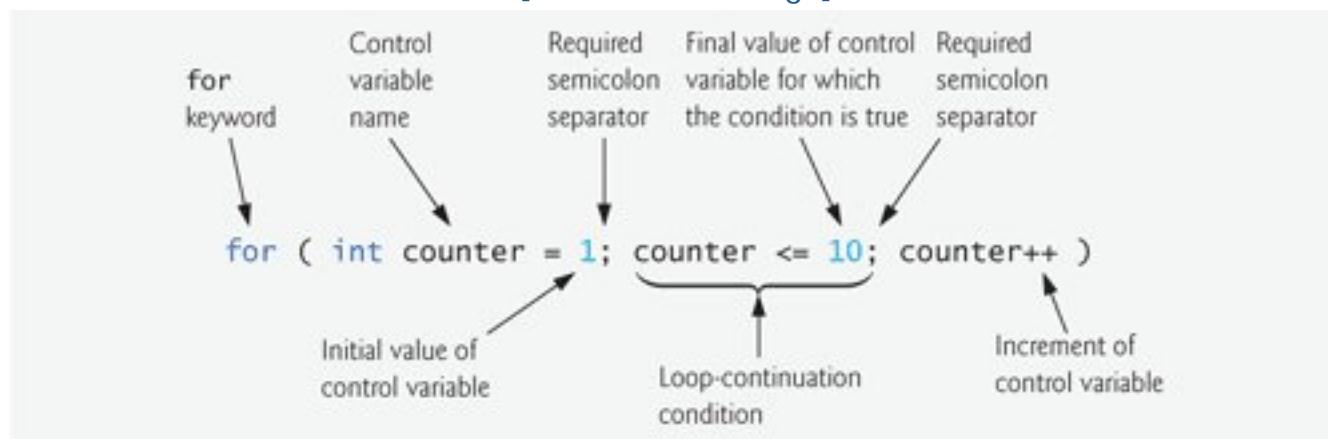
[Page 189]

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.

Figure 5.3. for statement header components.

[\[View full size image\]](#)



Notice that [Fig. 5.2](#) uses the loop-continuation condition `counter <= 10`. If the programmer 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.

[Page 190]

Good Programming Practice 5.4



Using the final value in the condition of a `while` or `for` statement and using the `<=` relational operator will help avoid off-by-one errors. For a loop used to print the values 1 to 10, for example, the loop-continuation condition should be `counter <= 10` rather than `counter < 10` (which is an off-by-one error) or `counter < 11` (which is nevertheless correct). Many programmers prefer so-called **zero-based counting**, in which, to count 10 times through the loop, `counter` would be initialized to zero and the loop-continuation test would be `counter < 10`.

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 will 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` statementthe 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 pre-standard 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.

[Page 191]

As we will 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 most often is used in `for` statements. Its primary application is to enable the programmer 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 5.5



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 stand-alone 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.

Software Engineering Observation 5.1



Placing a semicolon immediately after a `for` header is sometimes used to create a so-called **delay loop**. Such a `for` loop with an empty body still loops the indicated number of times, doing nothing other than the counting. For example, you might use a delay loop to slow down a program that is producing outputs on the screen too quickly for you to read them. Be careful though, because such a time delay will vary among systems with different processor speeds.

[Page 192]

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.

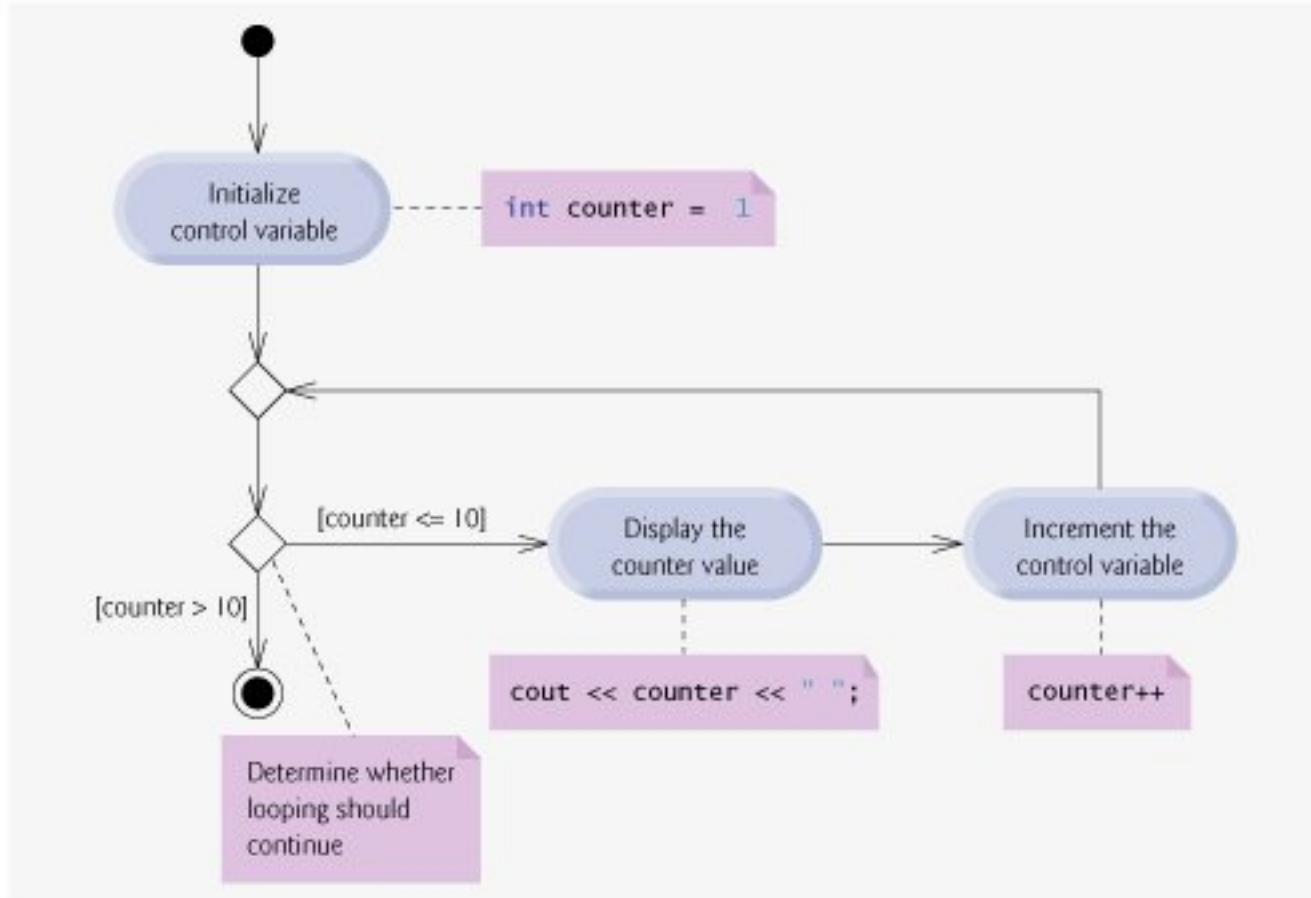
`for` Statement UML Activity Diagram

The `for` statement's UML activity diagram is similar to that of the `while` statement ([Fig. 4.6](#)). [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. Imagine, again, that the programmer has a bin of empty `for` statement UML activity diagrams as many as the programmer might need to stack and nest with the activity diagrams of other control statements to form a structured implementation of an algorithm. The programmer fills in the action states and decision symbols with action expressions and guard conditions appropriate to the algorithm.

[Page 193]

Figure 5.4. UML activity diagram for the `for` statement in [Fig. 5.2](#).

(This item is displayed on page 192 in the print version)

[\[View full size image\]](#)

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 194]

Figure 5.5. Summing integers with the for statement.

```

1 // Fig. 5.5: fig05_05.cpp
2 // Summing integers with the for statement.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int total = 0; // initialize total
10
11    // total even integers from 2 through 20
12    for ( int number = 2; number <= 20; number += 2 )
13        total += number;
14
15    cout << "Sum is " << total << endl; // display results
16    return 0; // successful termination
17 } // 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 5.6



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 5.7



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.

[Page 195]

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.

Figure 5.6. Compound interest calculations with `for`.

(This item is displayed on pages 195 - 196 in the print version)

```
1 // Fig. 5.6: fig05_06.cpp
2 // Compound interest calculations with for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setw; // enables program to set a field width
10 using std::setprecision;
11
12 #include <cmath> // standard C++ math library
13 using std::pow; // enables program to use function pow
14
15 int main()
16 {
17     double amount; // amount on deposit at end of each year
18     double principal = 1000.0; // initial amount before interest
19     double rate = .05; // interest rate
20
21     // display headers
22     cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;
23
24     // set floating-point number format
25     cout << fixed << setprecision( 2 );
26
27     // calculate amount on deposit for each of ten years
28     for ( int year = 1; year <= 10; year++ )
29     {
30         // calculate new amount for specified year
31         amount = principal * pow( 1.0 + rate, year );
32
33         // display the year and the amount
34         cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
35     } // end for
36
37     return 0; // indicate successful termination
38 } // 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 2835) 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 `yth` 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](#) provides a summary of other math library functions.

[Page 196]

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 `double` for Monetary Amounts

Notice that lines 1719 declare the variables `amount`, `principal` and `rate` to be of type `double`. We have done this for simplicity because we are 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 5.8



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. In the Exercises, we explore the use of integers to perform monetary calculations. [Note: Some third-party vendors sell C++ class libraries that perform precise monetary calculations. We include several URLs in [Appendix I](#)].

Using Stream Manipulators to Format Numeric Output

The output statement at line 25 before the `for` loop and the output statement at 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`.

[Page 197]

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**. Thus, 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 detail in [Chapter 15](#).

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.

For fun, be sure to try our Peter Minuit problem in [Exercise 5.29](#). This problem demonstrates the wonders of compound interest.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 198]

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. The last line `while(condition);` might be misinterpreted by the reader as a `while` statement containing as its body an empty statement. Thus, the `do...while` with one statement is often written as follows to avoid confusion:

```
do
{
    statement
} while ( condition );
```

Good Programming Practice 5.9



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 110. 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).

Figure 5.7. `do...while` repetition statement.

(This item is displayed on pages 198 - 199 in the print version)

```

1 // Fig. 5.7: fig05_07.cpp
2 // do...while repetition statement.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int counter = 1; // initialize counter
10
11    do
12    {
13        cout << counter << " "; // display counter
14        counter++; // increment counter
15    } while ( counter <= 10 ); // end do...while
16
17    cout << endl; // output a newline
18    return 0; // indicate successful termination
19 } // end main

```

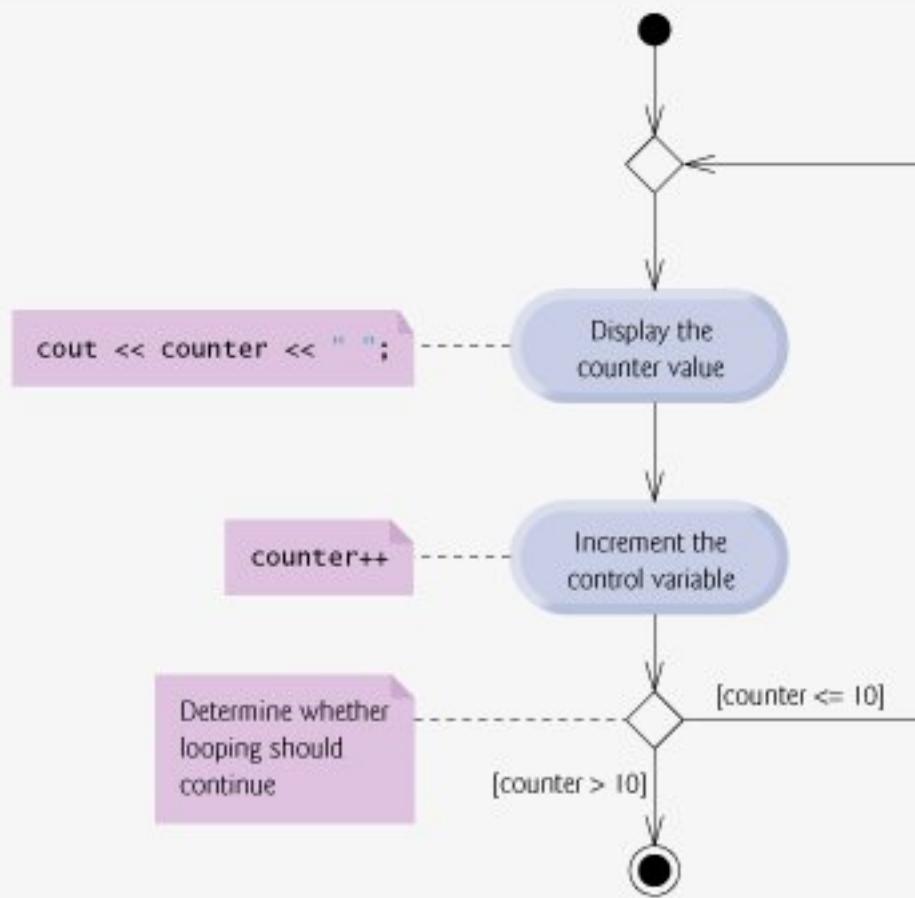
1 2 3 4 5 6 7 8 9 10

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. 4.6). 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 the programmer has access to a bin of empty do...while statement UML activity diagrams as many as the programmer might need to stack and nest with the activity diagrams of other control statements to form a structured implementation of an algorithm. The programmer fills in the action states and decision symbols with action expressions and guard conditions appropriate to the algorithm.

[Page 199]

Figure 5.8. UML activity diagram for the do...while repetition statement of Fig. 5.7.

[\[View full size image\]](#)[◀ PREV](#)[NEXT ▶](#)

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 200]

GradeBook Class with switch Statement to Count A, B, C, D and F Grades

In the next example, we 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.

Figure 5.9. GradeBook class definition.

```

1 // Fig. 5.9: GradeBook.h
2 // Definition of class GradeBook that counts A, B, C, D and F grades.
3 // Member functions are defined in GradeBook.cpp
4
5 #include <string> // program uses C++ standard string class
6 using std::string;
7
8 // GradeBook class definition
9 class GradeBook
10 {
11 public:
12     GradeBook( string ); // constructor initializes course name
13     void setCourseName( string ); // function to set the course name
14     string getCourseName(); // function to retrieve the course name
15     void displayMessage(); // display a welcome message
16     void inputGrades(); // input arbitrary number of grades from user
17     void displayGradeReport(); // display a report based on the grades
18 private:
19     string courseName; // course name for this GradeBook
20     int aCount; // count of A grades
21     int bCount; // count of B grades
22     int cCount; // count of C grades
23     int dCount; // count of D grades
24     int fCount; // count of F grades
25 };// end class GradeBook

```

Figure 5.10. GradeBook class uses switch statement to count letter grades A, B, C, D and F.

(This item is displayed on pages 201 - 203 in the print version)

```

1 // Fig. 5.10: GradeBook.cpp
2 // Member-function definitions for class GradeBook that
3 // uses a switch statement to count A, B, C, D and F grades.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "GradeBook.h" // include definition of class GradeBook
10
11 // constructor initializes courseName with string supplied as argument;
12 // initializes counter data members to 0
13 GradeBook::GradeBook( string name )
14 {
15     setCourseName( name ); // validate and store courseName
16     aCount = 0; // initialize count of A grades to 0
17     bCount = 0; // initialize count of B grades to 0
18     cCount = 0; // initialize count of C grades to 0
19     dCount = 0; // initialize count of D grades to 0
20     fCount = 0; // initialize count of F grades to 0
21 } // end GradeBook constructor
22
23 // function to set the course name; limits name to 25 or fewer characters
24 void GradeBook::setCourseName( string name )
25 {
26     if ( name.length() <= 25 ) // if name has 25 or fewer characters
27         courseName = name; // store the course name in the object
28     else // if name is longer than 25 characters
29     { // set courseName to first 25 characters of parameter name
30         courseName = name.substr( 0, 25 ); // select first 25 characters
31         cout << "Name " << name << "\ exceeds maximum length (25).\n"
32             << "Limiting courseName to first 25 characters.\n" << endl;
33     } // end if...else
34 } // end function setCourseName
35
36 // function to retrieve the course name
37 string GradeBook::getCourseName( )
38 {
39     return courseName;
40 } // end function getCourseName
41
42 // display a welcome message to the GradeBook user
43 void GradeBook::displayMessage( )
44 {

```

```
45     // this statement calls getCourseName to get the
46     // name of the course this GradeBook represents
47     cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
48         << endl;
49 } // end function displayMessage
50
51 // input arbitrary number of grades from user; update grade counter
52 void GradeBook::inputGrades()
53 {
54     int grade; // grade entered by user
55
56     cout << "Enter the letter grades." << endl
57         << "Enter the EOF character to end input." << endl;
58
59 // loop until user types end-of-file key sequence
60 while ( ( grade = cin.get() ) != EOF )
61 {
62     // determine which grade was entered
63     switch ( grade ) // switch statement nested in while
64     {
65         case 'A': // grade was uppercase A
66             case 'a': // or lowercase a
67                 aCount++; // increment aCount
68                 break; // necessary to exit switch
69
70         case 'B': // grade was uppercase B
71             case 'b': // or lowercase b
72                 bCount++; // increment bCount
73                 break; // exit switch
74
75         case 'C': // grade was uppercase C
76             case 'c': // or lowercase c
77                 cCount++; // increment cCount
78                 break; // exit switch
79
80         case 'D': // grade was uppercase D
81             case 'd': // or lowercase d
82                 dCount++; // increment dCount
83                 break; // exit switch
84
85         case 'F': // grade was uppercase F
86             case 'f': // or lowercase f
87                 fCount++; // increment fCount
88                 break; // exit switch
89
90         case '\n': // ignore newlines,
91         case '\t': // tabs,
92         case ' ': // and spaces in input
93                 break; // exit switch
```

```

94         default: // catch all other characters
95             cout << "Incorrect letter grade entered."
96                 << " Enter a new grade." << endl;
97             break; // optional; will exit switch anyway
98     } // end switch
99 }
100 } // end while
101 } // end function inputGrades
102
103 // display a report based on the grades entered by user
104 void GradeBook::displayGradeReport()
105 {
106     // output summary of results
107     cout << "\n\nNumber of students who received each letter grade:"
108         << "\nA: " << aCount // display number of A grades
109         << "\nB: " << bCount // display number of B grades
110         << "\nC: " << cCount // display number of C grades
111         << "\nD: " << dCount // display number of D grades
112         << "\nF: " << fCount // display number of F grades
113         << endl;
114 } // end function displayGradeReport

```

Figure 5.11. Creating a GradeBook object and calling its member functions.

(This item is displayed on pages 206 - 207 in the print version)

```

1 // Fig. 5.11: fig05_11.cpp
2 // Create GradeBook object, input grades and display grade report.
3
4 #include "GradeBook.h" // include definition of class GradeBook
5
6 int main()
7 {
8     // create GradeBook object
9     GradeBook myGradeBook( "CS101 C++ Programming" );
10
11     myGradeBook.displayMessage(); // display welcome message
12     myGradeBook.inputGrades(); // read grades from user
13     myGradeBook.displayGradeReport(); // display report based on grades
14     return 0; // indicate successful termination
15 } // 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 will 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.

[Page 203]

Reading Character Input

The user enters letter grades for a course in member function `inputGrades` (lines 52101). Inside the while header, at 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 they are represented as 1-byte integers in the computer. 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](#).

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 statements have values can be useful for assigning the same value to several variables. For example,

```
a = b = c = 0;
```

[Page 204]

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 60100) terminates. We have chosen to represent the characters entered into this program as `ints`, because `EOF` has an integer value.

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 is 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 6399), 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`' at 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 60100), so control flows to the `while`'s condition (line 60) to determine whether the loop should continue executing.

[Page 205]

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 at lines 6566 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 at lines 6768 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. (This feature is perfect for writing a concise program that displays the iterative song "The Twelve Days of Christmas" in [Exercise 5.28](#).)

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 writing `case 3:` simply creates an unused label. We will say more about this in [Appendix E, C Legacy Code Topics](#). 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 9598) 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 white-space characters shortly). If no match occurs, the `default` case executes, and lines 9697 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 5.10



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 the programmer 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 5.11



In a `switch` statement that lists the `default` clause last, the `default` clause does not require a `break` statement. Some programmers include this `break` for clarity and for symmetry with other cases.

[Page 206]

Ignoring Newline, Tab and Blank Characters in Input

Note that lines 9093 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 white-space 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 object's `displayMessage` member function to output a welcome message to the user. Line 12 invokes the object's `inputGrades` member function to read a set of grades from the user and keep track of the number of students who received each grade. Note that the input/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 104114 of Fig. 5.10), which outputs a report based on the grades entered (as in the output in Fig. 5.11).

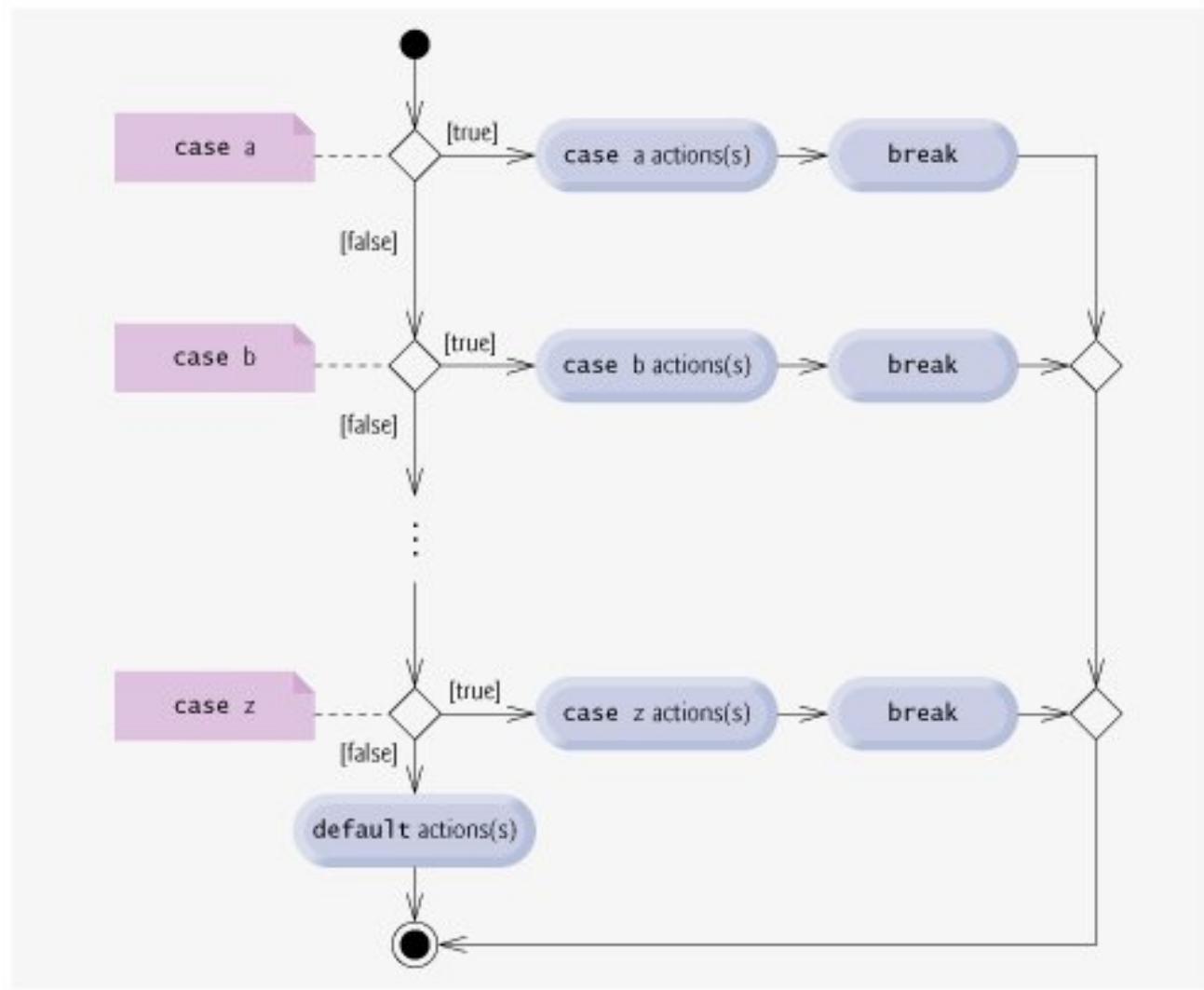
switch Statement UML Activity Diagram

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.

Figure 5.12. switch multiple-selection statement UML activity diagram with break statements.

(This item is displayed on page 208 in the print version)

[View full size image]



[Page 207]

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 the programmer has a bin of empty `switch` statement UML activity diagrams as many as the programmer might need to stack and nest with the activity diagrams of other control statements to form a structured implementation of an algorithm. The programmer fills 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 it can be used only for testing 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 an expression including variables (e.g., `a + b`) in a `switch` statement's `case` label is a syntax error.

[Page 208]

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 will 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, `ints` 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.

[Page 209]

Portability Tip 5.4



Because ints 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.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 210]

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 value of the control variable 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 body of the loop 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 `if` (lines 1112) 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.

Figure 5.14. continue statement terminating a single iteration of a for statement.

(This item is displayed on pages 210 - 211 in the print version)

```

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
```

[Page 211]

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 5.12



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 learn, 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 5.2



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.

[◀ PREV](#)

page footer

[NEXT ▶](#)

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 212]

```
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.

Figure 5.15. `&&` (logical AND) operator truth table.

expression1	expression2	expression1 <code>&&</code> 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 )
```

[Page 213]

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 (`||`).

Figure 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, the programmer 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 (!).

Figure 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. However, we use `stream manipulator boolalpha` in line 11 to specify that the value of each `bool` expression should be displayed either as 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 1115 produce the truth table for `&&`. Lines 1822 produce the truth table for `||`. Lines 2527 produce the truth table for `!`.

[Page 215]

Figure 5.18. Logical operators.

(This item is displayed on pages 214 - 215 in the print version)

```

1 // Fig. 5.18: fig05_18.cpp
2 // Logical operators.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::boolalpha; // causes bool values to print as "true" or "false"
7
8 int main()
9 {
10     // create truth table for && (logical AND) operator
11     cout << boolalpha << "Logical AND (&&)"
12         << "\nfalse && false: " << ( false && false )
13         << "\nfalse && true: " << ( false && true )
14         << "\ntrue && false: " << ( true && false )
15         << "\ntrue && true: " << ( true && true ) << "\n\n";
16
17     // create truth table for || (logical OR) operator
18     cout << "Logical OR (||)"
19         << "\nfalse || false: " << ( false || false )
20         << "\nfalse || true: " << ( false || true )
21         << "\ntrue || false: " << ( true || false )
22         << "\ntrue || true: " << ( true || true ) << "\n\n";
23
24     // create truth table for ! (logical negation) operator
25     cout << "Logical NOT (!)"
26         << "\n!false: " << ( !false )
27         << "\n!true: " << ( !true ) << endl;
28     return 0; // indicate successful termination
29 } // 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.

Figure 5.19. Operator precedence and associativity.

Operators						Associativity	Type
()						left to right	parentheses
++	--	<code>static_cast< type >()</code>				left to right	unary (postfix)
++	--	+	-	!		right to left	unary (prefix)
*	/	%				left to right	multiplicative
+	-					left to right	additive
<<	>>					left to right	insertion/extraction
<	<code><=</code>	>	<code>>=</code>			left to right	relational
<code>==</code>	<code>!=</code>					left to right	equality
<code>&&</code>						left to right	logical AND
<code> </code>						left to right	logical OR
<code>? :</code>						right to left	conditional
=	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	right to left	assignment
,						left to right	comma

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 217]

```
x = 1;
```

but instead writes

```
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.

[PREV](#)

page footer

[NEXT](#)

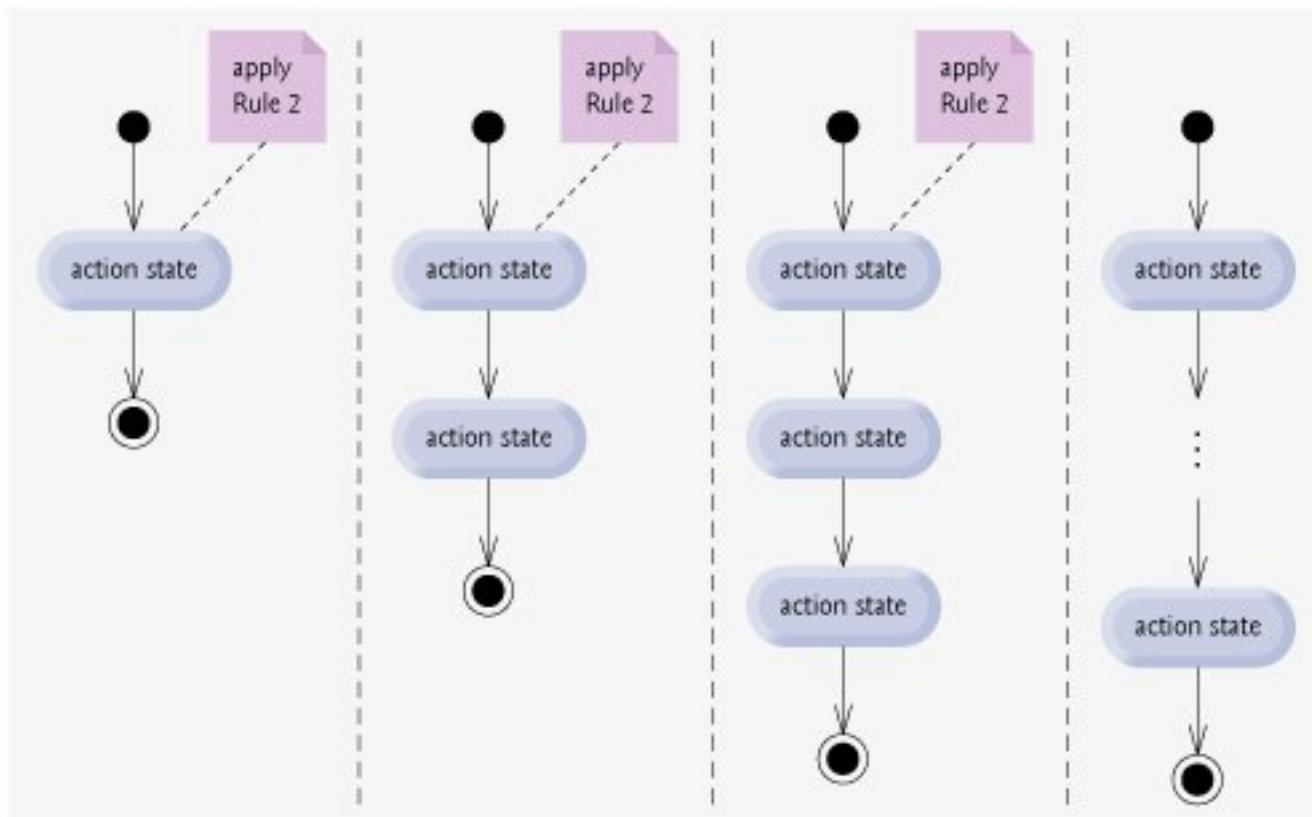
The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 219]

Figure 5.23. Repeatedly applying Rule 2 of Fig. 5.21 to the simplest activity diagram.

[\[View full size image\]](#)

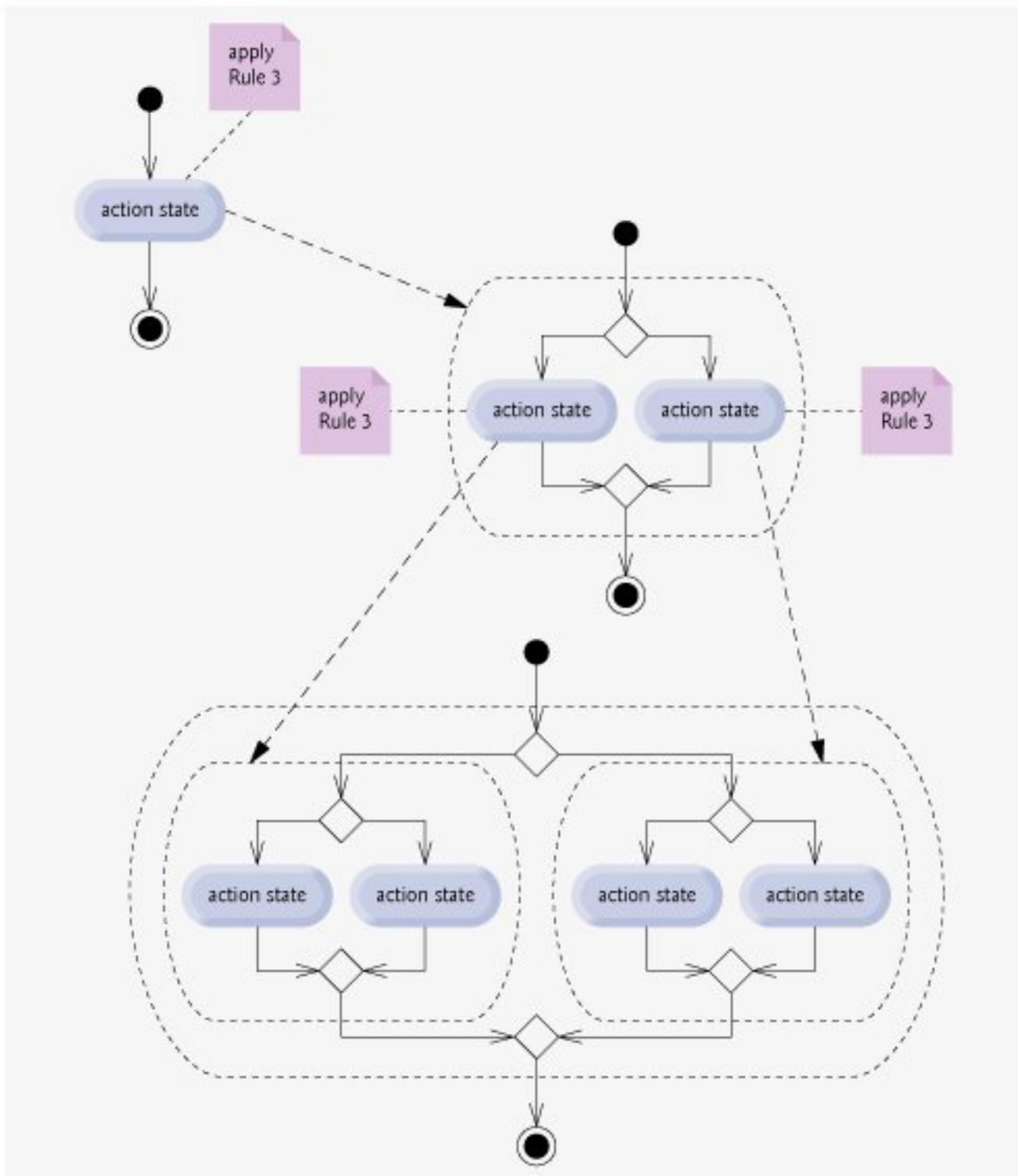


[Page 220]

Rule 3 is called the **nesting rule**. Repeatedly applying Rule 3 to the simplest activity diagram results in an activity diagram with neatly nested control statements. For example, in Fig. 5.24, the action state in the simplest activity diagram is replaced with a double-selection (`if...else`) statement. Then Rule 3 is applied again to the action states in the double-selection statement, replacing each of these action states with a double-selection statement. The dashed action-state symbols around each of the double-selection statements represent an action state that was replaced in the preceding activity diagram. [Note: The dashed arrows and dashed action state symbols shown in Fig. 5.24 are not part of the UML. They are used here as pedagogic devices to illustrate that any action state may be replaced with a control statement.]

Figure 5.24. Applying Rule 3 of Fig. 5.21 to the simplest activity diagram several times.

(This item is displayed on page 220 in the print version)

[\[View full size image\]](#)

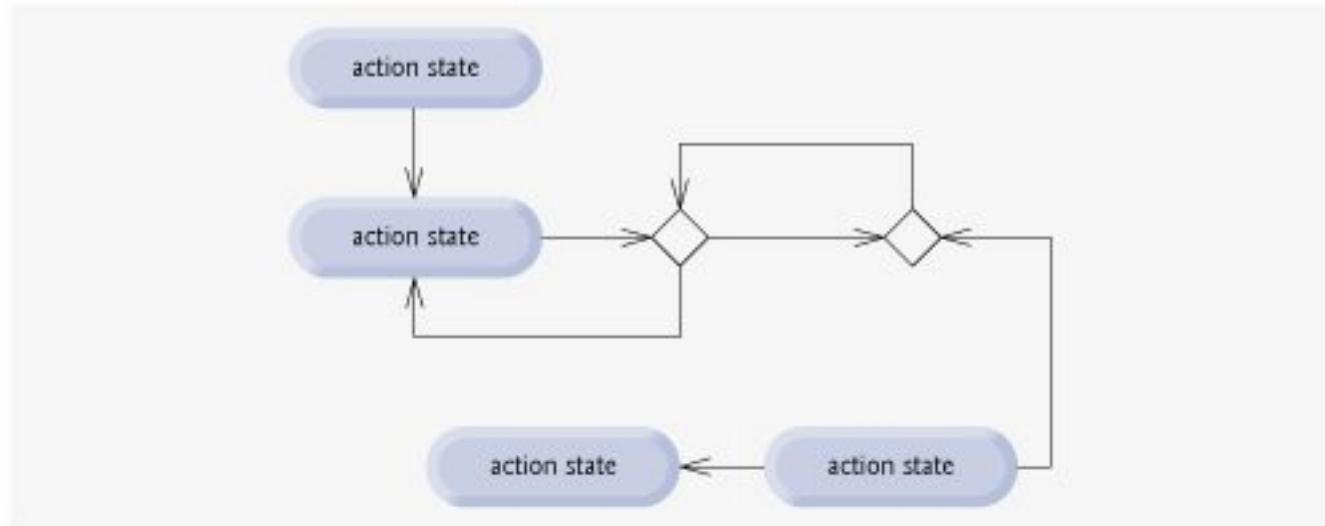
Rule 4 generates larger, more involved and more deeply nested statements. The diagrams that emerge from applying the rules in Fig. 5.21 constitute the set of all possible activity diagrams and hence the set of all possible structured programs. The beauty of the structured approach is that we use only seven simple single-entry/single-exit control statements and assemble them in only two simple ways.

If the rules in Fig. 5.21 are followed, an activity diagram with illegal syntax (such as that in Fig. 5.25) cannot be created. If you are uncertain about whether a particular diagram is legal, apply the rules of Fig. 5.21 in reverse to reduce the diagram to the simplest activity diagram. If it is reducible to the simplest activity diagram, the original diagram is structured; otherwise, it is not.

Figure 5.25. Activity diagram with illegal syntax.

(This item is displayed on page 222 in the print version)

[View full size image]



Structured programming promotes simplicity. Böhm and Jacopini have given us the result that only three forms of control are needed:

- Sequence
- Selection
- Repetition

The sequence structure is trivial. Simply list the statements to execute in the order in which they should execute.

Selection is implemented in one of three ways:

- `if` statement (single selection)
- `if...else` statement (double selection)
- `switch` statement (multiple selection)

It is straightforward to prove that the simple `if` statement is sufficient to provide any form of selection; everything that can be done with the `if...else` statement and the `switch` statement can be implemented (although perhaps not as clearly and efficiently) by combining `if` statements.

Repetition is implemented in one of three ways:

- `while` statement
- `do...while` statement
- `for` statement

It is straightforward to prove that the `while` statement is sufficient to provide any form of repetition. Everything that can be done with the `do...while` statement and the `for` statement can be done (although perhaps not as smoothly) with the `while` statement.

Combining these results illustrates that any form of control ever needed in a C++ program can be expressed in terms of the following:

- sequence
- `if` statement (selection)
- `while` statement (repetition)

and that these control statements can be combined in only two ways: stacking and nesting. Indeed, structured programming promotes simplicity.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 223]

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 document 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 5.3



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

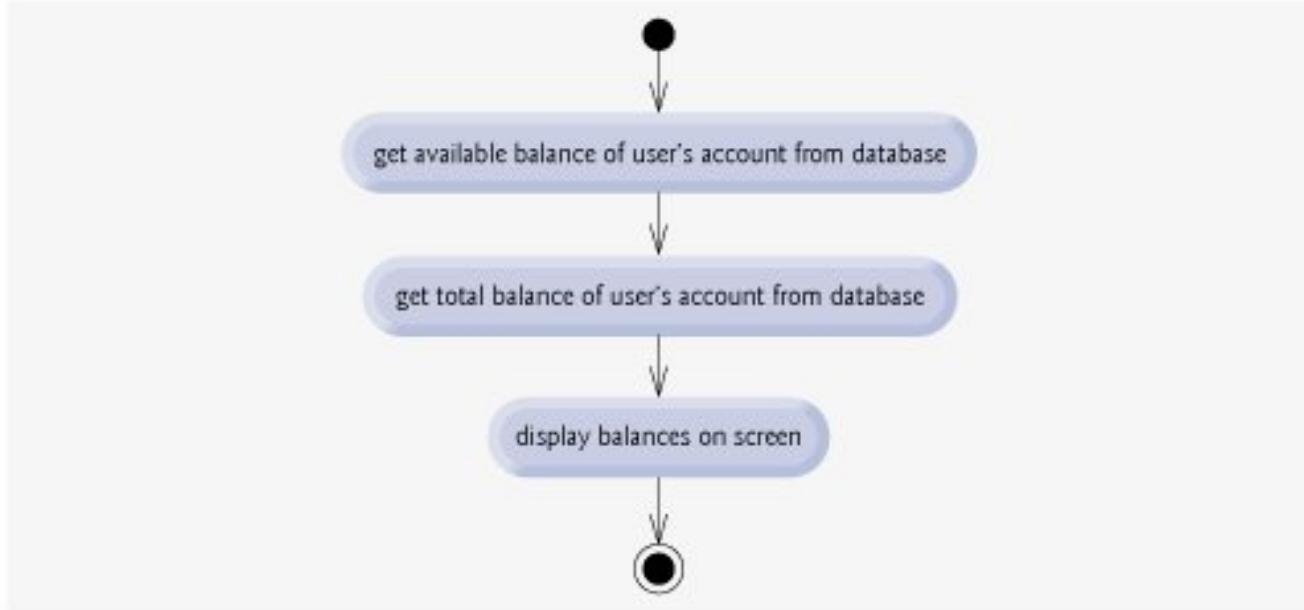
Like a state diagram, an activity diagram models aspects of system behavior. Unlike a state diagram, an activity diagram models an object's workflow (sequence of events) during program execution. An activity diagram models the actions the object will perform and in what order. Recall that we used UML activity diagrams to illustrate the flow of control for the control statements presented in [Chapter 4](#) and [Chapter 5](#).

The activity diagram in [Fig. 5.27](#) 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.

[Page 224]

Figure 5.27. Activity diagram for a BalanceInquiry transaction.

(This item is displayed on page 223 in the print version)

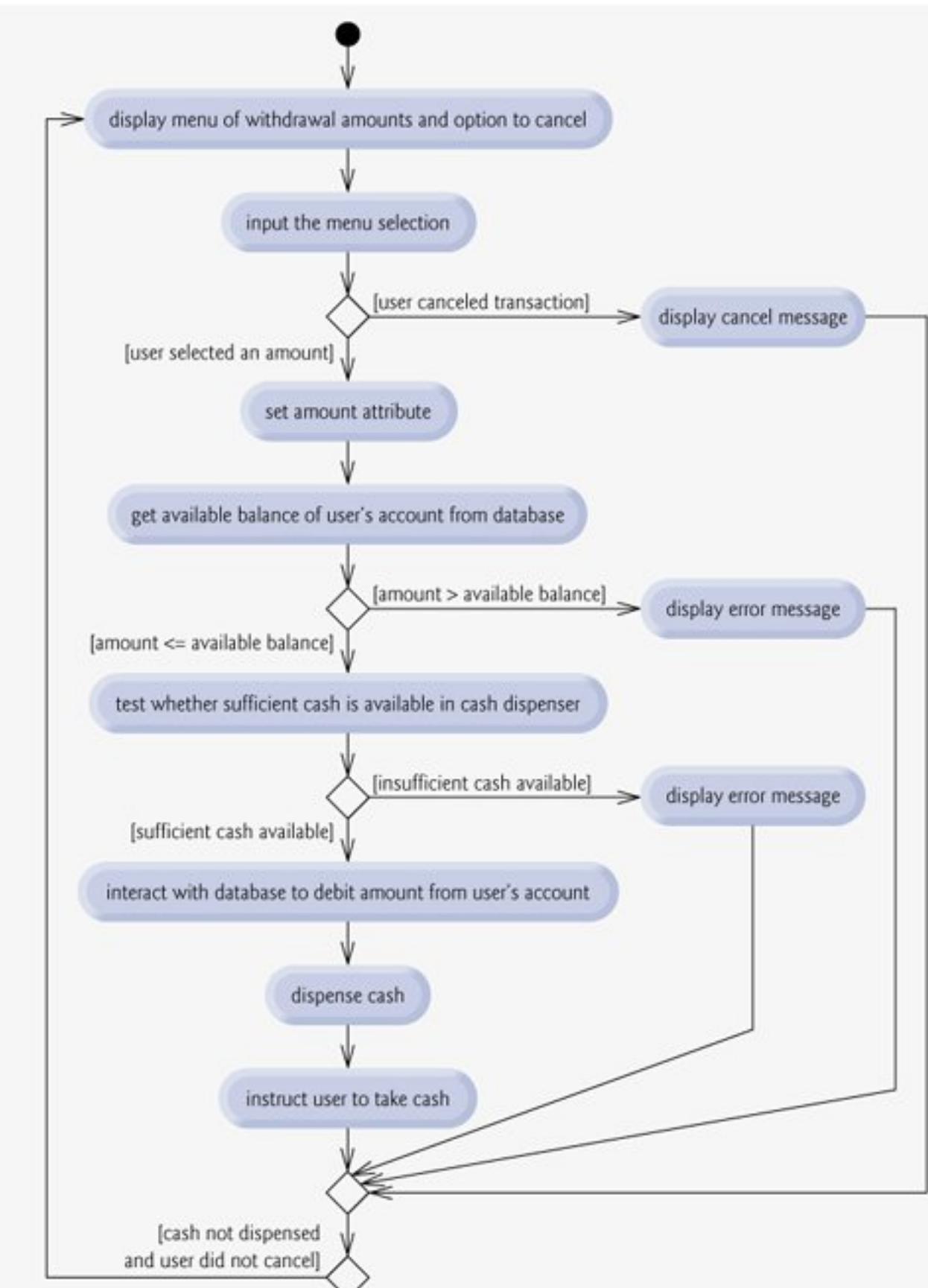
[\[View full size image\]](#)

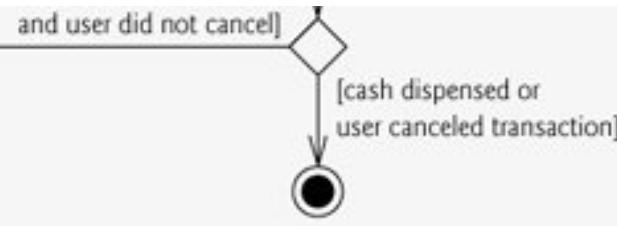
The UML represents an action in an activity diagram as an action state modeled by a rectangle with its left and right sides replaced by arcs curving outward. Each action state contains an action expression for example, "get available balance of user's account from database" that specifies an action to 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. 5.27) 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. 5.27) represents the final state—the end of the workflow after the object performs the modeled actions.

Figure 5.28 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. 2.17) 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.

Figure 5.28. Activity diagram for a withdrawal TTranscation.

(This item is displayed on page 225 in the print version)

[\[View full size image\]](#)



If the user selects a withdrawal amount from the menu, the transaction sets `amount` (an attribute of class `Withdrawal` originally modeled in Fig. 4.24) 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.

[Page 226]

We have 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 document, 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.

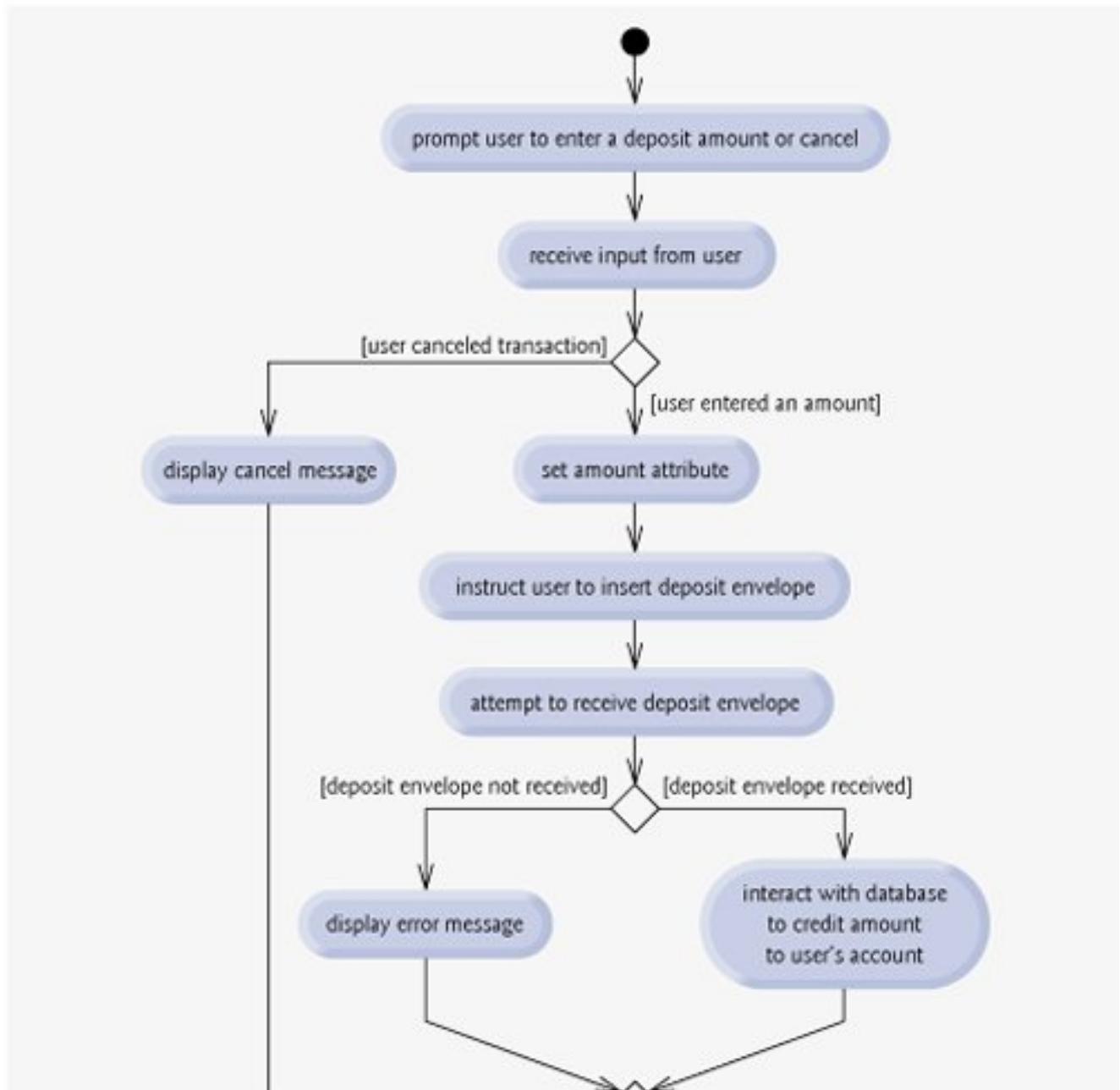
5.2 a.

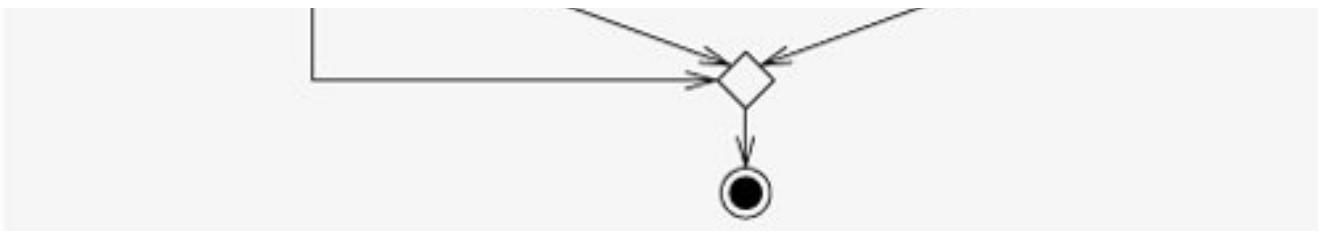
- 5.3** Figure 5.29 presents an activity diagram for a deposit transaction. The diagram models the actions that occur after the user chooses the deposit option from the main menu and before the ATM returns the user to the main menu. Recall that part of receiving a deposit amount from the user involves 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 clearthis occurs independently of the ATM system.

Figure 5.29. Activity diagram for a Deposit Transaction.

(This item is displayed on page 227 in the print version)

[View full size image]





PREV

NEXT

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 227]

In [Chapter 3](#), we introduced C++ programming with the basic concepts of objects, classes and member functions. [Chapter 4](#) and this chapter provided a thorough introduction to the types of control statements that programmers typically use to specify program logic in functions. In [Chapter 6](#), we examine functions in greater depth.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 229]

- The end-of-file indicator is a system-dependent keystroke combination that specifies that there is no more data to input. `EOF` is a symbolic integer constant defined in the `<iostream>` header file that indicates "end-of-file."
- The expression in the parentheses following keyword `switch` is called the controlling expression of the `switch`. The `switch` statement compares the value of the controlling expression with each `case` label.
- Listing `cases` consecutively with no statements between them enables the `cases` to perform the same set of statements.
- 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`.
- The `switch` statement can be used only for testing a constant integral expression. 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.
- C++ provides several data types to represent integers `int`, `char`, `short` and `long`. The range of integer values for each type depends on the particular computer's hardware.
- The `break` statement, when executed in one of the repetition statements (`for`, `while` and `do...while`), causes immediate exit from the statement.
- The `continue` statement, when executed in one of the repetition statements (`for`, `while` and `do...while`), skips any remaining statements in the body of the repetition statement and proceeds with the next iteration of the loop. In a `while` or `do...while` statement, execution continues with the next evaluation of the condition. In a `for` statement, execution continues with the increment expression in the `for` statement header.
- Logical operators enable programmers to form complex conditions by combining simple conditions. The logical operators are `&&` (logical AND), `||` (logical OR) and `!` (logical NOT, also called logical negation).
- The `&&` (logical AND) operator ensures that two conditions are both `TRUE` before choosing a certain path of execution.
- The `||` (logical OR) operator ensures that either or both of two conditions are `true` before choosing a certain path of execution.
- An expression containing `&&` or `||` operators evaluates only until the truth or falsehood of the expression is known. This performance feature for the evaluation of logical AND and logical OR expressions is called short-circuit evaluation.
- The `!` (logical NOT, also called logical negation) operator enables a programmer to "reverse" the meaning of a condition. The unary logical negation operator is placed before a condition to choose a path of execution if the original condition (without the logical negation operator) is `false`. In most cases, the programmer can avoid using logical negation by expressing the condition with an appropriate relational or equality operator.
- When used as a condition, any nonzero value implicitly converts to `true`; 0 (zero) implicitly converts to `false`.
- By default, `bool` values `TRUE` and `false` are displayed by `cout` as 1 and 0, respectively. Stream manipulator `boolalpha` specifies that the value of each `bool` expression should be displayed as either the word "true" or the word "false."
- Any form of control ever needed in a C++ program can be expressed in terms of sequence, selection

and repetition statements, and these can be combined in only two waysstacking and nesting.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 230]

Terminology

`!`, logical NOT operator

`&&`, logical AND operator

`||`, logical OR operator

ASCII character set

`boolalpha` stream manipulator

`break` statement

`case` label

`char` fundamental type

comma operator

constant integral expression

`continue` statement

controlling expression of a `switch`

decrement a control variable

`default case in switch`

definition

delay loop

field width

final value of a control variable

for repetition statement

for header

increment a control variable

initial value of a control variable

left justification

left stream manipulator

logical AND (**&&**)

logical negation (**!**)

logical NOT (**!**)

logical operator

logical OR (**| |**)

loop-continuation condition

lvalue ("left value")

name of a control variable

nesting rule

off-by-one error

right justification

right stream manipulator

rvalue ("right value")

scope of a variable

setw stream manipulator

short-circuit evaluation

simple condition

stacking rule

standard library function pow

sticky setting

switch multiple-selection statement

truth table

zero-based counting

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 231]

5.3

Find the error(s) in each of the following code segments and explain how to correct it (them).

a.

```
x = 1;
while ( x <= 10 );
    x++;
}
```

b.

```
for ( y = .1; y != 1.0; y += .1 )
    cout << y << endl;
```

c.

```
switch ( n )
{
    case 1:
        cout << "The number is 1" << endl;
    case 2:
        cout << "The number is 2" << endl;
        break;
    default:
        cout << "The number is not 1 or 2" << endl;
        break;
}
```

d.

The following code should print the values 1 to 10.

```
n = 1;
while ( n < 10 )
    cout << n++ << endl;
```

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 232]

•

```
x = 1;
```

```
while ( x <= 20 )
{
    cout << x;

    if ( x % 5 == 0 )
        cout << endl;
    else
        cout << '\t';

    x++;
}
```

•

```
for ( x = 1; x <= 20; x++ )
{
    cout << x;

    if ( x % 5 == 0 )
        cout << endl;
    else
        cout << '\t';
}
```

or

```
for ( x = 1; x <= 20; x++ )
{
    if ( x % 5 == 0 )
        cout << x << endl;
    else
        cout << x << '\t';
}
```

5.3

a.

Error: The semicolon after the `while` header causes an infinite loop.

Correction: Replace the semicolon by a `{`, or remove both the `;` and the `}`.

b.

Error: Using a floating-point number to control a `for` repetition statement.

Correction: Use an integer and perform the proper calculation in order to get the values you desire.

```
for ( y = 1; y != 10; y++ )
    cout << ( static_cast< double >( y ) / 10 ) << endl;
```

c.

Error: Missing `break` statement in the first case.

Correction: Add a `break` statement at the end of the statements for the first case. Note that this is not an error if the programmer wants the statement of `case 2:` to execute every time the `case 1:` statement executes.

d.

Error: Improper relational operator used in the while repetition-continuation condition.

Correction: Use `<=` rather than `<`, or change 10 to 11.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 234]

```
13 // prompt user for input 14 cout << "Enter two integers in the range 1-20: "; 15 cin >> x >> y; //  
read values for x and y 16 17 for ( int i = 1; i <= y; i++ ) // count from 1 to y 18 { 19 for ( int j = 1; j  
<= x; j++ ) // count from 1 to x 20 cout << '@'; // output @ 21 22 cout << endl; // begin new line  
23 } // end outer for 24 25 return 0; // indicate successful termination 26 } // end main
```

5.8

Write a program that uses a `for` statement to find the smallest of several integers. Assume that the first value read specifies the number of values remaining and that the first number is not one of the integers to compare.

5.9

Write a program that uses a `for` statement to calculate and print the product of the odd integers from 1 to 15.

5.10

The factorial function is used frequently in probability problems. Using the definition of factorial in [Exercise 4.35](#), write a program that uses a `for` statement to evaluate the factorials of the integers from 1 to 5. Print the results in tabular format. What difficulty might prevent you from calculating the factorial of 20?

5.11

Modify the compound interest program of [Section 5.4](#) to repeat its steps for the interest rates 5 percent, 6 percent, 7 percent, 8 percent, 9 percent and 10 percent. Use a `for` statement to vary the interest rate.

5.12

Write a program that uses `for` statements to print the following patterns separately, one below the other. Use `for` loops to generate the patterns. All asterisks (*) should be printed by a single statement of the form `cout << '*' ;` (this causes the asterisks to print side by side). [Hint: The last two patterns require that each line begin with an appropriate number of blanks. Extra credit: Combine your code from the four separate problems into a single program that prints all four patterns side by side by making clever use of nested `for` loops.]

(a)

```
*
```

```
**
```

(b)

```
*****
```

```
*****
```

(c)

```
*****
```

```
*****
```

(d)

```
*
```

```
**
```

Exercises

```
***      **** *      *****      ***
***      **** *      ****      ***
****      **** *      ****      ****
*****      ***      ***      ****
*****      **      **      ****
*****      *      *      ****
*****      *      *      ****
*****      *      *      ****
*****      *      *      ****
```

5.13

One interesting application of computers is the drawing of graphs and bar charts. Write a program that reads five numbers (each between 1 and 30). Assume that the user enters only valid values. For each number that is read, your program should print a line containing that number of adjacent asterisks. For example, if your program reads the number 7, it should print *****.

[Page 235]

5.14

A mail order house sells five different products whose retail prices are: product 1 \$2.98, product 2\$4.50, product 3\$9.98, product 4\$4.49 and product 5\$6.87. Write a program that reads a series of pairs of numbers as follows:

a.

product number

b.

quantity sold

Your program should use a `switch` statement to determine the retail price for each product. Your program should calculate and display the total retail value of all products sold. Use a sentinel-controlled loop to determine when the program should stop looping and display the final results.

5.15

Modify the `GradeBook` program of Fig. 5.9Fig. 5.11 so that it calculates the grade-point average for the set of grades. A grade of A is worth 4 points, B is worth 3 points, etc.

5.16

Modify the program in Fig. 5.6 so it uses only integers to calculate the compound interest. [Hint: Treat all

monetary amounts as integral numbers of pennies. Then "break" the result into its dollar portion and cents portion by using the division and modulus operations. Insert a period.]

5.17

Assume $i = 1$, $j = 2$, $k = 3$ and $m = 2$. What does each of the following statements print? Are the parentheses necessary in each case?

a.

```
cout << ( i == 1 ) << endl;
```

b.

```
cout << ( j == 3 ) << endl;
```

c.

```
cout << ( i >= 1 && j < 4 ) << endl;
```

d.

```
cout << ( m <= 99 && k < m ) << endl;
```

e.

```
cout << ( j >= i || k == m ) << endl;
```

f.

```
cout << ( k + m < j || 3 - j >= k ) << endl;
```

g.

```
cout << ( !m ) << endl;
```

h.

```
cout << ( !( j - m ) ) << endl;
```

i.

```
cout << ( !( k > m ) ) << endl;
```

5.18

Write a program that prints a table of the binary, octal and hexadecimal equivalents of the decimal numbers in the range 1 through 256. If you are not familiar with these number systems, read [Appendix D, Number Systems](#), first.

5.19

Calculate the value of π from the infinite series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \dots$$

Print a table that shows the approximate value of π after each of the first 1,000 terms of this series.

5.20

(Pythagorean Triples) A right triangle can have sides that are all integers. A set of three integer values for the sides of a right triangle is called a Pythagorean triple. These three sides must satisfy the relationship that the sum of the squares of two of the sides is equal to the square of the hypotenuse. Find all Pythagorean triples for `side1`, `side2` and `hypotenuse` all no larger than 500. Use a triple-nested `for` loop that tries all possibilities. This is an example of [brute force](#) computing. You will learn in more advanced computer-science courses that there are many interesting problems for which there is no known algorithmic approach other than sheer brute force.

5.21

A company pays its employees as managers (who receive a fixed weekly salary), hourly workers (who receive a fixed hourly wage for up to the first 40 hours they work and "time-and-a-half" 1.5 times their hourly wage for overtime hours worked), commission workers (who receive \$250 plus 5.7 percent of their gross weekly sales), or pieceworkers (who receive a fixed amount of money per item for each of the items they produce). Each pieceworker in this company works on only one type of item). Write a program to compute the weekly pay for each employee. You do not know the number of employees in advance. Each type of employee has its own pay code: Managers have code 1, hourly workers have code 2, commission workers have code 3 and pieceworkers have code 4. Use a `switch` to compute each employee's pay according to that employee's paycode. Within the `switch`, prompt the user (i.e., the payroll clerk) to enter the appropriate facts your program needs to calculate each employee's pay according to that employee's paycode.

5.22

(De Morgan's Laws) In this chapter, we discussed the logical operators `&&`, `||` and `!`. De Morgan's laws can sometimes make it more convenient for us to express a logical expression. These laws state that the expression `!(condition1 && condition2)` is logically equivalent to the expression `!(condition1 || condition2)`. Also, the expression `!(condition1 || condition2)` is logically equivalent to the expression `(!condition1 && ! condition2)`. Use De Morgan's laws to write equivalent expressions for each of the following, then write a program to show that the original expression and the new expression in each case are equivalent:

a.

```
!( x < 5 ) && !( y >= 7 )
```

b.

```
!( a == b ) || !( g != 5 )
```

c.

```
!( ( x <= 8 ) && ( y > 4 ) )
```

d.

```
!( ( i > 4 ) || ( j <= 6 ) )
```

5.23

Write a program that prints the following diamond shape. You may use output statements that print either a single asterisk (*) or a single blank. Maximize your use of repetition (with nested `for` statements) and minimize the number of output statements.

```
*  
***  
*****  
*****  
*****  
*****  
***  
*  
*
```

5.24

Modify the program you wrote in [Exercise 5.23](#) to read an odd number in the range 1 to 19 to specify the number of rows in the diamond, then display a diamond of the appropriate size.

5.25

A criticism of the `break` and `continue` statements is that each is unstructured. Actually they statements can always be replaced by structured statements, although doing so can be awkward. Describe in general how you would remove any `break` statement from a loop in a program and replace it with some structured equivalent. [Hint: The `break` statement leaves a loop from within the body of the loop. Another way to leave is by failing the loop-continuation test. Consider using in the loop-continuation test a second test that indicates "early exit because of a 'break' condition."] Use the technique you developed here to remove the `break` statement from the program of [Fig. 5.13](#).

5.26

What does the following program segment do?

```

1  for ( int i = 1; i <= 5; i++ )
2  {
3      for ( int j = 1; j <= 3; j++ )
4      {
5          for ( int k = 1; k <= 4 ; k++ )
6              cout << '*';
7
8          cout << endl;
9      } // end inner for
10
11     cout << endl;
12 } // end outer for

```

[Page 237]

5.27

Describe in general how you would remove any `continue` statement from a loop in a program and replace it with some structured equivalent. Use the technique you developed here to remove the `continue` statement from the program of [Fig. 5.14](#).

5.28

("The Twelve Days of Christmas" Song) Write a program that uses repetition and `switch` statements to print the song "The Twelve Days of Christmas." One `switch` statement should be used to print the day (i.

e., "First," "Second," etc.). A separate `switch` statement should be used to print the remainder of each verse. Visit the Web site www.12days.com/library/carols/12daysofxmas.htm for the complete lyrics to the song.

5.29

(Peter Minuit Problem) Legend has it that, in 1626, Peter Minuit purchased Manhattan Island for \$24.00 in barter. Did he make a good investment? To answer this question, modify the compound interest program of Fig. 5.6 to begin with a principal of \$24.00 and to calculate the amount of interest on deposit if that money had been kept on deposit until this year (e.g., 379 years through 2005). Place the `for` loop that performs the compound interest calculation in an outer `for` loop that varies the interest rate from 5 percent to 10 percent to observe the wonders of compound interest.

 PREVNEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 239]

Outline

[6.1](#) Introduction

[6.2](#) Program Components in C++

[6.3](#) Math Library Functions

[6.4](#) Function Definitions with Multiple Parameters

[6.5](#) Function Prototypes and Argument Coercion

[6.6](#) C++ Standard Library Header Files

[6.7](#) Case Study: Random Number Generation

[6.8](#) Case Study: Game of Chance and Introducing enum

[6.9](#) Storage Classes

[6.10](#) Scope Rules

[6.11](#) Function Call Stack and Activation Records

[6.12](#) Functions with Empty Parameter Lists

[6.13](#) Inline Functions

[6.14](#) References and Reference Parameters

[6.15](#) Default Arguments

[6.16](#) Unary Scope Resolution Operator

[6.17](#) Function Overloading

[6.18 Function Templates](#)

[6.19 Recursion](#)

[6.20 Example Using Recursion: Fibonacci Series](#)

[6.21 Recursion vs. Iteration](#)

[6.22 \(Optional\) Software Engineering Case Study: Identifying Class Operations in the ATM System](#)

[6.23 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

[Exercises](#)

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 240]

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 will also learn 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 performanceinline 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 templatesa 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 that is discussed at length in upper-level computer science courses.

 PREV

page footer

NEXT 

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 241]

Software Engineering Observation 6.2



To promote software reusability, every function should be limited to performing a single, well-defined task, and the name of the function should express that task effectively. Such functions make programs easier to write, test, debug and maintain.

Error-Prevention Tip 6.1



A small function that performs one task is easier to test and debug than a larger function that performs many tasks.

Software Engineering Observation 6.3

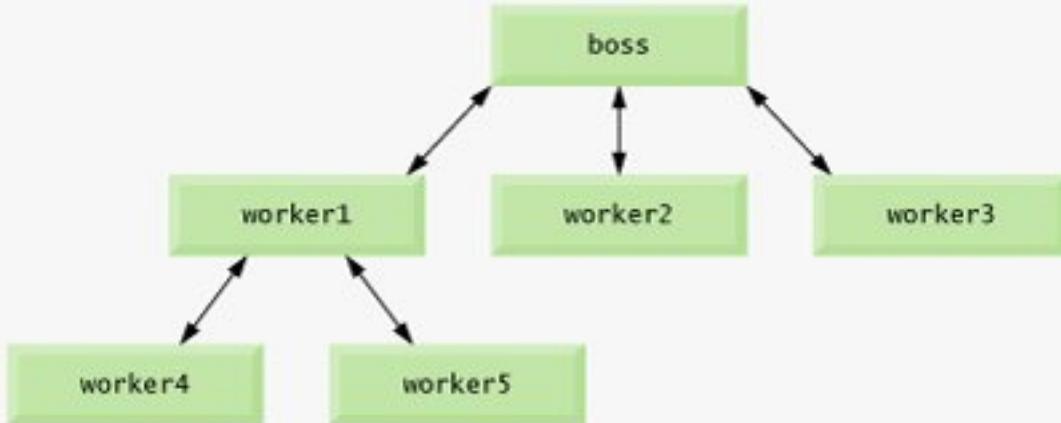


If you cannot choose a concise name that expresses a function's task, your function might be attempting to perform too many diverse tasks. It is usually best to break such a function into several smaller functions.

As you know, a function is invoked by a function call, and when the called function completes its task, it either returns a result or simply returns control to the caller. An analogy to this program structure is the hierarchical form of management ([Figure 6.1](#)). A boss (similar to the calling function) asks a worker (similar to the called function) to perform a task and report back (i.e., return) the results after completing the task. The boss function does not know how the worker function performs its designated tasks. The worker may also call other worker functions, unbeknownst to the boss. This hiding of implementation details promotes good software engineering. [Figure 6.1](#) shows the boss function communicating with several worker functions in a hierarchical manner. The `boss` function divides the responsibilities among the various worker functions. Note that `worker1` acts as a "boss function" to `worker4` and `worker5`.

Figure 6.1. Hierarchical boss function/worker function relationship.

[\[View full size image\]](#)



[◀ PREV](#)

page footer

[NEXT ▶](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 242]

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. 6.2. In the figure, the variables `x` and `y` are of type `double`.

Figure 6.2. Math library functions.

[Page 243]

Function	Description	Example
<code>ceil(x)</code>	rounds <code>x</code> to the smallest integer not less than <code>x</code>	<code>ceil(9.2)</code> is <code>10.0</code> <code>ceil(-9.8)</code> is <code>-9.0</code>
<code>cos(x)</code>	trigonometric cosine of <code>x</code> (<code>x</code> in radians)	<code>cos(0.0)</code> is <code>1.0</code>
<code>exp(x)</code>	exponential function e^x	<code>exp(1.0)</code> is <code>2.71828</code> <code>exp(2.0)</code> is <code>7.38906</code>

<code>fabs(x)</code>	absolute value of x	<code>fabs(5.1)</code> is 5.1 <code>fabs(0.0)</code> is 0.0 <code>fabs(-8.76)</code> is 8.76
<code>floor(x)</code>	rounds x to the largest integer not greater than x	<code>floor(9.2)</code> is 9.0 <code>floor(-9.8)</code> is -10.0
<code>fmod(x, y)</code>	remainder of x/y as a floating-point number	<code>fmod(2.6, 1.2)</code> is 0.2
<code>log(x)</code>	natural logarithm of x (base e)	<code>log(2.718282)</code> is 1.0 <code>log(7.389056)</code> is 2.0
<code>log10(x)</code>	logarithm of x (base 10)	<code>log10(10.0)</code> is 1.0 <code>log10(100.0)</code> is 2.0
<code>pow(x, y)</code>	x raised to power y (x^y)	<code>pow(2, 7)</code> is 128 <code>pow(9, .5)</code> is 3
<code>sin(x)</code>	trigonometric sine of x (x in radians)	<code>sin(0.0)</code> is 0
<code>sqrt(x)</code>	square root of x (where x is a nonnegative value)	<code>sqrt(9.0)</code> is 3.0
<code>tan(x)</code>	trigonometric tangent of x (x in radians)	<code>tan(0.0)</code> is 0

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 244]

Figure 6.4. GradeBook class defines function maximum.

(This item is displayed on pages 244 - 245 in the print version)

```
1 // Fig. 6.4: GradeBook.cpp
2 // Member-function definitions for class GradeBook that
3 // determines the maximum of three grades.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "GradeBook.h" // include definition of class GradeBook
10
11 // constructor initializes courseName with string supplied as argument;
12 // initializes studentMaximum to 0
13 GradeBook::GradeBook( string name )
14 {
15     setCourseName( name ); // validate and store courseName
16     maximumGrade = 0; // this value will be replaced by the maximum grade
17 } // end GradeBook constructor
18
19 // function to set the course name; limits name to 25 or fewer characters
20 void GradeBook::setCourseName( string name )
21 {
22     if ( name.length() <= 25 ) // if name has 25 or fewer characters
23         courseName = name; // store the course name in the object
24     else // if name is longer than 25 characters
25     { // set courseName to first 25 characters of parameter name
26         courseName = name.substr( 0, 25 ); // select first 25 characters
27         cout << "Name \" " << name << "\" exceeds maximum length (25).\n"
28             << "Limiting courseName to first 25 characters.\n" << endl;
29     } // end if...else
30 } // end function setCourseName
31
32 // function to retrieve the course name
33 string GradeBook::getCourseName()
34 {
35     return courseName;
36 } // end function getCourseName
37
```

```

38 // display a welcome message to the GradeBook user
39 void GradeBook::displayMessage()
40 {
41     // this statement calls getCourseName to get the
42     // name of the course this GradeBook represents
43     cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
44     << endl;
45 } // end function displayMessage
46
47 // input three grades from user; determine maximum
48 void GradeBook::inputGrades()
49 {
50     int grade1; // first grade entered by user
51     int grade2; // second grade entered by user
52     int grade3; // third grade entered by user
53
54     cout << "Enter three integer grades: ";
55     cin >> grade1 >> grade2 >> grade3;
56
57     // store maximum in member studentMaximum
58     maximumGrade = maximum( grade1, grade2, grade3 );
59 } // end function inputGrades
60
61 // returns the maximum of its three integer parameters
62 int GradeBook::maximum( int x, int y, int z )
63 {
64     int maximumValue = x; // assume x is the largest to start
65
66     // determine whether y is greater than maximumValue
67     if ( y > maximumValue )
68         maximumValue = y; // make y the new maximumValue
69
70     // determine whether z is greater than maximumValue
71     if ( z > maximumValue )
72         maximumValue = z; // make z the new maximumValue
73
74     return maximumValue;
75 } // end function maximum
76
77 // display a report based on the grades entered by user
78 void GradeBook::displayGradeReport()
79 {
80     // output maximum of grades entered
81     cout << "Maximum of grades entered: " << maximumGrade << endl;
82 } // end function displayGradeReport

```

Figure 6.5. Demonstrating function maximum.

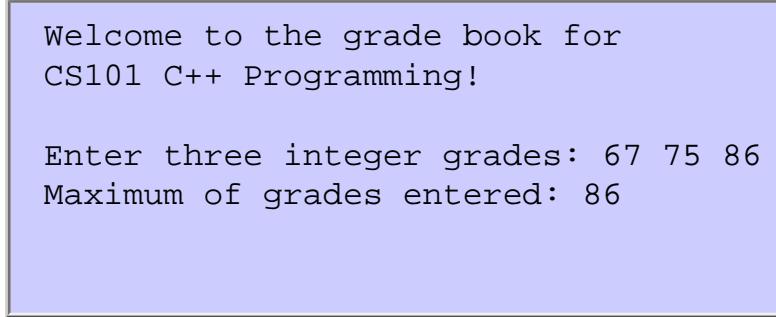
```
1 // Fig. 6.5: fig06_05.cpp
2 // Create GradeBook object, input grades and display grade report.
3 #include "GradeBook.h" // include definition of class GradeBook
4
5 int main()
6 {
7     // create GradeBook object
8     GradeBook myGradeBook( "CS101 C++ Programming" );
9
10    myGradeBook.displayMessage(); // display welcome message
11    myGradeBook.inputGrades(); // read grades from user
12    myGradeBook.displayGradeReport(); // display report based on grades
13    return 0; // indicate successful termination
14 } // 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 6.4



The commas used in line 58 of Fig. 6.4 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 guarantee that all arguments in a function call are 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.

[Page 247]

Error-Prevention Tip 6.2



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. 6.3, 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. 6.4, line 62) matches the function prototype and indicates that the parameter names are `x`, `y` and `z`. When `maximum` is called (Fig. 6.4, 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 6.5



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 6275 of Fig. 6.4), 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 at lines 6768 determines whether `y` is greater than `maximumValue` and, if so, assigns `y` to `maximumValue`. The `if` statement at lines 7172 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.

[Page 248]

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.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 249]

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. 6.3](#), 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, 9, 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 argumentthe 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 expressionthe 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 6.6](#) lists the fundamental data types in order from "highest type" to "lowest type."

Figure 6.6. Promotion hierarchy for fundamental data types.

(This item is displayed on page 250 in the print version)

Data types	
<code>long double</code>	
<code>double</code>	
<code>float</code>	
<code>unsigned long int</code>	(synonymous with <code>unsigned long</code>)
<code>long int</code>	(synonymous with <code>long</code>)
<code>unsigned int</code>	(synonymous with <code>unsigned</code>)
<code>int</code>	
<code>unsigned short int</code>	(synonymous with <code>unsigned short</code>)
<code>short int</code>	(synonymous with <code>short</code>)
<code>unsigned char</code>	
<code>char</code>	
<code>bool</code>	

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 4.9](#)). 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.

[Page 250]

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.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 251]

Figure 6.7. C++ Standard Library header files.

[Page 252]

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 4.9 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; Chapter 19 , Web Programming; Chapter 22 , Bits, Characters, C-Strings and structs; and Appendix E , C Legacy Code Topics. 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 .

<pre><vector>, <list>, <deque>, <queue>, <stack>, <map>, <set>, <bitset></pre>	<p>These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The <code><vector></code> header is first introduced in Chapter 7, Arrays and Vectors. We discuss all these header files in Chapter 23, Standard Template Library (STL).</p>
<code><cctype></code>	<p>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 <code><ctype.h></code>. These topics are discussed in Chapter 8, Pointers and Pointer-Based Strings, and Chapter 22, Bits, Characters, C-Strings and <code>structs</code>.</p>
<code><cstring></code>	<p>Contains function prototypes for C-style string-processing functions. This header file replaces header file <code><string.h></code>. This header file is used in Chapter 11, Operator Overloading; String and Array Objects.</p>
<code><typeinfo></code>	<p>Contains classes for runtime type identification (determining data types at execution time). This header file is discussed in Section 13.8.</p>
<code><exception>, <stdexcept></code>	<p>These header files contain classes that are used for exception handling (discussed in Chapter 16).</p>
<code><memory></code>	<p>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.</p>
<code><fstream></code>	<p>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 <code><fstream.h></code>.</p>
<code><string></code>	<p>Contains the definition of class <code>string</code> from the C++ Standard Library (discussed in Chapter 18).</p>
<code><iostream></code>	<p>Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 18, Class <code>string</code> and String Stream Processing).</p>

<functional>	Contains classes and functions used by C++ Standard Library algorithms. This header file is used in Chapter 23 .
<iterator>	Contains classes for accessing data in the C++ Standard Library containers. This header file is used in Chapter 23 , Standard Template Library (STL).
<algorithm>	Contains functions for manipulating data in C++ Standard Library containers. This header file is used in Chapter 23 .
<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 F , 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.
<utility>	Contains classes and functions that are used by many C++ Standard Library header files.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 253]

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 214748647; 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. 6.8](#)) 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
```

Figure 6.8. Shifted, scaled integers produced by 1 + rand() % 6.

(This item is displayed on pages 253 - 254 in the print version)

```

1 // Fig. 6.8: fig06_08.cpp
2 // Shifted and scaled random integers.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib> // contains function prototype for rand
11 using std::rand;
12
13 int main()
14 {
15     // loop 20 times
16     for ( int counter = 1; counter <= 20; counter++ )
17     {
18         // pick random number from 1 to 6 and output it
19         cout << setw( 10 ) << ( 1 + rand() % 6 );
20
21         // if counter is divisible by 5, start a new line of output
22         if ( counter % 5 == 0 )
23             cout << endl;
24     } // end for
25
26     return 0; // indicates successful termination
27 } // end main

```

6	6	5	5	6
5	1	1	5	3
6	6	2	4	2
6	2	3	4	1

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 6.8 confirms that the results are in the range 1 to 6.

[Page 254]

Rolling a Six-Sided Die 6,000,000 Times

To show that the numbers produced by function `rand` occur with approximately equal likelihood, Fig. 6.9 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. 6.9.

Figure 6.9. Rolling a six-sided die 6,000,000 times.

(This item is displayed on pages 254 - 255 in the print version)

```

1 // Fig. 6.9: fig06_09.cpp
2 // Roll a six-sided die 6,000,000 times.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib> // contains function prototype for rand
11 using std::rand;
12
13 int main()
14 {
15     int frequency1 = 0; // count of 1s rolled
16     int frequency2 = 0; // count of 2s rolled
17     int frequency3 = 0; // count of 3s rolled
18     int frequency4 = 0; // count of 4s rolled
19     int frequency5 = 0; // count of 5s rolled
20     int frequency6 = 0; // count of 6s rolled
21
22     int face; // stores most recently rolled value
23
24     // summarize results of 6,000,000 rolls of a die
25     for ( int roll = 1; roll <= 6000000; roll++ )
26     {
27         face = 1 + rand() % 6; // random number from 1 to 6
28
29         // determine roll value 1-6 and increment appropriate counter
30         switch ( face )
31         {
32             case 1:
33                 ++frequency1; // increment the 1s counter
34                 break;
35             case 2:
36                 ++frequency2; // increment the 2s counter
37                 break;
38             case 3:

```

```

39             ++frequency3; // increment the 3s counter
40             break;
41         case 4:
42             ++frequency4; // increment the 4s counter
43             break;
44         case 5:
45             ++frequency5; // increment the 5s counter
46             break;
47         case 6:
48             ++frequency6; // increment the 6s counter
49             break;
50     default: // invalid value
51         cout << "Program should never get here!" ;
52     } // end switch
53 } // end for
54
55 cout << "Face" << setw( 13 ) << "Frequency" << endl; // output headers
56 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

[Page 255]

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. 6.9](#) elegantly with a single-line statement.

[Page 256]

Error-Prevention Tip 6.3



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. 6.8](#) again produces

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. 6.8](#). 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 6.10](#) 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 04294967295. Function `srand` takes an `unsigned int` value as an argument. The function prototype for the `srand` function is in header file `<cstdlib>`.

Figure 6.10. Randomizing the die-rolling program.

(This item is displayed on pages 256 - 257 in the print version)

```
1 // Fig. 6.10: fig06_10.cpp
2 // Randomizing die-rolling program.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <cstdlib> // contains prototypes for functions srand and rand
12 using std::rand;
13 using std::srand;
14
15 int main()
16 {
17     unsigned seed; // stores the seed entered by the user
18
19     cout << "Enter seed: ";
20     cin >> seed;
21     srand( seed ); // seed random number generator
22
23     // loop 10 times
24     for ( int counter = 1; counter <= 10; counter++ )
25     {
26         // pick random number from 1 to 6 and output it
27         cout << setw( 10 ) << ( 1 + rand() % 6 );
28
29         // if counter is divisible by 5, start a new line of output
30         if ( counter % 5 == 0 )
31             cout << endl;
32     } // end for
33
34     return 0; // indicates successful termination
35 } // 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

[Page 257]

Let us 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 ) );
```

[Page 258]

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 $1 \leq \text{face} \leq 6$. 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. The exercises show that it is possible to choose integers at random from sets of values other than ranges of consecutive integers.

Common Programming Error 6.8



Using `srand` in place of `rand` to attempt to generate random numbers is a compilation error if function `srand` does not return a value.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 259]

[Page 261]

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 the programmer 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.

[Page 262]

After the first roll, if the game is won or lost, the program skips the body of the `while` statement (lines 4959) because `gameStatus` is not equal to `CONTINUE`. The program proceeds to the `if...else` statement at lines 6265, 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. In the exercises, we investigate various interesting characteristics of the game of craps.

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 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.

[◀ PREV](#)

[page footer](#)

[NEXT ▶](#)

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 263]

An identifier's linkage determines whether an identifier 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.

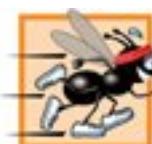
Local Variables

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` variables `x` and `y` are local variables of automatic storage class—they exist only in the nearest enclosing pair of curly braces within the body of the function in which the definition appears:

```
auto double x, y;
```

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 6.8



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.

[Page 264]

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. Today's optimizing compilers are capable of recognizing frequently used variables and can decide to place them in registers without needing a `register` declaration from the programmer.

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 will 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 6.9



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 6.10



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 they are not explicitly initialized by the programmer, 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. In [Appendix E, C Legacy Code Topics](#), we discuss using `extern` and `static` with external identifiers and multiple-source-file programs.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 266]

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 the programmer wants 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. 6.12](#) demonstrates scoping issues with global variables, automatic local variables and static local variables.

Figure 6.12. Scoping example.

(This item is displayed on pages 266 - 267 in the print version)

```

1 // Fig. 6.12: fig06_12.cpp
2 // A scoping example.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void useLocal( void ); // function prototype
8 void useStaticLocal( void ); // function prototype
9 void useGlobal( void ); // function prototype
10
11 int x = 1; // global variable
12
13 int main()
14 {
15     int x = 5; // local variable to main
16

```

```

17     cout << "local x in main's outer scope is " << x << endl;
18
19 { // start new scope
20     int x = 7; // hides x in outer scope
21
22     cout << "local x in main's inner scope is " << x << endl;
23 } // end new scope
24
25 cout << "local x in main's outer scope is " << x << endl;
26
27 useLocal(); // useLocal has local x
28 useStaticLocal(); // useStaticLocal has static local x
29 useGlobal(); // useGlobal uses global x
30 useLocal(); // useLocal reinitializes its local x
31 useStaticLocal(); // static local x retains its prior value
32 useGlobal(); // global x also retains its value
33
34 cout << "\nlocal x in main is " << x << endl;
35 return 0; // indicates successful termination
36 } // end main
37
38 // useLocal reinitializes local variable x during each call
39 void useLocal( void )
40 {
41     int x = 25; // initialized each time useLocal is called
42
43     cout << "\nlocal x is " << x << " on entering useLocal" << endl;
44     x++;
45     cout << "local x is " << x << " on exiting useLocal" << endl;
46 } // end function useLocal
47
48 // useStaticLocal initializes static local variable x only the
49 // first time the function is called; value of x is saved
50 // between calls to this function
51 void useStaticLocal( void )
52 {
53     static int x = 50; // initialized first time useStaticLocal is called
54
55     cout << "\nlocal static x is " << x << " on entering useStaticLocal"
56         << endl;
57     x++;
58     cout << "local static x is " << x << " on exiting useStaticLocal"
59         << endl;
60 } // end function useStaticLocal
61
62 // useGlobal modifies global variable x during each call
63 void useGlobal( void )
64 {

```

```

65     cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
66     x *= 10;
67     cout << "global x is " << x << " on exiting useGlobal" << endl;
68 } // end function useGlobal

```

```

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

```

[Page 268]

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 declares a local variable `x` and initializes it to 5. Line 17 outputs this variable to show that the global `x` is hidden in `main`. Next, lines 19-23 define a new block in `main` in which another local variable `x` is initialized to 7 (line 20). Line 22 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 25 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 3946) declares automatic variable `x` (line 41) 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 5160) 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 53 occurs only once the first time `useStaticLocal` is called.

Function `useGlobal` (lines 6368) does not declare any variables. Therefore, when it refers to variable `x`, the global `x` (preceding `main`) is used. When the program calls `useGlobal`, the function prints the global variable `x`, multiplies it by 10 and prints 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.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 269]

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 function that called it. So, somehow, we must keep track of the return addresses that each function needs to return control to the function that called it. The function call stack is the perfect data structure for handling this information. Each time a function calls another function, 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. It also contains some additional information we will soon discuss. If the called function returns, instead of calling another function before returning, 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 reserve the memory for the called function's automatic variables. That stack frame exists as long as the called function is active. When the called 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 have 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 1117 of Fig. 6.13). First the operating system calls `main` this pushes an activation record onto the stack (shown in Fig. 6.14). 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).

[Page 271]

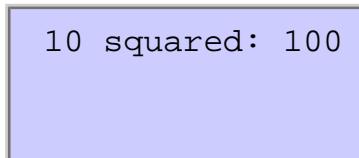
Figure 6.13. square function used to demonstrate the function call stack and activation records.

(This item is displayed on pages 269 - 270 in the print version)

```

1 // Fig. 6.13: fig06_13.cpp
2 // square function used to demonstrate the function
3 // call stack and activation records.
4 #include <iostream>
5 using std::cin;
6 using std::cout;
7 using std::endl;
8
9 int square( int ); // prototype for function square
10
11 int main()
12 {
13     int a = 10; // value to square (local automatic variable in main)
14
15     cout << a << " squared: " << square( a ) << endl; // display a squared
16     return 0; // indicate successful termination
17 } // end main
18
19 // returns the square of an integer
20 int square( int x ) // x is a local variable
21 {
22     return x * x; // calculate square and return result
23 } // end function square

```

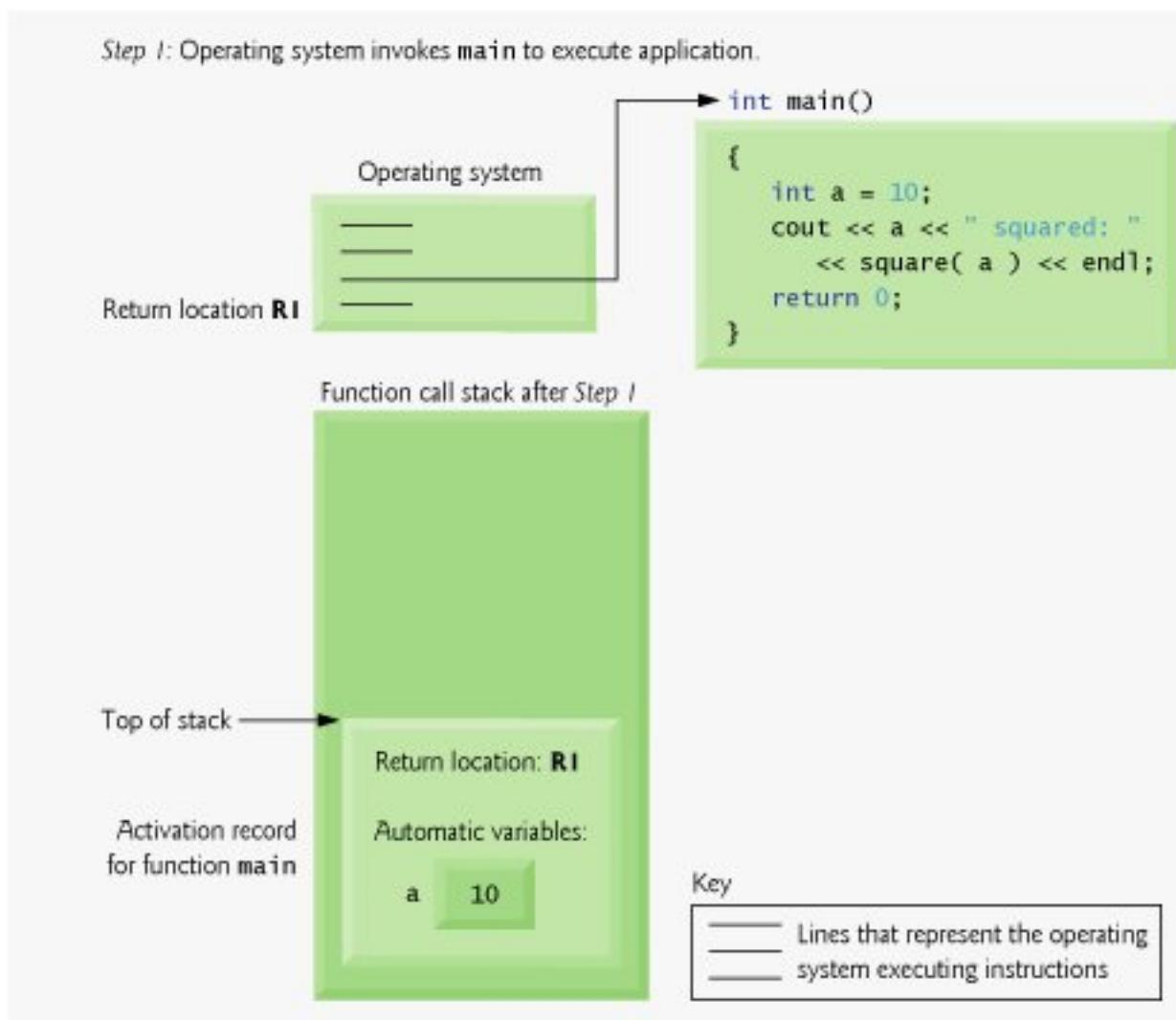


10 squared: 100

Figure 6.14. Function call stack after the operating system invokes `main` to execute the application.

(This item is displayed on page 270 in the print version)

[\[View full size image\]](#)

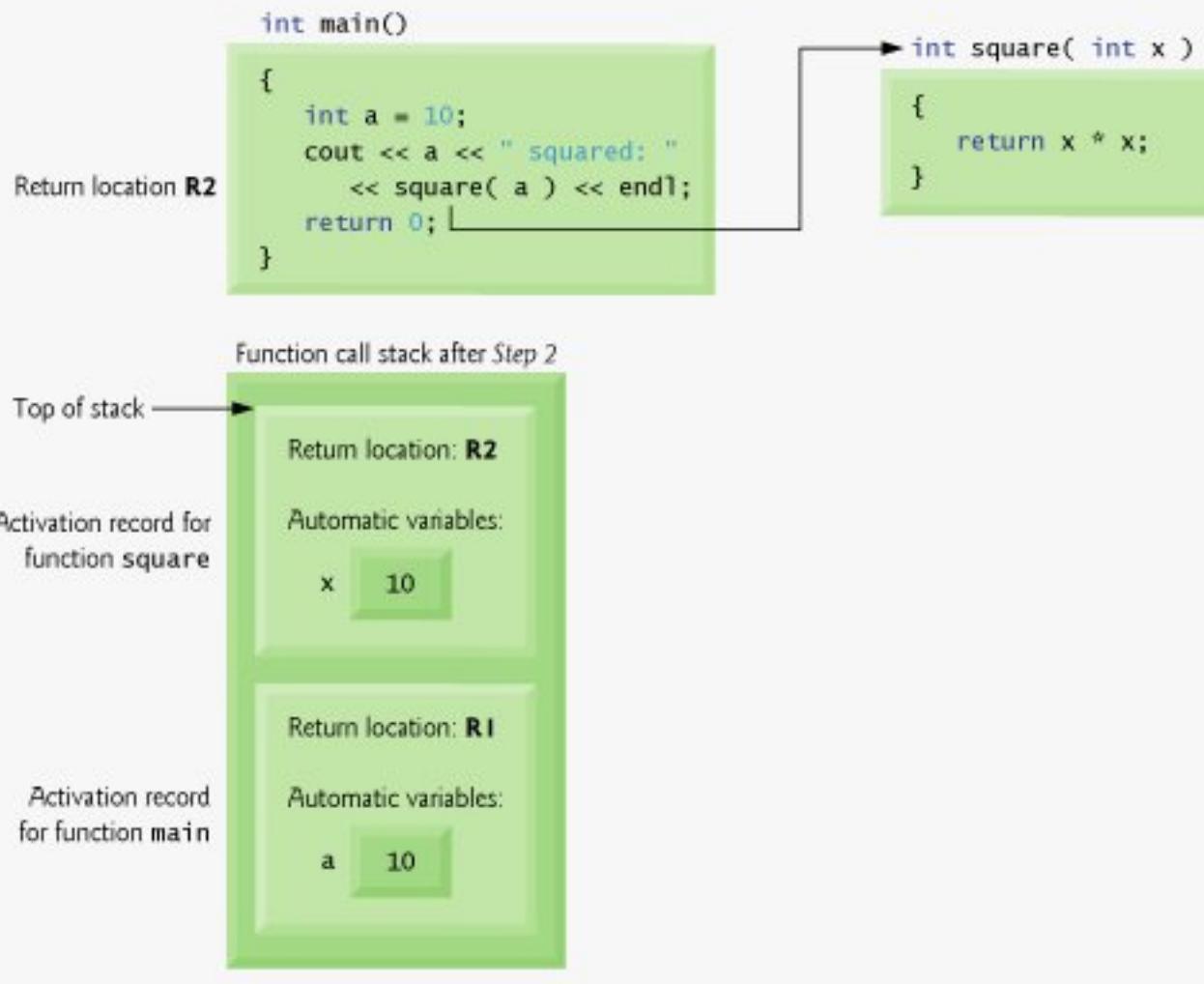


Function `main` before returning to the operating system now calls function `square` in line 15 of Fig. 6.13. This causes a stack frame for `square` (lines 2023) to be pushed onto the function call stack (Fig. 6.15). This stack frame contains the return address that `square` needs to return to `main` (i.e., `R2`) and the memory for `square`'s automatic variable (i.e., `x`).

Figure 6.15. Function call stack after `main` invokes function `square` to perform the calculation.

[View full size image]

Step 2: main invokes function square to perform calculation.



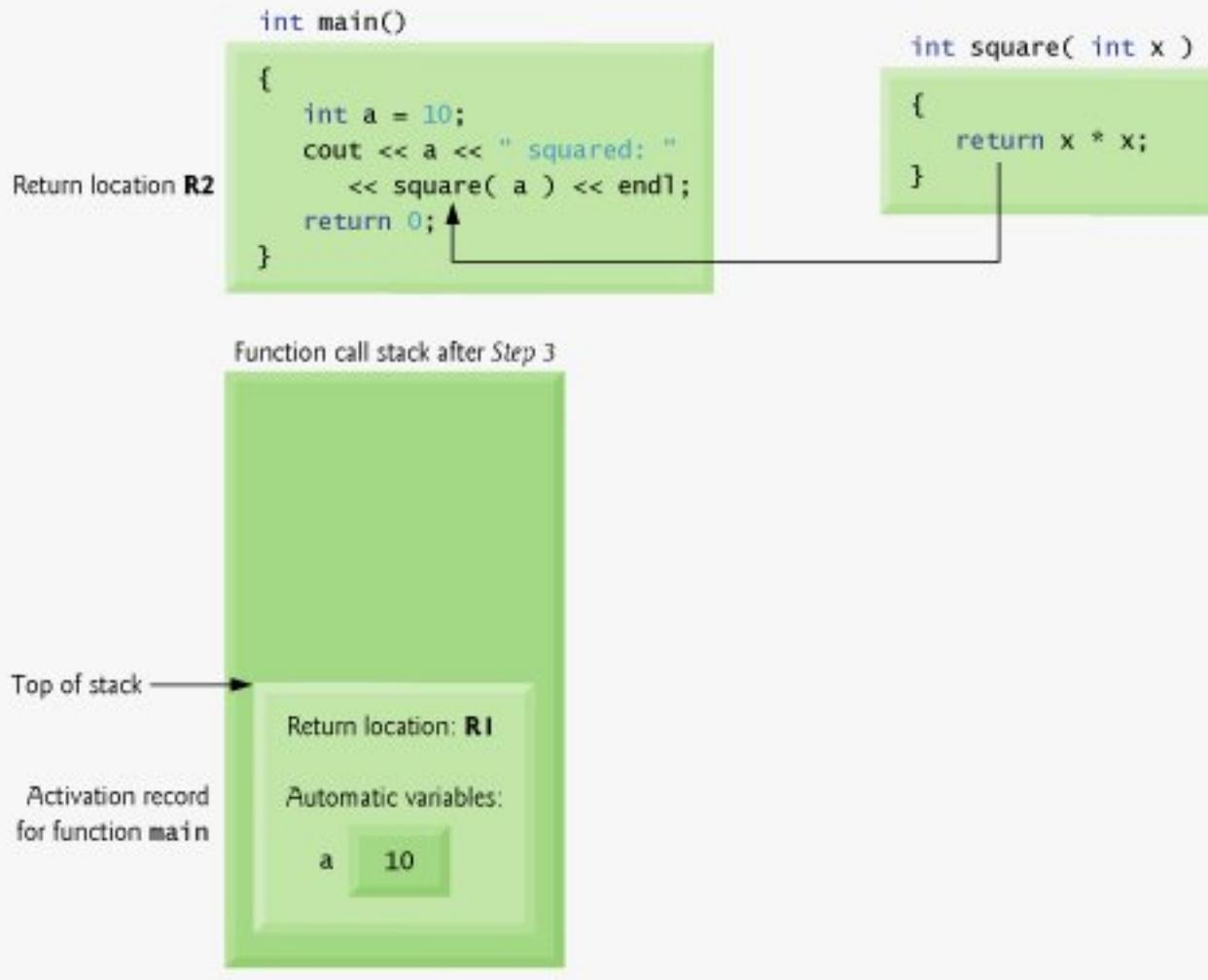
After square calculates the square of its argument, it needs to return to main and no longer needs the memory for its automatic variable x. So the stack is popped giving square the return location in main (i.e., R2) and losing square's automatic variable. Figure 6.16 shows the function call stack after square's activation record has been popped.

Figure 6.16. Function call stack after function square returns to main.

(This item is displayed on page 272 in the print version)

[\[View full size image\]](#)

Step 3: `square` returns its result to `main`.



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. 6.14) and causes the memory for `main`'s automatic variable (i.e., `a`) to become unavailable.

You have now seen how valuable the notion of the stack data structure is in implementing a key mechanism that supports program execution. Data structures have many important applications in computer science. We discuss stacks, queues, lists, trees and other data structures in [Chapter 21, Data Structures](#), and [Chapter 23, Standard Template Library \(STL\)](#).

[Page 272]

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 273]

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.

[PREV](#)

[page footer](#)

[NEXT](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 274]

Software Engineering Observation 6.11



Any change to an `inline` function could require 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 6.18 uses `inline` function `cube` (lines 1114) 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 [Chapter 7](#), [Chapter 8](#) and [Chapter 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.

Figure 6.18. `inline` function that calculates the volume of a cube.

(This item is displayed on pages 274 - 275 in the print version)

```

1 // Fig. 6.18: fig06_18.cpp
2 // Using an inline function to calculate the volume of a cube.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // Definition of inline function cube. Definition of function appears
9 // before function is called, so a function prototype is not required.
10 // First line of function definition acts as the prototype.
11 inline double cube( const double side )
12 {
13     return side * side * side; // calculate cube
14 } // end function cube
15
16 int main()
17 {
18     double sideValue; // stores value entered by user
19     cout << "Enter the side length of your cube: ";
20     cin >> sideValue; // read value from user
21
22     // calculate cube of sideValue and display result
23     cout << "Volume of cube with side "
24         << sideValue << " is " << cube( sideValue ) << endl;
25     return 0; // indicates successful termination
26 } // end main

```

Enter the side length of your cube: 3.5
 Volume of cube with side 3.5 is 42.875

Software Engineering Observation 6.12



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.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 276]

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 6.19 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, however, the compiler has no trouble resolving the ambiguity.

Figure 6.19. Passing arguments by value and by reference.

(This item is displayed on pages 276 - 277 in the print version)

```

1 // Fig. 6.19: fig06_19.cpp
2 // Comparing pass-by-value and pass-by-reference with references.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int squareByValue( int ); // function prototype (value pass)
8 void squareByReference( int & ); // function prototype (reference pass)
9
10 int main()
11 {
12     int x = 2; // value to square using squareByValue
13     int z = 4; // value to square using squareByReference
14
15     // demonstrate squareByValue
16     cout << "x = " << x << " before squareByValue\n";
17     cout << "Value returned by squareByValue: "
18         << squareByValue( x ) << endl;
19     cout << "x = " << x << " after squareByValue\n" << endl;
20 }
```

```

21 // demonstrate squareByReference
22 cout << "z = " << z << " before squareByReference" << endl;
23 squareByReference( z );
24 cout << "z = " << z << " after squareByReference" << endl;
25 return 0; // indicates successful termination
26 } // end main
27
28 // squareByValue multiplies number by itself, stores the
29 // result in number and returns the new value of number
30 int squareByValue( int number )
31 {
32     return number *= number; // caller's argument not modified
33 } // end function squareByValue
34
35 // squareByReference multiplies numberRef by itself and stores the result
36 // in the variable to which numberRef refers in function main
37 void squareByReference( int &numberRef )
38 {
39     numberRef *= numberRef; // caller's argument modified
40 } // end function squareByReference

```

```

x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference

```

[Page 277]

Common Programming Error 6.14



Because reference parameters are mentioned only by name in the body of the called function, the programmer 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 6.14



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 in line 37 of [Fig. 6.19](#) the placement of `&` in the parameter list of function `squareByReference`. Some C++ programmers prefer to write `int& numberRef`.

Software Engineering Observation 6.15



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. 6.19](#)). For example, the code

[Page 278]

```
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. 6.20 and Fig. 6.21) 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.

Figure 6.20. Initializing and using a reference.

```

1 // Fig. 6.20: fig06_20.cpp
2 // References must be initialized.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int x = 3;
10    int &y = x; // y refers to (is an alias for) x
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7; // actually modifies x
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main

```

```

x = 3
y = 3
x = 7
y = 7

```

Figure 6.21. Uninitialized reference causes a syntax error.

(This item is displayed on pages 278 - 279 in the print version)

```

1 // Fig. 6.21: fig06_21.cpp
2 // References must be initialized.
3 #include <iostream>
4 using std::cout;
5 using std::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:\cpphttp5_examples\ch06\Fig06_21\fig06_21.cpp 10:
Reference variable 'y' must be initialized in function main()
```

Microsoft Visual C++ compiler error message:

```
C:\cpphttp5_examples\ch06\Fig06_21\fig06_21.cpp(10) : error C2530: 'y' :
references must be initialized
```

GNU C++ compiler error message:

```
fig06_21.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

Note that the C++ standard does not specify the error messages that compilers use to indicate particular errors. For this reason, Fig. 6.21 shows the error messages produced by the Borland C++ 5.5 command-line compiler, Microsoft Visual C++.NET compiler and GNU C++ compiler when a reference is not initialized.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 281]

The first call to `boxVolume` (line 13) specifies no arguments, thus using all three default values of 1. The second call (line 17) passes a `length` argument, thus using default values of 1 for the `width` and `height` arguments. The third call (line 21) passes arguments for `length` and `width`, thus using a default value of 1 for the `height` argument. The last call (line 25) passes arguments for `length`, `width` and `height`, thus using no default values. Note that any arguments passed to the function explicitly are assigned to the function's parameters from left to right. Therefore, when `boxVolume` receives one argument, the function assigns the value of that argument to its `length` parameter (i.e., the leftmost parameter in the parameter list). When `boxVolume` receives two arguments, the function assigns the values of those arguments to its `length` and `width` parameters in that order. Finally, when `boxVolume` receives all three arguments, the function assigns the values of those arguments to its `length`, `width` and `height` parameters, respectively.

Good Programming Practice 6.6



Using default arguments can simplify writing function calls. However, some programmers feel that explicitly specifying all arguments is clearer.

Software Engineering Observation 6.16



If the default values for a function change, all client code using the function must be recompiled.

Common Programming Error 6.19



Specifying and attempting to use a default argument that is not a rightmost (trailing) argument (while not simultaneously defaulting all the rightmost arguments) is a syntax error.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 283]

Software Engineering Observation 6.17



Always using the unary scope resolution operator (::) to refer to global variables makes programs easier to modify by reducing the risk of name collisions with nonglobal variables.

Error-Prevention Tip 6.4



Always using the unary scope resolution operator (::) to refer to a global variable eliminates possible logic errors that might occur if a nonglobal variable hides the global variable.

Error-Prevention Tip 6.5



Avoid using variables of the same name for different purposes in a program. Although this is allowed in various circumstances, it can lead to errors.

[◀ PREV](#)

page footer

[NEXT ▶](#)

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 284]

How the Compiler Differentiates Overloaded Functions

Overloaded functions are distinguished by their signatures. A signature is a combination of a function's name and its parameter types (in order). The compiler encodes each function identifier with the number and types of its parameters (sometimes referred to as **name mangling** or **name decoration**) to enable **type-safe linkage**. Type-safe linkage ensures that the proper overloaded function is called and that the types of the arguments conform to the types of the parameters.

Figure 6.25 was compiled with the Borland C++ 5.6.4 command-line compiler. Rather than showing the execution output of the program (as we normally would), we show the mangled function names produced in assembly language by Borland C++. Each mangled name begins with @ followed by the function name. The function name is then separated from the mangled parameter list by \$q. In the parameter list for function nothing2 (line 25; see the fourth output line), c represents a char, i represents an int, rf represents a float & (i.e., a reference to a float) and rd represents a double & (i.e., a reference to a double). In the parameter list for function nothing1, i represents an int, f represents a float, c represents a char and ri represents an int &. The two square functions are distinguished by their parameter lists; one specifies d for double and the other specifies i for int. The return types of the functions are not specified in the mangled names. Overloaded functions can have different return types, but if they do, they must also have different parameter lists. Again, you cannot have two functions with the same signature and different return types. Note that function main is not mangled, because it cannot be overloaded.

[Page 285]

Figure 6.25. Name mangling to enable type-safe linkage.

```

1 // Fig. 6.25: fig06_25.cpp
2 // Name mangling.
3
4 // function square for int values
5 int square( int x )
6 {
7     return x * x;
8 } // end function square
9
10 // function square for double values
11 double square( double y )
12 {
13     return y * y;
14 } // end function square
15
16 // function that receives arguments of types
17 // int, float, char and int &
18 void nothing1( int a, float b, char c, int &d )
19 {
20     // empty function body
21 } // end function nothing1
22
23 // function that receives arguments of types
24 // char, int, float & and double &
25 int nothing2( char a, int b, float &c, double &d )
26 {
27     return 0;
28 } // end function nothing2
29
30 int main()
31 {
32     return 0; // indicates successful termination
33 } // end main

```

```

@square$qi
@square$qd
@nothing1$qifcri
@nothing2$qcirfrd
_main

```



Creating overloaded functions with identical parameter lists and different return types is a compilation error.

The compiler uses only the parameter lists to distinguish between functions of the same name. Overloaded functions need not have the same number of parameters. Programmers should use caution when overloading functions with default parameters, because this may cause ambiguity.

[Page 286]

Common Programming Error 6.22



A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error. For example, having in a program both a function that explicitly takes no arguments and a function of the same name that contains all default arguments results in a compilation error when an attempt is made to use that function name in a call passing no arguments. The compiler does not know which version of the function to choose.

Overloaded Operators

In [Chapter 11](#), we discuss how to overload operators to define how they should operate on objects of user-defined data types. (In fact, we have been using many overloaded operators to this point, including the stream insertion operator `<<` and the stream extraction operator `>>`, each of which is overloaded to be able to display data of all the fundamental types. We say more about overloading `<<` and `>>` to be able to handle objects of user-defined types in [Chapter 11](#).) [Section 6.18](#) introduces function templates for automatically generating overloaded functions that perform identical tasks on different data types.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 287]

Figure 6.26. Function template maximum header file.

(This item is displayed on page 286 in the print version)

```
1 // Fig. 6.26: maximum.h
2 // Definition of function template maximum.
3
4 template < class T > // or template< typename T >
5 T maximum( T value1, T value2, T value3 )
6 {
7     T maximumValue = value1; // assume value1 is maximum
8
9     // determine whether value2 is greater than maximumValue
10    if ( value2 > maximumValue )
11        maximumValue = value2;
12
13    // determine whether value3 is greater than maximumValue
14    if ( value3 > maximumValue )
15        maximumValue = value3;
16
17    return maximumValue;
18 } // end function template maximum
```

The function template in [Fig. 6.26](#) declares a single formal type parameter `T` (line 4) as a placeholder for the type of the data to be tested by function `maximum`. The name of a type parameter must be unique in the template parameter list for a particular template definition. When the compiler detects a `maximum` invocation in the program source code, the type of the data passed to `maximum` is substituted for `T` throughout the template definition, and C++ creates a complete function for determining the maximum of three values of the specified data type. Then the newly created function is compiled. Thus, templates are a means of code generation.

Common Programming Error 6.23



Not placing keyword `class` or keyword `typename` before every formal type parameter of a function template (e.g., writing `< class S, T >` instead of `< class S, class T >`) is a syntax error.

Figure 6.27 uses the `maximum` function template (lines 20, 30 and 40) to determine the largest of three `int` values, three `double` values and three `char` values.

Figure 6.27. Demonstrating function template `maximum`.

(This item is displayed on pages 287 - 288 in the print version)

```

1 // Fig. 6.27: fig06_27.cpp
2 // Function template maximum test program.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include "maximum.h" // include definition of function template maximum
9
10 int main()
11 {
12     // demonstrate maximum with int values
13     int int1, int2, int3;
14
15     cout << "Input three integer values: ";
16     cin >> int1 >> int2 >> int3;
17
18     // invoke int version of maximum
19     cout << "The maximum integer value is: "
20         << maximum( int1, int2, int3 );
21
22     // demonstrate maximum with double values
23     double double1, double2, double3;
24
25     cout << "\n\nInput three double values: ";
26     cin >> double1 >> double2 >> double3;
27
28     // invoke double version of maximum
29     cout << "The maximum double value is: "
30         << maximum( double1, double2, double3 );
31
32     // demonstrate maximum with char values
33     char char1, char2, char3;

```

```

34
35     cout << "\n\nInput three characters: ";
36     cin >> char1 >> char2 >> char3;
37
38     // invoke char version of maximum
39     cout << "The maximum character value is: "
40         << maximum( char1, char2, char3 ) << endl;
41     return 0; // indicates successful termination
42 } // end main

```

Input three integer values: 1 2 3
 The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
 The maximum double value is: 3.3

Input three characters: A C B
 The maximum character value is: C

[Page 288]

In Fig. 6.27, three functions are created as a result of the calls in lines 20, 30 and 40 expecting three int values, three double values and three char values, respectively. The function template specialization created for type int replaces each occurrence of T with int as follows:

```

int maximum( int value1, int value2, int value3 )
{
    int maximumValue = value1;

    // determine whether value2 is greater than maximumValue
    if ( value2 > maximumValue )
        maximumValue = value2;
    // determine whether value3 is greater than maximumValue
    if ( value3 > maximumValue )
        maximumValue = value3;
    return maximumValue;
} // end function template maximum

```

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 289]

We first consider recursion conceptually, then examine two programs containing recursive functions. Recursive problem-solving approaches have a number of elements in common. A recursive function is called to solve a problem. The function actually knows how to solve only the simplest case(s), or so-called **base case(s)**. If the function is called with a base case, the function simply returns a result. If the function is called with a more complex problem, it typically divides the problem into two conceptual piecesa piece that the function knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version. This new problem looks like the original problem, so the function launches (calls) a fresh copy of itself to work on the smaller problemthis is referred to as a **recursive call** and is also called the **recursion step**. The recursion step often includes the keyword `return`, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller, possibly `main`.

The recursion step executes while the original call to the function is still open, i.e., it has not yet finished executing. The recursion step can result in many more such recursive calls, as the function keeps dividing each new subproblem with which the function is called into two conceptual pieces. In order for the recursion to eventually terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller and smaller problems must eventually converge on the base case. At that point, the function recognizes the base case and returns a result to the previous copy of the function, and a sequence of returns ensues all the way up the line until the original function call eventually returns the final result to `main`. All of this sounds quite exotic compared to the kind of "conventional" problem solving we have been using to this point. As an example of these concepts at work, let us write a recursive program to perform a popular mathematical calculation.

The factorial of a nonnegative integer n , written $n!$ (and pronounced "n factorial"), is the product

$$n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

with $1!$ equal to 1, and $0!$ defined to be 1. For example, $5!$ is the product $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is equal to 120.

The factorial of an integer, `number`, greater than or equal to 0, can be calculated **iteratively** (nonrecursively) by using a `for` statement as follows:

```
factorial = 1;

for ( int counter = number; counter >= 1; counter-- )
    factorial *= counter;
```

A recursive definition of the factorial function is arrived at by observing the following relationship:

$$n! = n \cdot (n-1)!$$

For example, $5!$ is clearly equal to $5 * 4!$ as is shown by the following:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

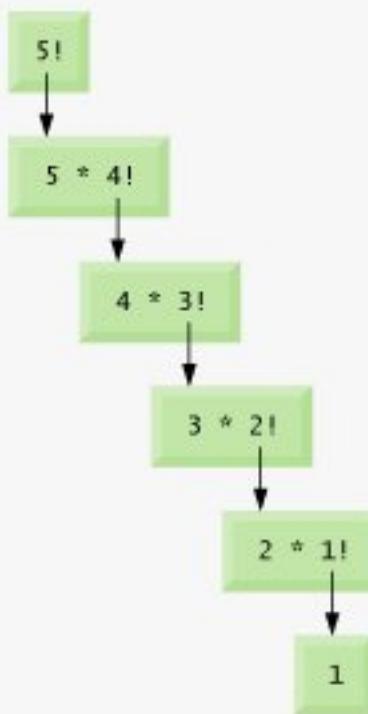
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

$$5! = 5 \cdot (4!)$$

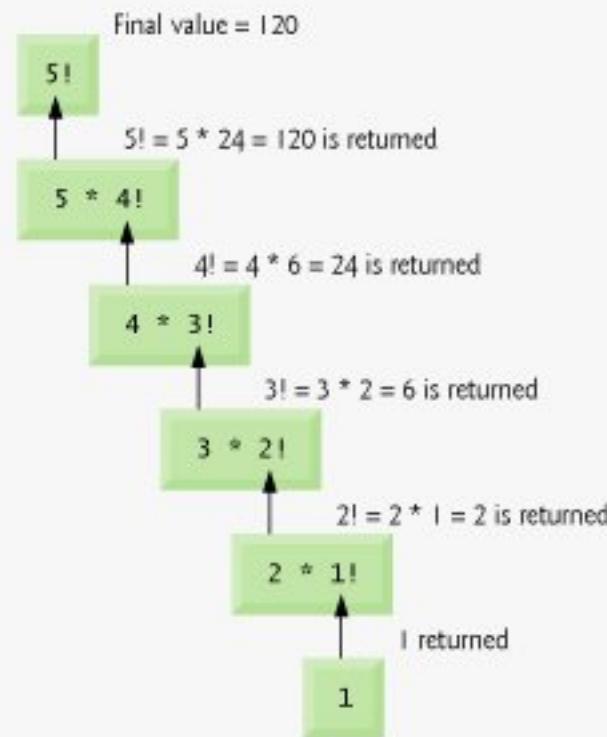
The evaluation of $5!$ would proceed as shown in Fig. 6.28. Figure 6.28(a) shows how the succession of recursive calls proceeds until $1!$ is evaluated to be 1, which terminates the recursion. Figure 6.28(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.

Figure 6.28. Recursive evaluation of $5!$.

[View full size image]



(a) Progression of recursive calls.



(b) Values returned from each recursive call.

The program of Fig. 6.29 uses recursion to calculate and print the factorials of the integers 010. (The choice of the data type `unsigned long` is explained momentarily.) The recursive function `factorial` (lines 2329) first determines whether the terminating condition `number <= 1` (line 25) is true. If `number` is indeed less than or equal to 1, function `factorial` returns 1 (line 26), no further recursion is necessary and the function terminates. If `number` is greater than 1, line 28 expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`. Note that `factorial(number - 1)` is a slightly simpler problem than the original calculation `factorial(number)`.

Figure 6.29. Demonstrating function `factorial`.

(This item is displayed on page 291 in the print version)

```

1 // Fig. 6.29: fig06_29.cpp
2 // Testing the recursive factorial function.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 unsigned long factorial( unsigned long ); // function prototype
11
12 int main()
13 {
14     // calculate the factorials of 0 through 10
15     for ( int counter = 0; counter <= 10; counter++ )
16         cout << setw( 2 ) << counter << " ! = " << factorial( counter )
17         << endl;
18
19     return 0; // indicates successful termination
20 } // end main
21
22 // recursive definition of function factorial
23 unsigned long factorial( unsigned long number )
24 {
25     if ( number <= 1 ) // test for base case
26         return 1; // base cases: 0! = 1 and 1! = 1
27     else // recursion step
28         return number * factorial( number - 1 );
29 } // end function factorial

```

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

```

Function `factorial` has been declared to receive a parameter of type `unsigned long` and return a result of type `unsigned long`. This is shorthand notation for `unsigned long int`. The C++ standard document requires that a variable of type `unsigned long int` be stored in at least four bytes (32 bits); thus, it can hold a value in the range 0 to at least 4294967295. (The data type `long int` is also stored in at least four bytes and can hold a value at least in the range 2147483648 to 2147483647.) As can be seen in Fig. 6.29, factorial values become large quickly. We chose the data type `unsigned long` so that the program can calculate factorials greater than 7! on computers with small (such as two-byte) integers. Unfortunately, function `factorial` produces large values so quickly that even `unsigned long` does not help us compute many factorial values before even the size of an `unsigned long` variable is exceeded.

[Page 291]

[Page 292]

The exercises explore using variables of data type `double` to calculate factorials of larger numbers. This points to a weakness in most programming languages, namely, that the languages are not easily extended to handle the unique requirements of various applications. As we will see when we discuss object-oriented programming in more depth, C++ is an extensible language that allows us to create classes that can represent arbitrarily large integers if we wish. Such classes already are available in popular class libraries,^[3] and we work on similar classes of our own in [Exercise 9.14](#) and [Exercise 11.5](#).

^[3] Such classes can be found at shoup.net/ntl, cliodhna.cop.uop.edu/~hetrick/c-sources.html and www.trumphurst.com/cpllibs/datapage.phtml?category='intro'.



Either omitting the base case, or writing the recursion step incorrectly so that it does not converge on the base case, causes "infinite" recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

[◀ PREV](#)

page footer

[NEXT ▶](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 293]

This figure raises some interesting issues about the order in which C++ compilers will evaluate the operands of operators. This is a separate issue from the order in which operators are applied to their operands, namely, the order dictated by the rules of operator precedence and associativity. [Figure 6.31](#) shows that evaluating `fibonacci(3)` causes two recursive calls, namely, `fibonacci(2)` and `fibonacci(1)`. But in what order are these calls made?

[Page 294]

Most programmers simply assume that the operands are evaluated left to right. The C++ language does not specify the order in which the operands of most operators (including +) are to be evaluated. Therefore, the programmer must make no assumption about the order in which these calls execute. The calls could in fact execute `fibonacci(2)` first, then `fibonacci(1)`, or they could execute in the reverse order: `fibonacci(1)`, then `fibonacci(2)`. In this program and in most others, it turns out that the final result would be the same. However, in some programs the evaluation of an operand can have **side effects** (changes to data values) that could affect the final result of the expression.

The C++ language specifies the order of evaluation of the operands of only four operatorsnamely, `&&`, `||`, the comma `(,)` operator and `? :`. The first three are binary operators whose two operands are guaranteed to be evaluated left to right. The last operator is C++'s only ternary operator. Its leftmost operand is always evaluated first; if it evaluates to nonzero (true), the middle operand evaluates next and the last operand is ignored; if the leftmost operand evaluates to zero (false), the third operand evaluates next and the middle operand is ignored.

Common Programming Error 6.25



Writing programs that depend on the order of evaluation of the operands of operators other than `&&`, `||`, `? :` and the comma `(,)` operator can lead to logic errors.

Portability Tip 6.3



Programs that depend on the order of evaluation of the operands of operators other than `&&`, `||`, `? :` and the comma `(,)` operator can function differently on systems with different compilers.

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each level of recursion in function `fibonacci` has a doubling effect on the number of function calls; i.e., the number of recursive calls that are required to calculate the n th Fibonacci number is on the order of 2^n . This rapidly gets out of hand. Calculating only the 20th Fibonacci number would require on the order of 2^{20} or about a million calls, calculating the 30th Fibonacci number would require on the order of 2^{30} or about a billion calls, and so on. Computer scientists refer to this as **exponential complexity**. Problems of this nature humble even the world's most powerful computers! Complexity issues in general, and exponential complexity in particular, are discussed in detail in the upper-level computer science course generally called "Algorithms."

[Page 295]

Performance Tip 6.8



Avoid Fibonacci-style recursive programs that result in an exponential "explosion" of calls.

PREV

NEXT

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 296]

Recursion has many negatives. It repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space. Each recursive call causes another copy of the function (actually only the function's variables) to be created; this can consume considerable memory. Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted. So why choose recursion?

Software Engineering Observation 6.18



Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution is not apparent.

[Page 297]

Performance Tip 6.9



Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

Common Programming Error 6.26



Accidentally having a nonrecursive function call itself, either directly or indirectly (through another function), is a logic error.

Most programming textbooks introduce recursion much later than we have done here. We feel that recursion is a sufficiently rich and complex topic that it is better to introduce it earlier and spread the examples over the remainder of the text. [Figure 6.33](#) summarizes the recursion examples and exercises in the text.

Figure 6.33. Summary of recursion examples and exercises in the text.

[Page 298]

Location in Text	Recursion Examples and Exercises
Chapter 6	
Section 6.19, Fig. 6.29	Factorial function
Section 6.19, Fig. 6.30	Fibonacci function
Exercise 6.7	Sum of two integers
Exercise 6.40	Raising an integer to an integer power
Exercise 6.42	Towers of Hanoi
Exercise 6.44	Visualizing recursion
Exercise 6.45	Greatest common divisor
Exercise 6.50, Exercise 6.51	Mystery "What does this program do?" exercise
Chapter 7	
Exercise 7.18	Mystery "What does this program do?" exercise
Exercise 7.21	Mystery "What does this program do?" exercise
Exercise 7.31	Selection sort
Exercise 7.32	Determine whether a string is a palindrome
Exercise 7.33	Linear search
Exercise 7.34	Binary search
Exercise 7.35	Eight Queens
Exercise 7.36	Print an array
Exercise 7.37	Print a string backward
Exercise 7.38	Minimum value in an array

Chapter 8	
Exercise 8.24	Quicksort
Exercise 8.25	Maze traversal
Exercise 8.26	Generating Mazes Randomly
Exercise 8.27	Mazes of Any Size
Chapter 20	
Section 20.3.3, Figs. 20.520.7	Mergesort
Exercise 20.8	Linear search
Exercise 20.9	Binary search
Exercise 20.10	Quicksort
Chapter 21	
Section 21.7, Figs. 21.2021.22	Binary tree insert
Section 21.7, Figs. 21.2021.22	Preorder traversal of a binary tree
Section 21.7, Figs. 21.2021.22	Inorder traversal of a binary tree
Section 21.7, Figs. 21.2021.22	Postorder traversal of a binary tree
Exercise 21.20	Print a linked list backward
Exercise 21.21	Search a linked list
Exercise 21.22	Binary tree delete
Exercise 21.25	Printing tree

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 299]

We can derive many of the operations of each class by examining the key verbs and verb phrases in the requirements document. We then relate each of these to particular classes in our system ([Fig. 6.34](#)). The verb phrases in [Fig. 6.34](#) help us determine the operations of each class.

Figure 6.34. Verbs and verb phrases for each class in the ATM system.

Class	Verbs and verb phrases
ATM	executes financial transactions
BalanceInquiry	[none in the requirements document]
Withdrawal	[none in the requirements document]
Deposit	[none in the requirements document]
BankDatabase	authenticates a user, retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
Account	retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
Screen	displays a message to the user
Keypad	receives numeric input from the user
CashDispenser	dispenses cash, indicates whether it contains enough cash to satisfy a withdrawal request
DepositSlot	receives a deposit envelope

Modeling Operations

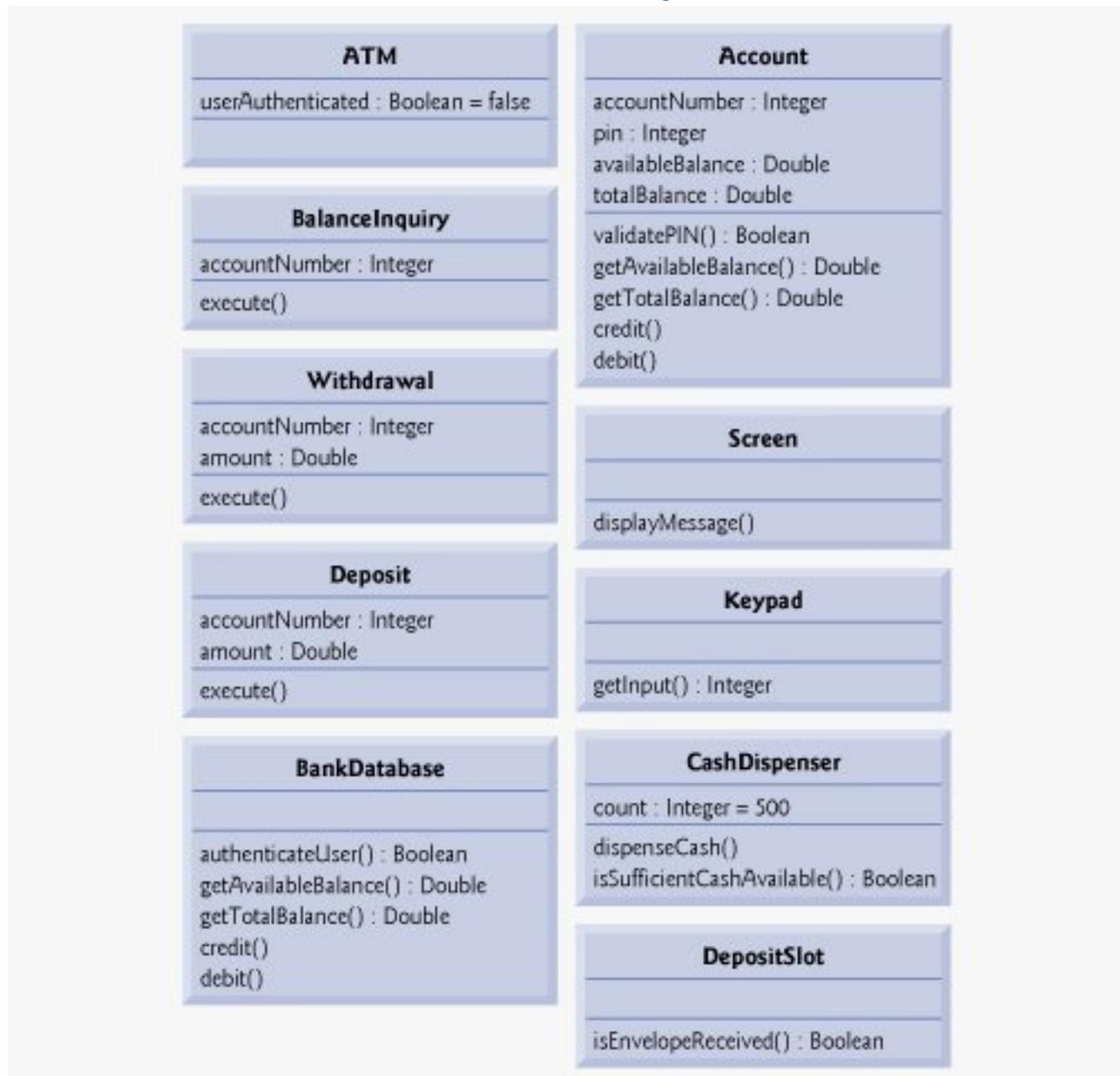
To identify operations, we examine the verb phrases listed for each class in [Fig. 6.34](#). The "executes financial transactions" phrase associated with class ATM implies that class ATM instructs transactions to execute. Therefore, classes BalanceInquiry, Withdrawal and Deposit each need an operation to provide this service to the ATM. We place this operation (which we have named `execute`) in the third

compartment of the three transaction classes in the updated class diagram of Fig. 6.35. During an ATM session, the ATM object will invoke the execute operation of each transaction object to tell it to execute.

Figure 6.35. Classes in the ATM system with attributes and operations.

(This item is displayed on page 300 in the print version)

[View full size image]



The UML represents operations (which are implemented as member functions in C++) by listing the operation name, followed by a comma-separated list of parameters in parentheses, a colon and the return type:

`operationName (parameter1, parameter2, ..., parameterN) : return type`

Each parameter in the comma-separated parameter list consists of a parameter name, followed by a colon and the parameter type:

```
parameterName : parameterType
```

For the moment, we do not list the parameters of our operations we will identify and model the parameters of some of the operations shortly. For some of the operations, we do not yet know the return types, so we also omit them from the diagram. These omissions are perfectly normal at this point. As our design and implementation proceed, we will add the remaining return types.

[Page 300]

Operations of Class BankDatabase and Class Account

[Figure 6.34](#) lists the phrase "authenticates a user" next to class `BankDatabase`—the database is the object that contains the account information necessary to determine whether the account number and PIN entered by a user match those of an account held at the bank. Therefore, class `BankDatabase` needs an operation that provides an authentication service to the ATM. We place the operation `authenticateUser` in the third compartment of class `BankDatabase` ([Fig. 6.35](#)). However, an object of class `Account`, not class `BankDatabase`, stores the account number and PIN that must be accessed to authenticate a user, so class `Account` must provide a service to validate a PIN obtained through user input against a PIN stored in an `Account` object. Therefore, we add a `validatePIN` operation to class `Account`. Note that we specify a return type of `Boolean` for the `authenticateUser` and `validatePIN` operations. Each operation returns a value indicating either that the operation was successful in performing its task (i.e., a return value of `true`) or that it was not (i.e., a return value of `false`).

[Page 301]

[Figure 6.34](#) lists several additional verb phrases for class `BankDatabase`: "retrieves an account balance," "credits a deposit amount to an account" and "debits a withdrawal amount from an account." Like "authenticates a user," these remaining phrases refer to services that the database must provide to the ATM, because the database holds all the account data used to authenticate a user and perform ATM transactions. However, objects of class `Account` actually perform the operations to which these phrases refer. Thus, we assign an operation to both class `BankDatabase` and class `Account` to correspond to each of these phrases. Recall from [Section 3.11](#) that, because a bank account contains sensitive information, we do not allow the ATM to access accounts directly. The database acts as an intermediary between the ATM and the account data, thus preventing unauthorized access. As we will see in [Section 7.12](#), class `ATM` invokes the operations of class `BankDatabase`, each of which in turn invokes the operation with the same name in class `Account`.

The phrase "retrieves an account balance" suggests that classes `BankDatabase` and `Account` each need a `getBalance` operation. However, recall that we created two attributes in class `Account` to represent a `balanceavailableBalance` and `totalBalance`. A balance inquiry requires access to both balance attributes so that it can display them to the user, but a withdrawal needs to check only the value of `availableBalance`. To allow objects in the system to obtain each balance attribute individually, we add operations `getAvailableBalance` and `getTotalBalance` to the third compartment of classes `BankDatabase` and `Account` (Fig. 6.35). We specify a return type of `Double` for each of these operations, because the balance attributes which they retrieve are of type `Double`.

The phrases "credits a deposit amount to an account" and "debits a withdrawal amount from an account" indicate that classes `BankDatabase` and `Account` must perform operations to update an account during a deposit and withdrawal, respectively. We therefore assign `credit` and `debit` operations to classes `BankDatabase` and `Account`. You may recall that crediting an account (as in a deposit) adds an amount only to the `totalBalance` attribute. Debiting an account (as in a withdrawal), on the other hand, subtracts the amount from both balance attributes. We hide these implementation details inside class `Account`. This is a good example of encapsulation and information hiding.

If this were a real ATM system, classes `BankDatabase` and `Account` would also provide a set of operations to allow another banking system to update a user's account balance after either confirming or rejecting all or part of a deposit. Operation `confirmDepositAmount`, for example, would add an amount to the `availableBalance` attribute, thus making deposited funds available for withdrawal. Operation `rejectDepositAmount` would subtract an amount from the `totalBalance` attribute to indicate that a specified amount, which had recently been deposited through the ATM and added to the `totalBalance`, was not found in the deposit envelope. The bank would invoke this operation after determining either that the user failed to include the correct amount of cash or that any checks did not clear (i.e., they "bounced"). While adding these operations would make our system more complete, we do not include them in our class diagrams or our implementation because they are beyond the scope of the case study.

Operations of Class Screen

Class `Screen` "displays a message to the user" at various times in an ATM session. All visual output occurs through the screen of the ATM. The requirements document describes many types of messages (e.g., a welcome message, an error message, a thank-you message) that the screen displays to the user. The requirements document also indicates that the screen displays prompts and menus to the user. However, a prompt is really just a message describing what the user should input next, and a menu is essentially a type of prompt consisting of a series of messages (i.e., menu options) displayed consecutively. Therefore, rather than assign class `Screen` an individual operation to display each type of message, prompt and menu, we simply create one operation that can display any message specified by a parameter. We place this operation (`displayMessage`) in the third compartment of class `Screen` in our class diagram (Fig. 6.35). Note that we do not worry about the parameter of this operation at this time—we model the parameter later in this section.

Operations of Class Keypad

From the phrase "receives numeric input from the user" listed by class `Keypad` in Fig. 6.34, we conclude that class `Keypad` should perform a `getInput` operation. Because the ATM's keypad, unlike a computer keyboard, contains only the numbers 09, we specify that this operation returns an integer value. Recall from the requirements document that in different situations the user may be required to enter a different type of number (e.g., an account number, a PIN, the number of a menu option, a deposit amount as a number of cents). Class `Keypad` simply obtains a numeric value for a client of the class it does not determine whether the value meets any specific criteria. Any class that uses this operation must verify that the user enters appropriate numbers, and if not, display error messages via class `Screen`). [Note: When we implement the system, we simulate the ATM's keypad with a computer keyboard, and for simplicity we assume that the user does not enter nonnumeric input using keys on the computer keyboard that do not appear on the ATM's keypad. Later in the book, you will learn how to examine inputs to determine if they are of particular types.]

Operations of Class CashDispenser and Class Depositslot

Figure 6.34 lists "dispenses cash" for class `CashDispenser`. Therefore, we create operation `dispenseCash` and list it under class `CashDispenser` in Fig. 6.35. Class `CashDispenser` also "indicates whether it contains enough cash to satisfy a withdrawal request." Thus, we include `isSufficientCashAvailable`, an operation that returns a value of UML type `Boolean`, in class `CashDispenser`. Figure 6.34 also lists "receives a deposit envelope" for class `Depositslot`. The deposit slot must indicate whether it received an envelope, so we place an operation `isEnvelopeReceived`, which returns a `Boolean` value, in the third compartment of class `Depositslot`. [Note: A real hardware deposit slot would most likely send the ATM a signal to indicate that an envelope was received. We simulate this behavior, however, with an operation in class `Depositslot` that class `ATM` can invoke to find out whether the deposit slot received an envelope.]

Operations of Class ATM

We do not list any operations for class `ATM` at this time. We are not yet aware of any services that class `ATM` provides to other classes in the system. When we implement the system with C++ code, however, operations of this class, and additional operations of the other classes in the system, may emerge.

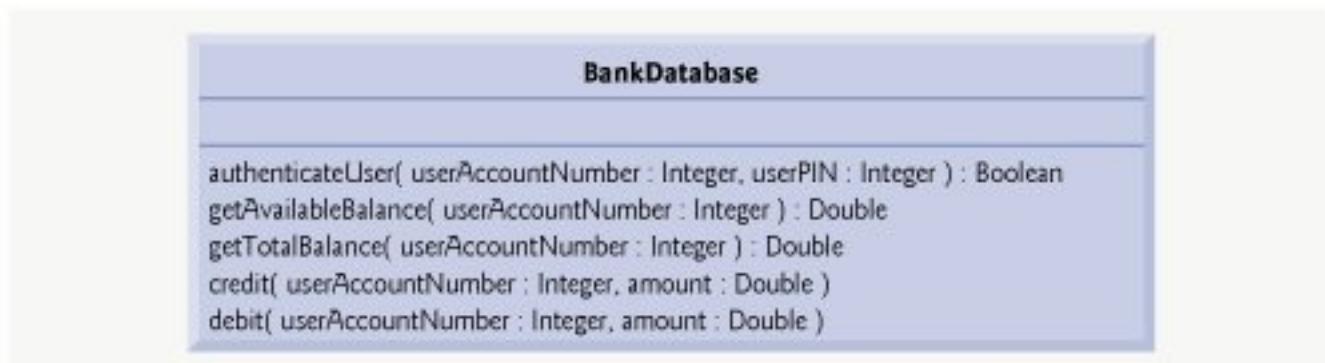
Identifying and Modeling Operation Parameters

So far, we have not been concerned with the parameters of our operationswe have attempted to gain only a basic understanding of the operations of each class. Let's now take a closer look at some operation parameters. We identify an operation's parameters by examining what data the operation requires to perform its assigned task.

Consider the `authenticateUser` operation of class `BankDatabase`. To authenticate a user, this operation must know the account number and PIN supplied by the user. Thus we specify that operation `authenticateUser` takes integer parameters `userAccountNumber` and `userPIN`, which the operation must compare to the account number and PIN of an `Account` object in the database. We prefix these parameter names with "user" to avoid confusion between the operation's parameter names and the attribute names that belong to class `Account`. We list these parameters in the class diagram in [Fig. 6.36](#) that models only class `BankDatabase`. [Note: It is perfectly normal to model only one class in a class diagram. In this case, we are most concerned with examining the parameters of this one class in particular, so we omit the other classes. In class diagrams later in the case study, in which parameters are no longer the focus of our attention, we omit these parameters to save space. Remember, however, that the operations listed in these diagrams still have parameters.]

Figure 6.36. Class `BankDatabase` with operation parameters.

[View full size image]



Recall that the UML models each parameter in an operation's comma-separated parameter list by listing the parameter name, followed by a colon and the parameter type (in UML notation). [Figure 6.36](#) thus specifies that operation `authenticateUser` takes two parameters `userAccountNumber` and `userPIN`, both of type `Integer`. When we implement the system in C++, we will represent these parameters with `int` values.

Class `BankDatabase` operations `getAvailableBalance`, `getTotalBalance`, `credit` and `debit` also each require a `userAccountNumber` parameter to identify the account to which the database must apply the operations, so we include these parameters in the class diagram of [Fig. 6.36](#). In addition, operations `credit` and `debit` each require a `Double` parameter `amount` to specify the amount of money to be credited or debited, respectively.

[Page 304]

The class diagram in [Fig. 6.37](#) models the parameters of class `Account`'s operations. Operation `validatePIN` requires only a `userPIN` parameter, which contains the user-specified PIN to be compared with the PIN associated with the account. Like their counterparts in class `BankDatabase`, operations `credit` and `debit` in class `Account` each require a `Double` parameter `amount` that

indicates the amount of money involved in the operation. Operations `getAvailableBalance` and `getTotalBalance` in class `Account` require no additional data to perform their tasks. Note that class `Account`'s operations do not require an account number parameter—each of these operations can be invoked only on a specific `Account` object, so including a parameter to specify an `Account` is unnecessary.

Figure 6.37. Class Account with operation parameters.

(This item is displayed on page 303 in the print version)

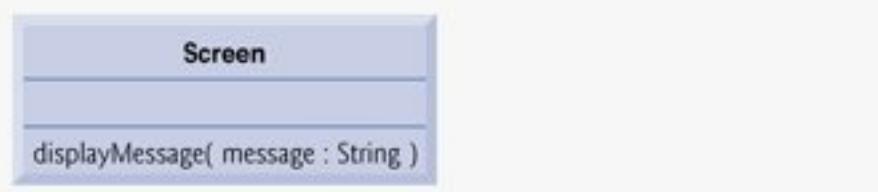
[View full size image]



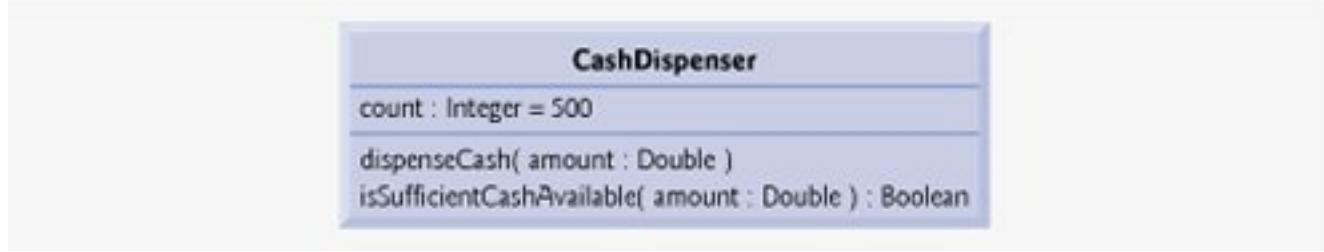
Figure 6.38 models class `Screen` with a parameter specified for operation `displayMessage`. This operation requires only a `String` parameter `message` that indicates the text to be displayed. Recall that the parameter types listed in our class diagrams are in UML notation, so the `String` type listed in Fig. 6.38 refers to the UML type. When we implement the system in C++, we will in fact use a C++ `string` object to represent this parameter.

Figure 6.38. Class Screen with operation parameters.

[View full size image]



The class diagram in Fig. 6.39 specifies that operation `dispenseCash` of class `CashDispenser` takes a `Double` parameter `amount` to indicate the amount of cash (in dollars) to be dispensed. Operation `isSufficientCashAvailable` also takes a `Double` parameter `amount` to indicate the amount of cash in question.

Figure 6.39. Class CashDispenser with operation parameters.[\[View full size image\]](#)

Note that we do not discuss parameters for operation execute of classes BalanceInquiry, Withdrawal and Deposit, operation getInquiry of class Keypad and operation isEnvelopeReceived of class DepositsSlot. At this point in our design process, we cannot determine whether these operations require additional data to perform their tasks, so we leave their parameter lists empty. As we progress through the case study, we may decide to add parameters to these operations.

In this section, we have determined many of the operations performed by the classes in the ATM system. We have identified the parameters and return types of some of the operations. As we continue our design process, the number of operations belonging to each class may varywe might find that new operations are needed or that some current operations are unnecessaryand we might determine that some of our class operations need additional parameters and different return types.

[Page 305]

Software Engineering Case Study Self-Review Exercises

6.1 Which of the following is not a behavior?

a.

reading data from a file

b.

printing output

c.

text output

obtaining input from the user

- 6.2** If you were to add to the ATM system an operation that returns the amount attribute of class `Withdrawal`, how and where would you specify this operation in the class diagram of Fig. 6.35?
- 6.3** Describe the meaning of the following operation listing that might appear in a class diagram for an object-oriented design of a calculator:

```
add( x : Integer, y : Integer ) : Integer
```

Answers to Software Engineering Case Study Self-Review Exercises

- 6.1** c.
- 6.2** To specify an operation that retrieves the amount attribute of class `Withdrawal`, the following operation would be placed in the operation (i.e., third) compartment of class `Withdrawal`:
- ```
getAmount() : Double
```
- 6.3** This is an operation named `add` that takes integers `x` and `y` as parameters and returns an integer value.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 305 (continued)]

## 6.23. Wrap-Up

In this chapter, you learned more about the details of function declarations. Functions have different pieces, such as the function prototype, function signature, function header and function body. You learned about argument coercion, or the forcing of arguments to the appropriate types specified by the parameter declarations of a function. We demonstrated how to use functions `rand` and `srand` to generate sets of random numbers that can be used for simulations. You also learned about the scope of variables, or the portion of a program where an identifier can be used. Two different ways to pass arguments to functions were covered—pass-by-value and pass-by-reference. For pass-by-reference, references are used as an alias to a variable. You learned that multiple functions in one class can be overloaded by providing functions with the same name and different signatures. Such functions can be used to perform the same or similar tasks, using different types or different numbers of parameters. We then demonstrated a simpler way of overloading functions using function templates, where a function is defined once but can be used for several different types. You were then introduced to the concept of recursion, where a function calls itself to solve a problem.

In [Chapter 7](#), you will learn how to maintain lists and tables of data in arrays. You will see a more elegant array-based implementation of the dice-rolling application and two enhanced versions of our `GradeBook` case study that you studied in [Chapters 35](#) that will use arrays to store the actual grades entered.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 307]

- An enumeration, introduced by the keyword `enum` and followed by a type name, 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.
- An identifier's storage class determines the period during which that identifier exists in memory.
- An identifier's scope is where the identifier can be referenced in a program.
- An identifier's linkage determines whether an identifier is known only in the source file where it is declared or across multiple files that are compiled, then linked together.
- 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.
- Only local variables of a function can be of automatic storage class.
- The storage class specifier `auto` explicitly declares variables of automatic storage class. Local variables are of automatic storage class by default, so keyword `auto` rarely is used.
- 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.
- 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.
- 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.
- An identifier declared outside any function or class has file scope.
- Labels 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.
- 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.
- The only identifiers with function-prototype scope are those used in the parameter list of a function prototype.
- Stacks are known as last-in, first-out (LIFO) data structures—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.
- One of the most important mechanisms for computer science students to understand is the function call stack (sometimes referred to as the program execution stack). This data structure supports the function call/return mechanism.
- The function call stack also supports the creation, maintenance and destruction of each called function's automatic variables.

- Each time a function calls another function, 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, as well as the automatic variables for the function call.
- 

[Page 308]

- The stack frame exists as long as the called function is active. When the called 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.
- In C++, an empty parameter list is specified by writing either `void` or nothing in parentheses.
- 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 to avoid a function call.
- 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.
  - With pass-by-reference, the caller gives the called function the ability to access the caller's data directly and to modify it if the called function chooses to do so.
- 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.
- 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.
- It is not uncommon for a program to invoke a function repeatedly with the same argument value for a particular parameter. In such cases, the programmer can specify that such a parameter has a default argument, i.e., a default value to be passed to that parameter.
- When a program omits an argument for a parameter with a default argument, the compiler rewrites the function call and inserts the default value of that argument to be passed as an argument to the function call.
- Default arguments must be the rightmost (trailing) arguments in a function's parameter list.
- Default arguments should be specified with the first occurrence of the function name typically, in the function prototype.
- C++ provides the unary scope resolution operator (`:::`) to access a global variable when a local variable of the same name is in scope.
- C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters. This capability is called function overloading.
- When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call.
- Overloaded functions are distinguished by their signatures.
- The compiler encodes each function identifier with the number and types of its parameters to enable type-safe linkage. Type-safe linkage ensures that the proper overloaded function is called and that the types of the arguments conform to the types of the parameters.
- Overloaded functions are normally used to perform similar operations that involve different program logic on different data types. If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently using function templates.

- The programmer writes a single function template definition. Given the argument types provided in calls to this function, C++ automatically generates separate function template specializations to handle each type of call appropriately. Thus, defining a single function template essentially defines a family of overloaded functions.

---

[Page 309]

- All function template definitions begin with the `template` keyword followed by a template parameter list to the function template enclosed in angle brackets (`<` and `>`).
- The formal type parameters are placeholders for fundamental types or user-defined types. These placeholders are used to specify the types of the function's parameters, to specify the function's return type and to declare variables within the body of the function definition.
- A recursive function is a function that calls itself, either directly or indirectly.
- A recursive function knows how to solve only the simplest case(s), or so-called base case(s). If the function is called with a base case, the function simply returns a result.
- If the function is called with a more complex problem, the function typically divides the problem into two conceptual piecesa piece that the function knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version of it.
- In order for the recursion to eventually terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller and smaller problems must eventually converge on the base case.
- The ratio of successive Fibonacci numbers converges on a constant value of 1.618.... This number frequently occurs in nature and has been called the golden ratio or the golden mean.
- Iteration and recursion have many similarities: both are based on a control statement, involve repetition, involve a termination test, gradually approach termination and can occur infinitely.
- Recursion has many negatives. It repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space. Each recursive call causes another copy of the function (actually only the function's variables) to be created; this can consume considerable memory.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 310]

infinite recursion

initializing a reference

inline function

`inline` keyword

inner block

integral size limits

invoke a method

iteration

iterative solution

label

LIFO (last-in, first-out)

linkage

"lowest type"

mandatory function prototypes

mangled function name

methods

mixed-type expression

modularizing a program with functions

mutable storage-class specifier

name decoration

name mangling

name of a variable

namespace scope

nested blocks

numerical data type limits

optimizing compiler

out of scope

outer block

overloading

parameter

pass-by-reference

pass-by-value

pop off a stack

"prepackaged" functions

principle of least privilege

procedure

program execution stack

programmer-defined function

promotion rules

pseudorandom numbers

push onto a stack

`rand` function

`RAND_MAX` symbolic constant

random number

randomizing

recursion

recursion overhead

recursion step

recursive call

recursive evaluation

recursive function

recursive solution

reference parameter

reference to a constant

reference to an automatic variable

`register` storage-class specifier

repeatability of function `rand`

returning a reference from a function

rightmost (trailing) arguments

scaling

scaling factor

scope of an identifier

seed

seed function `rand`

sequence of random numbers

shift a range of numbers

shifted, scaled integers

shifting value

side effect of an expression

signature

software reuse

`srand` function

stack

stack frame

stack overflow

`static` keyword

`static` local variable

static storage class

Terminology

static storage-class specifier

storage class

storage-class specifiers

template definition

template function

template keyword

template parameter list

terminating condition terminating right brace () of a block

termination test

truncate fractional part of a double

type name (enumerations)

type of a variable

type parameter

type-safe linkage

unary scope resolution operator (::)

user-defined function

user-defined type

validate a function call

void return type

width of random number range

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 312]

### 6.3

Write a program that tests whether the examples of the math library function calls shown in Fig. 6.2 actually produce the indicated results.

### 6.4

Give the function header for each of the following functions:

a.

Function `hypotenuse` that takes two double-precision, floating-point arguments, `side1` and `side2`, and returns a double-precision, floating-point result.

b.

Function `smallest` that takes three integers, `x`, `y` and `z`, and returns an integer.

c.

Function `instructions` that does not receive any arguments and does not return a value. [Note: Such functions are commonly used to display instructions to a user.]

d.

Function `intToDouble` that takes an integer argument, `number`, and returns a double-precision, floating-point result.

### 6.5

Give the function prototype for each of the following:

a.

The function described in Exercise 6.4(a).

b.

The function described in [Exercise 6.4\(b\)](#).

c.

The function described in [Exercise 6.4\(c\)](#).

d.

The function described in [Exercise 6.4\(d\)](#).

## 6.6

Write a declaration for each of the following:

a.

Integer `count` that should be maintained in a register. Initialize `count` to 0.

b.

Double-precision, floating-point variable `lastVal` that is to retain its value between calls to the function in which it is defined.

## 6.7

Find the error in each of the following program segments, and explain how the error can be corrected (see also [Exercise 6.53](#)):

---

[Page 313]

a.

```
int g(void)
{
 cout << "Inside function g" << endl;
 int h(void)
 {
 cout << "Inside function h" << endl;
 }
}
```

b.

```
int sum(int x, int y)
{
 int result;

 result = x + y;
}
```

**c.**

```
int sum(int n)
{
 if (n == 0)
 return 0;
 else
 n + sum(n - 1);
}
```

**d.**

```
void f (double a);
{
 float a;
 cout << a << endl;
}
```

**e.**

```
void product(void)
{
 int a;
 int b;
 int c;
 int result;
 cout << "Enter three integers: ";
 cin >> a >> b >> c;
 result = a * b * c;
 cout << "Result is " << result;
 return result;
}
```

## 6.8

Why would a function prototype contain a parameter type declaration such as `double &`?

## 6.9

(True/False) All arguments to function calls in C++ are passed by value.

## 6.10

Write a complete program that prompts the user for the radius of a sphere, and calculates and prints the volume of that sphere. Use an inline function `sphereVolume` that returns the result of the following expression: `( 4.0 / 3.0 ) * 3.14159 * pow( radius, 3 )`.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 314]

## 6.2

- a) block scope. b) block scope. c) file scope. d) file scope. e) file scope. f) function-prototype scope.

## 6.3

See the following program:

```

1 // Exercise 6.3: Ex06_03.cpp
2 // Testing the math library functions.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include <cmath>
12 using namespace std;
13
14 int main()
15 {
16 cout << fixed << setprecision(1);
17
18 cout << "sqrt(" << 900.0 << ") = " << sqrt(900.0)
19 << "\nsqrt(" << 9.0 << ") = " << sqrt(9.0);
20 cout << "\nexp(" << 1.0 << ") = " << setprecision(6)
21 << exp(1.0) << "\nexp(" << setprecision(1) << 2.0
22 << ") = " << setprecision(6) << exp(2.0);
23 cout << "\nlog(" << 2.718282 << ") = " << setprecision(1)
24 << log(2.718282)
25 << "\nlog(" << setprecision(6) << 7.389056 << ") = "
26 << setprecision(1) << log(7.389056);
27 cout << "\nlog10(" << 1.0 << ") = " << log10(1.0)
28 << "\nlog10(" << 10.0 << ") = " << log10(10.0)
29 << "\nlog10(" << 100.0 << ") = " << log10(100.0);
30 cout << "\nfabs(" << 13.5 << ") = " << fabs(13.5)
31 << "\nfabs(" << 0.0 << ") = " << fabs(0.0)
32 << "\nfabs(" << -13.5 << ") = " << fabs(-13.5);
33 cout << "\nceil(" << 9.2 << ") = " << ceil(9.2)

```

```

34 << "\nceil(" << -9.8 << ") = " << ceil(-9.8);
35 cout << "\nfloor(" << 9.2 << ") = " << floor(9.2)
36 << "\nfloor(" << -9.8 << ") = " << floor(-9.8);
37 cout << "\npow(" << 2.0 << " , " << 7.0 << ") = "
38 << pow(2.0, 7.0) << "\npow(" << 9.0 << " , "
39 << 0.5 << ") = " << pow(9.0, 0.5);
40 cout << setprecision(3) << "\nfmod("
41 << 13.675 << " , " << 2.333 << ") = "
42 << fmod(13.675, 2.333) << setprecision(1);
43 cout << "\nsin(" << 0.0 << ") = " << sin(0.0);
44 cout << "\ncos(" << 0.0 << ") = " << cos(0.0);
45 cout << "\ntan(" << 0.0 << ") = " << tan(0.0) << endl;
46 return 0; // indicates successful termination
47 } // end main

```

[Page 315]

```

sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(13.675, 2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0

```

a.

```
double hypotenuse(double side1, double side2)
```

b.

```
int smallest(int x, int y, int z)
```

c.

```
void instructions(void) // in C++ (void) can be written ()
```

d.

```
double intToDouble(int number)
```

## 6.5

a.

```
double hypotenuse(double, double);
```

b.

```
int smallest(int, int, int);
```

c.

```
void instructions(void); // in C++ (void) can be written ()
```

d.

```
double intToDouble(int);
```

## 6.6

a.

```
register int count = 0;
```

b.

```
static double lastVal;
```

## 6.7

a.

Error: Function h is defined in function g.

Correction: Move the definition of h out of the definition of g.

b.

Error: The function is supposed to return an integer, but does not.

Correction: Delete variable result and place the following statement in the function:

```
return x + y;
```

c.

Error: The result of n + sum( n - 1 ) is not returned; sum returns an improper result.

Correction: Rewrite the statement in the else clause as

```
return n + sum(n - 1);
```

d.

Errors: Semicolon after the right parenthesis that encloses the parameter list, and redefining the parameter a in the function definition.

Corrections: Delete the semicolon after the right parenthesis of the parameter list, and delete the declaration float a;.

e.

Error: The function returns a value when it is not supposed to.

Correction: Eliminate the return statement.

## 6.8

This creates a reference parameter of type "reference to double" that enables the function to modify the original variable in the calling function.

---

[Page 316]

## 6.9

False. C++ enables pass-by-reference using reference parameters (and pointers, as we discuss in Chapter 8).

## 6.10

See the following program:

```
1 // Exercise 6.10 Solution: Ex06_10.cpp
2 // Inline function that calculates the volume of a sphere.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include <cmath>
9 using std::pow;
10
11 const double PI = 3.14159; // define global constant PI
12
13 // calculates volume of a sphere
14 inline double sphereVolume(const double radius)
15 {
16 return 4.0 / 3.0 * PI * pow(radius, 3);
17 } // end inline function sphereVolume
18
19 int main()
20 {
21 double radiusValue;
22
23 // prompt user for radius
24 cout << "Enter the length of the radius of your sphere: ";
25 cin >> radiusValue; // input radius
26
27 // use radiusValue to calculate volume of sphere and display result
28 cout << "Volume of sphere with radius " << radiusValue
29 << " is " << sphereVolume(radiusValue) << endl;
30 return 0; // indicates successful termination
31 } // end main
```

 PREV

NEXT 

## page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 317]

| Car   | Hours | Charge |
|-------|-------|--------|
| 1     | 1.5   | 2.00   |
| 2     | 4.0   | 2.50   |
| 3     | 24.0  | 10.00  |
| TOTAL | 29.5  | 14.50  |

### 6.13

An application of function `floor` is rounding a value to the nearest integer. The statement

```
y = floor(x + .5);
```

rounds the number `x` to the nearest integer and assigns the result to `y`. Write a program that reads several numbers and uses the preceding statement to round each of these numbers to the nearest integer. For each number processed, print both the original number and the rounded number.

### 6.14

Function `floor` can be used to round a number to a specific decimal place. The statement

```
y = floor(x * 10 + .5) / 10;
```

rounds `x` to the tenths position (the first position to the right of the decimal point). The statement

```
y = floor(x * 100 + .5) / 100;
```

rounds `x` to the hundredths position (the second position to the right of the decimal point). Write a program that defines four functions to round a number `x` in various ways:

a.

```
roundToInteger(number)
```

b.

```
roundToTenths(number)
```

c.

```
roundToHundredths(number)
```

d.

```
roundToThousandths(number)
```

For each value read, your program should print the original value, the number rounded to the nearest integer, the number rounded to the nearest tenth, the number rounded to the nearest hundredth and the number rounded to the nearest thousandth.

## 6.15

Answer each of the following questions:

a.

What does it mean to choose numbers "at random?"

b.

Why is the `xrand` function useful for simulating games of chance?

c.

Why would you randomize a program by using `srand`? Under what circumstances is it desirable not to randomize?

d.

Why is it often necessary to scale or shift the values produced by `xrand`?

e.

Why is computerized simulation of real-world situations a useful technique?

## 6.16

Write statements that assign random integers to the variable n in the following ranges:

a.

$$1 \leq n \leq 2$$

b.

$$1 \leq n \leq 100$$

c.

$$0 \leq n \leq 9$$

d.

$$1000 \leq n \leq 1112$$

e.

$$1 \leq n \leq 1$$

f.

$$3 \leq n \leq 11$$

## 6.17

For each of the following sets of integers, write a single statement that prints a number at random from the set:

a.

2, 4, 6, 8, 10.

b.

3, 5, 7, 9, 11.

c.

6, 10, 14, 18, 22.

---

[Page 318]

### 6.18

Write a function `integerPower` (`base, exponent`) that returns the value of

`base exponent`

For example, `integerPower( 3, 4 ) = 3 * 3 * 3 * 3`. Assume that exponent is a positive, nonzero integer and that base is an integer. The function `integerPower` should use `for` or `while` to control the calculation. Do not use any math library functions.

### 6.19

(Hypotenuse) Define a function `hypotenuse` that calculates the length of the hypotenuse of a right triangle when the other two sides are given. Use this function in a program to determine the length of the hypotenuse for each of the triangles shown below. The function should take two double arguments and return the hypotenuse as a double.

| Triangle | Side 1 | Side 2 |
|----------|--------|--------|
| 1        | 3.0    | 4.0    |
| 2        | 5.0    | 12.0   |
| 3        | 8.0    | 15.0   |

### 6.20

Write a function `multiple` that determines for a pair of integers whether the second is a multiple of the first. The function should take two integer arguments and return `true` if the second is a multiple of the first, `false` otherwise. Use this function in a program that inputs a series of pairs of integers.

### 6.21

Write a program that inputs a series of integers and passes them one at a time to function `even`, which uses the modulus operator to determine whether an integer is even. The function should take an integer argument and return `true` if the integer is even and `false` otherwise.

**6.22**

Write a function that displays at the left margin of the screen a solid square of asterisks whose side is specified in integer parameter `side`. For example, if `side` is 4, the function displays the following:

```



```

**6.23**

Modify the function created in [Exercise 6.22](#) to form the square out of whatever character is contained in character parameter `fillCharacter`. Thus, if `side` is 5 and `fillCharacter` is `#`, then this function should print the following:

```


#####
```

**6.24**

Use techniques similar to those developed in [Exercise 6.22](#) and [Exercise 6.23](#) to produce a program that graphs a wide range of shapes.

---

[Page 319]

**6.25**

Write program segments that accomplish each of the following:

a.

Calculate the integer part of the quotient when integer `a` is divided by integer `b`.

Calculate the integer remainder when integer  $a$  is divided by integer  $b$ .

c.

Use the program pieces developed in (a) and (b) to write a function that inputs an integer between 1 and 32767 and prints it as a series of digits, each pair of which is separated by two spaces. For example, the integer 4562 should print as follows:

```
4 5 6 2
```

## 6.26

Write a function that takes the time as three integer arguments (hours, minutes and seconds) and returns the number of seconds since the last time the clock "struck 12." Use this function to calculate the amount of time in seconds between two times, both of which are within one 12-hour cycle of the clock.

## 6.27

(Celsius and Fahrenheit Temperatures) Implement the following integer functions:

a.

Function `celsius` returns the Celsius equivalent of a Fahrenheit temperature.

b.

Function `fahrenheit` returns the Fahrenheit equivalent of a Celsius temperature.

c.

Use these functions to write a program that prints charts showing the Fahrenheit equivalents of all Celsius temperatures from 0 to 100 degrees, and the Celsius equivalents of all Fahrenheit temperatures from 32 to 212 degrees. Print the outputs in a neat tabular format that minimizes the number of lines of output while remaining readable.

## 6.28

Write a program that inputs three double-precision, floating-point numbers and passes them to a function that returns the smallest number.

**6.29**

(Perfect Numbers) An integer is said to be a perfect number if the sum of its factors, including 1 (but not the number itself), is equal to the number. For example, 6 is a perfect number, because  $6 = 1 + 2 + 3$ . Write a function `perfect` that determines whether parameter `number` is a perfect number. Use this function in a program that determines and prints all the perfect numbers between 1 and 1000. Print the factors of each perfect number to confirm that the number is indeed perfect. Challenge the power of your computer by testing numbers much larger than 1000.

**6.30**

(PrimeNumbers) An integer is said to be prime if it is divisible by only 1 and itself. For example, 2, 3, 5 and 7 are prime, but 4, 6, 8 and 9 are not.

**a.**

Write a function that determines whether a number is prime.

**b.**

Use this function in a program that determines and prints all the prime numbers between 2 and 10,000. How many of these numbers do you really have to test before being sure that you have found all the primes?

**c.**

Initially, you might think that  $n/2$  is the upper limit for which you must test to see whether a number is prime, but you need only go as high as the square root of  $n$ . Why? Rewrite the program, and run it both ways. Estimate the performance improvement.

**6.31**

(Reverse Digits) Write a function that takes an integer value and returns the number with its digits reversed. For example, given the number 7631, the function should return 1367.

**6.32**

The greatest common divisor (GCD) of two integers is the largest integer that evenly divides each of the numbers. Write a function `gcd` that returns the greatest common divisor of two integers.

**6.33**

Write a function `qualityPoints` that inputs a student's average and returns 4 if a student's average is 90100, 3 if the average is 8089, 2 if the average is 7079, 1 if the average is 6069 and 0 if the average is

lower than 60.

### 6.34

Write a program that simulates coin tossing. For each toss of the coin, the program should print Heads or Tails. Let the program toss the coin 100 times and count the number of times each side of the coin appears. Print the results. The program should call a separate function `flip` that takes no arguments and returns 0 for tails and 1 for heads. [Note: If the program realistically simulates the coin tossing, then each side of the coin should appear approximately half the time.]

---

[Page 320]

### 6.35

(Computers in Education) Computers are playing an increasing role in education. Write a program that helps an elementary school student learn multiplication. Use `rand` to produce two positive one-digit integers. It should then type a question such as

How much is 6 times 7?

The student then types the answer. Your program checks the student's answer. If it is correct, print "Very good!", then ask another multiplication question. If the answer is wrong, print "No. Please try again.", then let the student try the same question repeatedly until the student finally gets it right.

### 6.36

(Computer Assisted Instruction) The use of computers in education is referred to as computer-assisted instruction (CAI). One problem that develops in CAI environments is student fatigue. This can be eliminated by varying the computer's dialogue to hold the student's attention. Modify the program of [Exercise 6.35](#) so the various comments are printed for each correct answer and each incorrect answer as follows:

Responses to a correct answer

Very good!  
Excellent!  
Nice work!  
Keep up the good work!

Responses to an incorrect answer

No. Please try again.  
Wrong. Try once more.

Don't give up!  
No. Keep trying.

Use the random number generator to choose a number from 1 to 4 to select an appropriate response to each answer. Use a switch statement to issue the responses.

### 6.37

More sophisticated computer-aided instruction systems monitor the student's performance over a period of time. The decision to begin a new topic often is based on the student's success with previous topics. Modify the program of [Exercise 6.36](#) to count the number of correct and incorrect responses typed by the student. After the student types 10 answers, your program should calculate the percentage of correct responses. If the percentage is lower than 75 percent, your program should print "Please ask your instructor for extra help" and terminate.

### 6.38

(Guess the Number Game) Write a program that plays the game of "guess the number" as follows: Your program chooses the number to be guessed by selecting an integer at random in the range 1 to 1000. The program then displays the following:

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

The player then types a first guess. The program responds with one of the following:

1. Excellent! You guessed the number!  
Would you like to play again (y or n)?
2. Too low. Try again.
3. Too high. Try again.

If the player's guess is incorrect, your program should loop until the player finally gets the number right. Your program should keep telling the player Too high or Too low to help the player "zero in" on the

correct answer.

### 6.39

Modify the program of [Exercise 6.38](#) to count the number of guesses the player makes. If the number is 10 or fewer, print "Either you know the secret or you got lucky!" If the player guesses the number in 10 tries, then print "Ahah! You know the secret!" If the player makes more than 10 guesses, then print "You should be able to do better!" Why should it take no more than 10 guesses? Well, with each "good guess" the player should be able to eliminate half of the numbers. Now show why any number from 1 to 1000 can be guessed in 10 or fewer tries.

### 6.40

Write a recursive function `power( base, exponent )` that, when invoked, returns

```
base exponent
```

For example, `power( 3, 4 ) = 3 * 3 * 3 * 3`. Assume that `exponent` is an integer greater than or equal to 1. Hint: The recursion step would use the relationship

```
base exponent = base · base exponent - 1
```

and the terminating condition occurs when `exponent` is equal to 1, because

```
base1 = base
```

### 6.41

(Fibonacci Series) The Fibonacci series

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

begins with the terms 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms. (a) Write a nonrecursive function `fibonacci( n )` that calculates the nth Fibonacci number. (b) Determine the largest `int` Fibonacci number that can be printed on your system. Modify the program of part (a) to use `double` instead of `int` to calculate and return Fibonacci numbers, and use this modified program to repeat part (b).

### 6.42

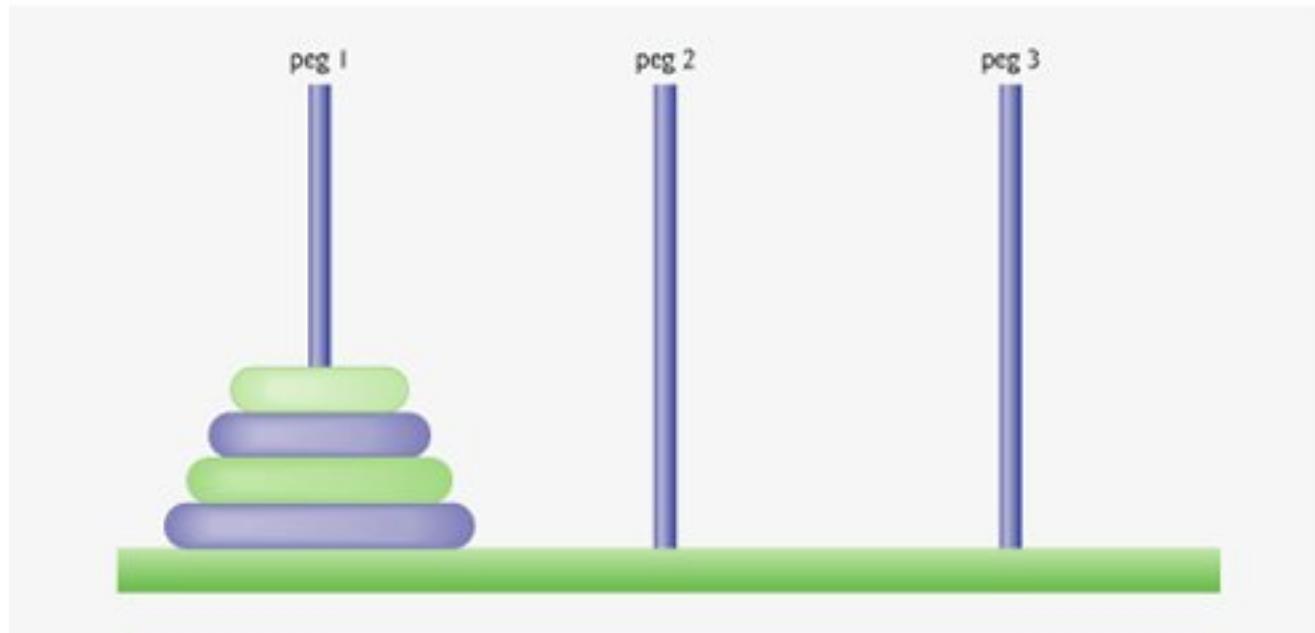
(Towers of Hanoi) In this chapter, you studied functions that can be easily implemented both recursively and iteratively. In this exercise, we present a problem whose recursive solution demonstrates the elegance of recursion, and whose iterative solution may not be as apparent.

The [Towers of Hanoi](#) is one of the most famous classic problems every budding computer scientist must grapple with. Legend has it that in a temple in the Far East, priests are attempting to move a stack of golden disks from one diamond peg to another ([Fig. 6.41](#)). The initial stack has 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from one peg to another under the constraints that exactly one disk is moved at a time and at no time may a larger disk be placed above a smaller disk. Three pegs are provided, one being used for temporarily holding disks. Supposedly, the world will end when the priests complete their task, so there is little incentive for us to facilitate their efforts.

**Figure 6.41. Towers of Hanoi for the case with four disks.**

(This item is displayed on page 322 in the print version)

[\[View full size image\]](#)



Let us assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that prints the precise sequence of peg-to-peg disk transfers.

If we were to approach this problem with conventional methods, we would rapidly find ourselves hopelessly knotted up in managing the disks. Instead, attacking this problem with recursion in mind allows the steps to be simple. Moving  $n$  disks can be viewed in terms of moving only  $n-1$  disks (hence, the recursion), as follows:

a.

Move  $n-1$  disks from peg 1 to peg 2, using peg 3 as a temporary holding area.

b.

Move the last disk (the largest) from peg 1 to peg 3.

c.

Move the  $n - 1$  disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

---

[Page 322]

The process ends when the last task involves moving  $n = 1$  disk (i.e., the base case). This task is accomplished by simply moving the disk, without the need for a temporary holding area.

Write a program to solve the Towers of Hanoi problem. Use a recursive function with four parameters:

a.

The number of disks to be moved

b.

The peg on which these disks are initially threaded

c.

The peg to which this stack of disks is to be moved

d.

The peg to be used as a temporary holding area

Your program should print the precise instructions it will take to move the disks from the starting peg to the destination peg. For example, to move a stack of three disks from peg 1 to peg 3, your program should print the following series of moves:

1 → 3 (This means move one disk from peg 1 to peg 3.)

1 → 2

3 → 2

1 → 3

2 → 1

2 → 3

1 → 3

**6.43**

Any program that can be implemented recursively can be implemented iteratively, although sometimes with more difficulty and less clarity. Try writing an iterative version of the Towers of Hanoi. If you succeed, compare your iterative version with the recursive version developed in [Exercise 6.42](#). Investigate issues of performance, clarity and your ability to demonstrate the correctness of the programs.

**6.44**

(Visualizing Recursion) It is interesting to watch recursion "in action." Modify the factorial function of [Fig. 6.29](#) to print its local variable and recursive call parameter. For each recursive call, display the outputs on a separate line and add a level of indentation. Do your utmost to make the outputs clear, interesting and meaningful. Your goal here is to design and implement an output format that helps a person understand recursion better. You may want to add such display capabilities to the many other recursion examples and exercises throughout the text.

**6.45**

(Recursive Greatest Common Divisor) The greatest common divisor of integers  $x$  and  $y$  is the largest integer that evenly divides both  $x$  and  $y$ . Write a recursive function `gcd` that returns the greatest common divisor of  $x$  and  $y$ , defined recursively as follows: If  $y$  is equal to 0, then  $\text{gcd}(x, y)$  is  $x$ ; otherwise,  $\text{gcd}(x, y)$  is  $\text{gcd}(y, x \% y)$ , where  $\%$  is the modulus operator. [Note: For this algorithm,  $x$  must be larger than  $y$ .]

[Page 323]

**6.46**

Can `main` be called recursively on your system? Write a program containing a function `main`. Include static local variable `count` and initialize it to 1. Postincrement and print the value of `count` each time `main` is called. Compile your program. What happens?

**6.47**

[Exercises 6.356.37](#) developed a computer-assisted instruction program to teach an elementary school student multiplication. This exercise suggests enhancements to that program.

**a.**

Modify the program to allow the user to enter a grade-level capability. A grade level of 1 means to use only single-digit numbers in the problems, a grade level of 2 means to use numbers as large as two digits, etc.

**b.**

Modify the program to allow the user to pick the type of arithmetic problems he or she wishes to study. An option of 1 means addition problems only, 2 means subtraction problems only, 3 means multiplication problems only, 4 means division problems only and 5 means a random mix of problems of all these types.

## 6.48

Write function `distance` that calculates the distance between two points  $(x_1, y_1)$  and  $(x_2, y_2)$ . All numbers and return values should be of type `double`.

## 6.49

What is wrong with the following program?

```

1 // Exercise 6.49: ex06_49.cpp
2 // What is wrong with this program?
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6
7 int main()
8 {
9 int c;
10
11 if ((c = cin.get()) != EOF)
12 {
13 main();
14 cout << c;
15 } // end if
16
17 return 0; // indicates successful termination
18 } // end main

```

## 6.50

What does the following program do?

```

1 // Exercise 6.50: ex06_50.cpp
2 // What does this program do?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int mystery(int, int); // function prototype
9

```

[Page 324]

```

10 int main()
11 {
12 int x, y;
13
14 cout << "Enter two integers: ";
15 cin >> x >> y;
16 cout << "The result is " << mystery(x, y) << endl;
17
18 return 0; // indicates successful termination
19 } // end main
20
21 // Parameter b must be a positive integer to prevent infinite recursion
22 int mystery(int a, int b)
23 {
24 if (b == 1) // base case
25 return a;
26 else // recursion step
27 return a + mystery(a, b - 1);
28 } // end function mystery

```

## 6.51

After you determine what the program of [Exercise 6.50](#) does, modify the program to function properly after removing the restriction that the second argument be nonnegative.

## 6.52

Write a program that tests as many of the math library functions in [Fig. 6.2](#) as you can. Exercise each of these functions by having your program print out tables of return values for a diversity of argument values.

## 6.53

Find the error in each of the following program segments and explain how to correct it:

**a.**

```
float cube(float); // function prototype

double cube(float number) // function definition
{
 return number * number * number;
}
```

**b.**

```
register auto int x = 7;
```

**c.**

```
int randomNumber = srand();
```

**d.**

```
float y = 123.45678;
int x;

x = y;
cout << static_cast < float > (x) << endl;
```

**e.**

```
double square(double number)
{
 double number;
 return number * number;
}
```

**f.**

```
int sum(int n)
{
 if (n == 0)
 return 0;
 else
 return n + sum(n);
}
```

**6.54**

Modify the craps program of Fig. 6.11 to allow wagering. Package as a function the portion of the program that runs one game of craps. Initialize variable bankBalance to 1000 dollars. Prompt the player to enter a wager. Use a while loop to check that wager is less than or equal to bankBalance and, if not, prompt the user to reenter wager until a valid wager is entered. After a correct wager is entered, run one game of craps. If the player wins, increase bankBalance by wager and print the new bankBalance. If the player loses, decrease bankBalance by wager, print the new bankBalance, check on whether bankBalance has become zero and, if so, print the message "Sorry. You busted!" As the game progresses, print various messages to create some "chatter" such as "Oh, you're going for broke, huh?", "Aw cmon, take a chance!" or "You're up big. Now's the time to cash in your chips!".

**6.55**

Write a C++ program that prompts the user for the radius of a circle then calls inline function circleArea to calculate the area of that circle.

**6.56**

Write a complete C++ program with the two alternate functions specified below, of which each simply triples the variable count defined in main. Then compare and contrast the two approaches. These two functions are

a.

function tripleByValue that passes a copy of count by value, triples the copy and returns the new value and

b.

function tripleByReference that passes count by reference via a reference parameter and triples the original value of count through its alias (i.e., the reference parameter).

**6.57**

What is the purpose of the unary scope resolution operator?

**6.58**

Write a program that uses a function template called min to determine the smaller of two arguments. Test the program using integer, character and floating-point number arguments.

**6.59**

Write a program that uses a function template called `max` to determine the largest of three arguments. Test the program using integer, character and floating-point number arguments.

**6.60**

Determine whether the following program segments contain errors. For each error, explain how it can be corrected. [Note: For a particular program segment, it is possible that no errors are present in the segment.]

**a.**

```
template < class A >
int sum(int num1, int num2, int num3)
{
 return num1 + num2 + num3;
}
```

**b.**

```
void printResults(int x, int y)
{
 cout << "The sum is " << x + y << '\n';
 return x + y;
}
```

**c.**

```
template < A >
A product(A num1, A num2, A num3)
{
 return num1 * num2 * num3;
}
```

**d.**

```
double cube(int);
int cube(int);
```



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 327]

## Outline

[7.1 Introduction](#)

[7.2 Arrays](#)

[7.3 Declaring Arrays](#)

[7.4 Examples Using Arrays](#)

[7.5 Passing Arrays to Functions](#)

[7.6 Case Study: Class GradeBook Using an Array to Store Grades](#)

[7.7 Searching Arrays with Linear Search](#)

[7.8 Sorting Arrays with Insertion Sort](#)

[7.9 Multidimensional Arrays](#)

[7.10 Case Study: Class GradeBook Using a Two-Dimensional Array](#)

[7.11 Introduction to C++ Standard Library Class Template `vector`](#)

[7.12 \(Optional\) Software Engineering Case Study: Collaboration Among Objects in the ATM System](#)

[7.13 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

## Exercises

### Recursion Exercises

### vector Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 327 (continued)]

## 7.1. Introduction

This chapter introduces the important topic of **data structures**—collections of related data items. Arrays are data structures consisting of related data items of the same type. You learned about classes in [Chapter 3](#). In [Chapter 9](#), we discuss the notion of **structures**. Structures and classes are each capable of holding related data items of possibly different types. Arrays, structures and classes are "static" entities in that they remain the same size throughout program execution. (They may, of course, be of automatic storage class and hence be created and destroyed each time the blocks in which they are defined are entered and exited.)

After discussing how arrays are declared, created and initialized, this chapter presents a series of practical examples that demonstrate several common array manipulations. We then explain how character strings (represented until now by `string` objects) can also be represented by character arrays. We present an example of searching arrays to find particular elements. The chapter also introduces one of the most important computing applications—sorting data (i.e., putting the data in some particular order). Two sections of the chapter enhance the case study of class `GradeBook` in [Chapters 36](#). In particular, we use arrays to enable the class to maintain a set of grades in memory and analyze student grades from multiple exams in a semester—two capabilities that were absent from previous versions of the `GradeBook` class. These and other chapter examples demonstrate the ways in which arrays allow programmers to organize and manipulate data.

The style of arrays we use throughout most of this chapter are C-style pointer-based arrays. (We will study pointers in [Chapter 8](#).) In the final section of this chapter, and in [Chapter 23](#), Standard Template Library (STL), we will cover arrays as full-fledged objects called vectors. We will discover that these object-based arrays are safer and more versatile than the C-like, pointer-based arrays we discuss in the early part of this chapter.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 329]

```
cout << c[0] + c[1] + c[2] << endl;
```

To divide the value of `c[ 6 ]` by 2 and assign the result to the variable `x`, we would write

```
x = c[6] / 2;
```

### Common Programming Error 7.1



It is important to note the difference between the "seventh element of the array" and "array element 7." Array subscripts begin at 0, so the "seventh element of the array" has a subscript of 6, while "array element 7" has a subscript of 7 and is actually the eighth element of the array. Unfortunately, this distinction frequently is a source of off-by-one errors. To avoid such errors, we refer to specific array elements explicitly by their array name and subscript number (e.g., `c[ 6 ]` or `c[ 7 ]`).

The brackets used to enclose the subscript of an array are actually an operator in C++. Brackets have the same level of precedence as parentheses. [Figure 7.2](#) shows the precedence and associativity of the operators introduced so far. Note that brackets (`[]`) have been added to the first row of [Fig. 7.2](#). The operators are shown top to bottom in decreasing order of precedence with their associativity and type.

**Figure 7.2. Operator precedence and associativity.**

| Operators |     |                                                 |   |   |  |  | Associativity | Type            |
|-----------|-----|-------------------------------------------------|---|---|--|--|---------------|-----------------|
| ( )       | [ ] |                                                 |   |   |  |  | left to right | highest         |
| ++        | --  | <code>static_cast&lt;type&gt;( operand )</code> |   |   |  |  | 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                |

 PREVNEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 330]

and the compiler reserves the appropriate amount of memory. The arraySize must be an integer constant greater than zero. For example, to tell the compiler to reserve 12 elements for integer array c, use the declaration

```
int c[12]; // c is an array of 12 integers
```

Memory can be reserved for several arrays with a single declaration. The following declaration reserves 100 elements for the integer array b and 27 elements for the integer array x.

```
int b[100], // b is an array of 100 integers
x[27]; // x is an array of 27 integers
```

## Good Programming Practice 7.1



We prefer to declare one array per declaration for readability, modifiability and ease of commenting.

Arrays can be declared to contain values of any non-reference data type. For example, an array of type `char` can be used to store a character string. Until now, we have used `string` objects to store character strings. [Section 7.4](#) introduces using character arrays to store strings. Character strings and their similarity to arrays (a relationship C++ inherited from C), and the relationship between pointers and arrays, are discussed in [Chapter 8](#).

[◀ PREV](#)

[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 332]

If the array size is omitted from a declaration with an initializer list, the compiler determines the number of elements in the array by counting the number of elements in the initializer list. For example,

```
int n[] = { 1, 2, 3, 4, 5 };
```

creates a five-element array.

If the array size and an initializer list are specified in an array declaration, the number of initializers must be less than or equal to the array size. The array declaration

```
int n[5] = { 32, 27, 64, 18, 95, 14 };
```

causes a compilation error, because there are six initializers and only five array elements.

## Common Programming Error 7.2



Providing more initializers in an array initializer list than there are elements in the array is a compilation error.

## Common Programming Error 7.3



Forgetting to initialize the elements of an array whose elements should be initialized is a logic error.

## Specifying an Array's Size with a Constant Variable and Setting Array Elements with Calculations

Figure 7.5 sets the elements of a 10-element array `s` to the even integers 2, 4, 6, ..., 20 (lines 1718) and prints the array in tabular format (lines 2024). These numbers are generated (line 18) by multiplying each successive value of the loop counter by 2 and adding 2.

**Figure 7.5. Generating values to be placed into elements of an array.**

```
1 // Fig. 7.5: fig07_05.cpp
2 // Set array s to the even integers from 2 to 20.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 // constant variable can be used to specify array size
13 const int arraySize = 10;
14
15 int s[arraySize]; // array s has 10 elements
16
17 for (int i = 0; i < arraySize; i++) // set the values
18 s[i] = 2 + 2 * i;
19
20 cout << "Element" << setw(13) << "Value" << endl;
21
22 // output contents of array s in tabular format
23 for (int j = 0; j < arraySize; j++)
24 cout << setw(7) << j << setw(13) << s[j] << endl;
25
26 return 0; // indicates successful termination
27 } // end main
```

| Element | Value |
|---------|-------|
| 0       | 2     |
| 1       | 4     |
| 2       | 6     |
| 3       | 8     |
| 4       | 10    |
| 5       | 12    |
| 6       | 14    |
| 7       | 16    |
| 8       | 18    |
| 9       | 20    |

Line 13 uses the `const` qualifier to declare a so-called **constant variable** `arraySize` with the value 10. Constant variables must be initialized with a constant expression when they are declared and cannot be modified thereafter (as shown in Fig. 7.6 and Fig. 7.7). Constant variables are also called **named constants** or **read-only variables**.

**Figure 7.6. Initializing and using a constant variable.**

(This item is displayed on page 334 in the print version)

```
1 // Fig. 7.6: fig07_06.cpp
2 // Using a properly initialized constant variable.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 const int x = 7; // initialized constant variable
10
11 cout << "The value of constant variable x is: " << x << endl;
12
13 return 0; // indicates successful termination
14 } // end main
```

The value of constant variable x is: 7

**Figure 7.7. `const` variables must be initialized.**

(This item is displayed on page 334 in the print version)

```

1 // Fig. 7.7: fig07_07.cpp
2 // A const variable must be initialized.
3
4 int main()
5 {
6 const int x; // Error: x must be initialized
7
8 x = 7; // Error: cannot modify a const variable
9
10 return 0; // indicates successful termination
11 } // end main

```

Borland C++ command-line compiler error message:

```

Error E2304 fig07_07.cpp 6: Constant variable 'x' must be initialized
in function main()
Error E2024 fig07_07.cpp 8: Cannot modify a const object in function main()

```

Microsoft Visual C++ .NET compiler error message:

```

C:\cpphttp5_examples\ch07\fig07_07.cpp(6) : error C2734: 'x' : const object
must be initialized if not extern
C:\cpphttp5_examples\ch07\fig07_07.cpp(8) : error C2166: l-value specifies
const object

```

GNU C++ compiler error message:

```

fig07_07.cpp:6: error: uninitialized const `x'
fig07_07.cpp:8: error: assignment of read-only variable `x'

```



Not assigning a value to a constant variable when it is declared is a compilation error.

## Common Programming Error 7.5



Assigning a value to a constant variable in an executable statement is a compilation error.

[Page 335]

Constant variables can be placed anywhere a constant expression is expected. In Fig. 7.5, constant variable `arraySize` specifies the size of array `s` in line 15.

## Common Programming Error 7.6



Only constants can be used to declare the size of automatic and static arrays. Not using a constant for this purpose is a compilation error.

Using constant variables to specify array sizes makes programs more **scalable**. In Fig. 7.5, the first `for` statement could fill a 1000-element array by simply changing the value of `arraySize` in its declaration from 10 to 1000. If the constant variable `arraySize` had not been used, we would have to change lines 15, 17 and 23 of the program to scale the program to handle 1000 array elements. As programs get larger, this technique becomes more useful for writing clearer, easier-to-modify programs.

## Software Engineering Observation 7.1



Defining the size of each array as a constant variable instead of a literal constant can make programs more scalable.

## Good Programming Practice 7.2



Defining the size of an array as a constant variable instead of a literal constant makes programs clearer. This technique eliminates so-called **magic numbers**. For example, repeatedly mentioning the size 10 in array-processing code for a 10-element array gives the number 10 an artificial significance and can unfortunately confuse the reader when the program includes other 10s that have nothing to do with the array size.

## Summing the Elements of an Array

Often, the elements of an array represent a series of values to be used in a calculation. For example, if the elements of an array represent exam grades, a professor may wish to total the elements of the array and use that sum to calculate the class average for the exam. The examples using class GradeBook later in the chapter, namely [Figs. 7.167.17](#) and [Figs. 7.237.24](#), use this technique.

The program in [Fig. 7.8](#) sums the values contained in the 10-element integer array `a`. The program declares, creates and initializes the array at line 10. The `for` statement (lines 1415) performs the calculations. The values being supplied as initializers for array `a` also could be read into the program from the user at the keyboard, or from a file on disk (see [Chapter 17](#), File Processing). For example, the `for` statement

```
for (int j = 0; j < arraySize; j++)
 cin >> a[j];
```

reads one value at a time from the keyboard and stores the value in element `a[ j ]`.

**Figure 7.8. Computing the sum of the elements of an array.**

(This item is displayed on pages 335 - 336 in the print version)

```

1 // Fig. 7.8: fig07_08.cpp
2 // Compute the sum of the elements of the array.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 const int arraySize = 10; // constant variable indicating size of array
10 int a[arraySize] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11 int total = 0;
12
13 // sum contents of array a
14 for (int i = 0; i < arraySize; i++)
15 total += a[i];

```

```
16 cout << "Total of array elements: " << total << endl;
17
18
19 return 0; // indicates successful termination
20 } // end main
```

```
Total of array elements: 849
```

[Page 336]

## Using Bar Charts to Display Array Data Graphically

Many programs present data to users in a graphical manner. For example, numeric values are often displayed as bars in a bar chart. In such a chart, longer bars represent proportionally larger numeric values. One simple way to display numeric data graphically is with a bar chart that shows each numeric value as a bar of asterisks (\*).

Professors often like to examine the distribution of grades on an exam. A professor might graph the number of grades in each of several categories to visualize the grade distribution. Suppose the grades were 87, 68, 94, 100, 83, 78, 85, 91, 76 and 87. Note that there was one grade of 100, two grades in the 90s, four grades in the 80s, two grades in the 70s, one grade in the 60s and no grades below 60. Our next program (Fig. 7.9) stores this grade distribution data in an array of 11 elements, each corresponding to a category of grades. For example, `n[ 0 ]` indicates the number of grades in the range 09, `n[ 7 ]` indicates the number of grades in the range 7079 and `n[ 10 ]` indicates the number of grades of 100. The two versions of class `GradeBook` later in the chapter (Figs. 7.167.17 and Figs. 7.237.24) contain code that calculates these grade frequencies based on a set of grades. For now, we manually create the array by looking at the set of grades.

**Figure 7.9.** Bar chart printing program.

(This item is displayed on pages 336 - 337 in the print version)

```
1 // Fig. 7.9: fig07_09.cpp
2 // Bar chart printing program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 const int arraySize = 11;
13 int n[arraySize] = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
14
15 cout << "Grade distribution:" << endl;
16
17 // for each element of array n, output a bar of the chart
18 for (int i = 0; i < arraySize; i++)
19 {
20 // output bar labels ("0-9:", ..., "90-99:", "100:")
21 if (i == 0)
22 cout << " 0-9: ";
23 else if (i == 10)
24 cout << " 100: ";
25 else
26 cout << i * 10 << "-" << (i * 10) + 9 << ": ";
27
28 // print bar of asterisks
29 for (int stars = 0; stars < n[i]; stars++)
30 cout << '*';
31
32 cout << endl; // start a new line of output
33 } // end outer for
34
35 return 0; // indicates successful termination
36 } // end main
```

```

Grade distribution:
 0-9:
 10-19:
 20-29:
 30-39:
 40-49:
 50-59:
 60-69: *
 70-79: **
 80-89: ****
 90-99: **
 100: *

```

[Page 337]

The program reads the numbers from the array and graphs the information as a bar chart. The program displays each grade range followed by a bar of asterisks indicating the number of grades in that range. To label each bar, lines 2126 output a grade range (e.g., "70-79: ") based on the current value of counter variable *i*. The nested `for` statement (lines 2930) outputs the bars. Note the loop-continuation condition at line 29 (`stars < n[ i ]`). Each time the program reaches the inner `for`, the loop counts from 0 up to `n[ i ]`, thus using a value in array *n* to determine the number of asterisks to display. In this example, `n[ 0 ]n[ 5 ]` contain zeros because no students received a grade below 60. Thus, the program displays no asterisks next to the first six grade ranges.

### Common Programming Error 7.7



Although it is possible to use the same control variable in a `for` statement and a second `for` statement nested inside, this is confusing and can lead to logic errors.

[Page 338]

## Using the Elements of an Array as Counters

Sometimes, programs use counter variables to summarize data, such as the results of a survey. In Fig. 6.9, we used separate counters in our die-rolling program to track the number of occurrences of each side of a die as the program rolled the die 6,000,000 times. An array version of this program is shown in Fig. 7.10.

**Figure 7.10. Die-rolling program using an array instead of switch.**

```
1 // Fig. 7.10: fig07_10.cpp
2 // Roll a six-sided die 6,000,000 times.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib>
11 using std::rand;
12 using std::srand;
13
14 #include <ctime>
15 using std::time;
16
17 int main()
18 {
19 const int arraySize = 7; // ignore element zero
20 int frequency[arraySize] = { 0 };
21
22 srand(time(0)); // seed random number generator
23
24 // roll die 6,000,000 times; use die value as frequency index
25 for (int roll = 1; roll <= 6000000; roll++)
26 frequency[1 + rand() % 6]++;
27
28 cout << "Face" << setw(13) << "Frequency" << endl;
29
30 // output each array element's value
31 for (int face = 1; face < arraySize; face++)
32 cout << setw(4) << face << setw(13) << frequency[face]
33 << endl;
34
35 return 0; // indicates successful termination
36 } // end main
```

| Face | Frequency |
|------|-----------|
| 1    | 1000167   |
| 2    | 1000149   |
| 3    | 1000152   |
| 4    | 998748    |
| 5    | 999626    |
| 6    | 1001158   |

---

[Page 339]

Figure 7.10 uses the array `frequency` (line 20) to count the occurrences of each side of the die. The single statement in line 26 of this program replaces the `switch` statement in lines 3052 of Fig. 6.9. Line 26 uses a random value to determine which `frequency` element to increment during each iteration of the loop. The calculation in line 26 produces a random subscript from 1 to 6, so array `frequency` must be large enough to store six counters. However, we use a seven-element array in which we ignore `frequency[ 0 ]` it is more logical to have the die face value 1 increment `frequency[ 1 ]` than `frequency[ 0 ]`. Thus, each face value is used as a subscript for array `frequency`. We also replace lines 5661 of Fig. 6.9 by looping through array `frequency` to output the results (lines 3133).

## Using Arrays to Summarize Survey Results

Our next example (Fig. 7.11) uses arrays to summarize the results of data collected in a survey. Consider the following problem statement:

Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10 (1 meaning awful and 10 meaning excellent). Place the 40 responses in an integer array and summarize the results of the poll.

**Figure 7.11. Poll analysis program.**

(This item is displayed on pages 339 - 340 in the print version)

```
1 // Fig. 7.11: fig07_11.cpp
2 // Student poll program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 // define array sizes
13 const int responseSize = 40; // size of array responses
14 const int frequencySize = 11; // size of array frequency
15
16 // place survey responses in array responses
17 const int responses[responseSize] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
18 10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
19 5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
21 // initialize frequency counters to 0
22 int frequency[frequencySize] = { 0 };
23
24 // for each answer, select responses element and use that value
25 // as frequency subscript to determine element to increment
26 for (int answer = 0; answer < responseSize; answer++)
27 frequency[responses[answer]]++;
28
29 cout << "Rating" << setw(17) << "Frequency" << endl;
30
31 // output each array element's value
32 for (int rating = 1; rating < frequencySize; rating++)
33 cout << setw(6) << rating << setw(17) << frequency[rating]
34 << endl;
35
36 return 0; // indicates successful termination
37 } // end main
```

| Rating | Frequency |
|--------|-----------|
| 1      | 2         |
| 2      | 2         |
| 3      | 2         |
| 4      | 2         |
| 5      | 5         |
| 6      | 11        |
| 7      | 5         |
| 8      | 7         |
| 9      | 1         |
| 10     | 3         |

This is a typical array-processing application. We wish to summarize the number of responses of each type (i.e., 1 through 10). The array `responses` (lines 1719) is a 40-element integer array of the students' responses to the survey. Note that array `responses` is declared `const`, as its values do not (and should not) change. We use an 11-element array `frequency` (line 22) to count the number of occurrences of each response. Each element of the array is used as a counter for one of the survey responses and is initialized to zero. As in Fig. 7.10, we ignore `frequency[ 0 ]`.

---

[Page 340]

## Software Engineering Observation 7.2



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.

## Good Programming Practice 7.3



Strive for program clarity. It is sometimes worthwhile to trade off the most efficient use of memory or processor time in favor of writing clearer programs.

## Performance Tip 7.1



Sometimes performance considerations far outweigh clarity considerations.

The first `for` statement (lines 2627) takes the responses one at a time from the array `responses` and increments one of the 10 counters in the `frequency` array (`frequency[ 1 ]` to `frequency[ 10 ]`). The key statement in the loop is line 27, which increments the appropriate `frequency` counter, depending on the value of `responses[ answer ]`.

Let's consider several iterations of the `for` loop. When control variable `answer` is 0, the value of `responses[ answer ]` is the value of `responses[ 0 ]` (i.e., 1 in line 17), so the program interprets `frequency[ responses[ answer ] ]++` as

```
frequency[1]++
```

[Page 341]

which increments the value in array element 1. To evaluate the expression, start with the value in the innermost set of square brackets (`answer`). Once you know `answer`'s value (which is the value of the loop control variable in line 26), plug it into the expression and evaluate the next outer set of square brackets (i.e., `responses[ answer ]`, which is a value selected from the `responses` array in lines 1719). Then use the resulting value as the subscript for the `frequency` array to specify which counter to increment.

When `answer` is 1, `responses[ answer ]` is the value of `responses[ 1 ]`, which is 2, so the program interprets `frequency[ responses[ answer ] ]++` as

```
frequency[2]++
```

which increments array element 2.

When `answer` is 2, `responses[ answer ]` is the value of `responses[ 2 ]`, which is 6, so the program interprets `frequency[ responses[ answer ] ]++` as

```
frequency[6]++
```

which increments array element 6, and so on. Regardless of the number of responses processed in the survey, the program requires only an 11-element array (ignoring element zero) to summarize the results, because all the response values are between 1 and 10 and the subscript values for an 11-element array are 0 through 10.

If the data in the `responses` array had contained an invalid value, such as 13, the program would have attempted to add 1 to `frequency[ 13 ]`, which is outside the bounds of the array. C++ has no array bounds checking to prevent the computer from referring to an element that does not exist. Thus, an executing program can "walk off" either end of an array without warning. The programmer should ensure that all array references remain within the bounds of the array.

### Common Programming Error 7.8



Referring to an element outside the array bounds is an execution-time logic error.  
It is not a syntax error.

### Error-Prevention Tip 7.1



When looping through an array, the array subscript should never go below 0 and should always be less than the total number of elements in the array (one less than the size of the array). Make sure that the loop-termination condition prevents accessing elements outside this range.

### Portability Tip 7.1



The (normally serious) effects of referencing elements outside the array bounds are system dependent. Often this results in changes to the value of an unrelated variable or a fatal error that terminates program execution.

C++ is an extensible language. [Section 7.11](#) presents C++ Standard Library class template `vector`, which enables programmers to perform many operations that are not available for C++'s built-in arrays. For example, we will be able to compare `vectors` directly and assign one `vector` to another. In [Chapter 11](#), we extend C++ further by implementing an array as a user-defined class of our own. This new array definition will enable us to input and output entire arrays with `cin` and `cout`, initialize arrays when they are created, prevent access to out-of-range array elements and change the range of subscripts (and even their subscript type) so that the first element of an array is not required to be element 0. We will even be able to use noninteger subscripts.



In [Chapter 11](#), we will see how to develop a class representing a "smart array," which checks that all subscript references are in bounds at runtime. Using such smart data types helps eliminate bugs.

## Using Character Arrays to Store and Manipulate Strings

To this point, we have discussed only integer arrays. However, arrays may be of any type. We now introduce storing character strings in character arrays. Recall that, starting in [Chapter 3](#), we have been using `string` objects to store character strings, such as the course name in our `GradeBook` class. A string such as "hello" is actually an array of characters. While `string` objects are convenient to use and reduce the potential for errors, character arrays that represent strings have several unique features, which we discuss in this section. As you continue your study of C++, you may encounter C++ capabilities that require you to use character arrays in preference to `string` objects. You may also be asked to update existing code using character arrays.

A character array can be initialized using a string literal. For example, the declaration

```
char string1[] = "first";
```

initializes the elements of array `string1` to the individual characters in the string literal "first". The size of array `string1` in the preceding declaration is determined by the compiler based on the length of the string. It is important to note that the string "first" contains five characters plus a special string-termination character called the **null character**. Thus, array `string1` actually contains six elements. The character constant representation of the null character is '\0' (backslash followed by zero). All strings represented by character arrays end with this character. A character array representing a string should always be declared large enough to hold the number of characters in the string and the terminating null character.

Character arrays also can be initialized with individual character constants in an initializer list. The preceding declaration is equivalent to the more tedious form

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

Note the use of single quotes to delineate each character constant. Also, note that we explicitly provided the terminating null character as the last initializer value. Without it, this array would simply represent an array of characters, not a string. As we discuss in [Chapter 8](#), not providing a terminating null character for a string can cause logic errors.

Because a string is an array of characters, we can access individual characters in a string directly with array subscript notation. For example, `string1[ 0 ]` is the character 'f', `string1[ 3 ]` is the character 's' and `string1[ 5 ]` is the null character.

We also can input a string directly into a character array from the keyboard using `cin` and `>>`. For example, the declaration

```
char string2[20];
```

creates a character array capable of storing a string of 19 characters and a terminating null character. The statement

---

[Page 343]

```
cin >> string2;
```

reads a string from the keyboard into `string2` and appends the null character to the end of the string input by the user. Note that the preceding statement provides only the name of the array and no information about the size of the array. It is the programmer's responsibility to ensure that the array into which the string is read is capable of holding any string the user types at the keyboard. By default, `cin` reads characters from the keyboard until the first white-space character is encountered regardless of the array size. Thus, inputting data with `cin` and `>>` can insert data beyond the end of the array (see [Section 8.13](#) for information on preventing insertion beyond the end of a `char` array).

### Common Programming Error 7.9



Not providing `cin >>` with a character array large enough to store a string typed at the keyboard can result in loss of data in a program and other serious runtime errors.

A character array representing a null-terminated string can be output with `cout` and `<<`. The statement

```
cout << string2;
```

prints the array `string2`. Note that `cout <<`, like `cin >>`, does not care how large the character array is. The characters of the string are output until a terminating null character is encountered. [Note: `cin` and `cout` assume that character arrays should be processed as strings terminated by null characters; `cin` and `cout` do not provide similar input and output processing capabilities for other array types.]

[Figure 7.12](#) demonstrates initializing a character array with a string literal, reading a string into a character array, printing a character array as a string and accessing individual characters of a string.

### Figure 7.12. Character arrays processed as strings.

(This item is displayed on pages 343 - 344 in the print version)

```
1 // Fig. 7.12: fig07_12.cpp
2 // Treating character arrays as strings.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int main()
9 {
10 char string1[20]; // reserves 20 characters
11 char string2[] = "string literal"; // reserves 15 characters
12
13 // read string from user into array string1
14 cout << "Enter the string \"hello there\": ";
15 cin >> string1; // reads "hello" [space terminates input]
16
17 // output strings
18 cout << "string1 is: " << string1 << "\nstring2 is: " << string2;
19
20 cout << "\nstring1 with spaces between characters is:\n";
21
22 // output characters until null character is reached
23 for (int i = 0; string1[i] != '\0'; i++)
24 cout << string1[i] << ' ';
25
26 cin >> string1; // reads "there"
27 cout << "\nstring1 is: " << string1 << endl;
28
29 return 0; // indicates successful termination
30 } // end main
```

```
Enter the string "hello there": hello there
string1 is: hello
string2 is: string literal
string1 with spaces between characters is:
h e l l o
string1 is: there
```

Lines 2324 of Fig. 7.12 use a `for` statement to loop through the `string1` array and print the individual characters separated by spaces. The condition in the `for` statement, `string1[ i ] != '\0'`, is true until the loop encounters the terminating null character of the string.

## Static Local Arrays and Automatic Local Arrays

Chapter 6 discussed the storage class specifier `static`. A static local variable in a function definition exists for the duration of the program, but is visible only in the function body.

### Performance Tip 7.2



We can apply `static` to a local array declaration so that the array is not created and initialized each time the program calls the function and is not destroyed each time the function terminates in the program. This can improve performance, especially when using large arrays.

A program initializes static local arrays when their declarations are first encountered. If a static array is not initialized explicitly by the programmer, each element of that array is initialized to zero by the compiler when the array is created. Recall that C++ does not perform such default initialization for automatic variables.

Figure 7.13 demonstrates function `staticArrayInit` (lines 2541) with a static local array (line 28) and function `automaticArrayInit` (lines 4460) with an automatic local array (line 47).

### Figure 7.13. static array initialization and automatic array initialization.

(This item is displayed on pages 344 - 346 in the print version)

```
1 // Fig. 7.13: fig07_13.cpp
2 // Static arrays are initialized to zero.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void staticArrayInit(void); // function prototype
8 void automaticArrayInit(void); // function prototype
9
10 int main()
11 {
12 cout << "First call to each function:\n";
13 staticArrayInit();
14 automaticArrayInit();
```

```
15
16 cout << "\n\nSecond call to each function:\n";
17 staticArrayInit();
18 automaticArrayInit();
19 cout << endl;
20
21 return 0; // indicates successful termination
22 } // end main
23
24 // function to demonstrate a static local array
25 void staticArrayInit(void)
26 {
27 // initializes elements to 0 first time function is called
28 static int array1[3]; // static local array
29
30 cout << "\nValues on entering staticArrayInit:\n";
31
32 // output contents of array1
33 for (int i = 0; i < 3; i++)
34 cout << "array1[" << i << "] = " << array1[i] << " ";
35
36 cout << "\nValues on exiting staticArrayInit:\n";
37
38 // modify and output contents of array1
39 for (int j = 0; j < 3; j++)
40 cout << "array1[" << j << "] = " << (array1[j] += 5) << " ";
41 } // end function staticArrayInit
42
43 // function to demonstrate an automatic local array
44 void automaticArrayInit(void)
45 {
46 // initializes elements each time function is called
47 int array2[3] = { 1, 2, 3 }; // automatic local array
48
49 cout << "\n\nValues on entering automaticArrayInit:\n";
50
51 // output contents of array2
52 for (int i = 0; i < 3; i++)
53 cout << "array2[" << i << "] = " << array2[i] << " ";
54
55 cout << "\nValues on exiting automaticArrayInit:\n";
56
57 // modify and output contents of array2
58 for (int j = 0; j < 3; j++)
59 cout << "array2[" << j << "] = " << (array2[j] += 5) << " ";
60 } // end function automaticArrayInit
```

First call to each function:

```
Values on entering staticArrayInit:
array1[0] = 0 array1[1] = 0 array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5 array1[1] = 5 array1[2] = 5
```

```
Values on entering automaticArrayInit:
array2[0] = 1 array2[1] = 2 array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8
```

Second call to each function:

```
Values on entering staticArrayInit:
array1[0] = 5 array1[1] = 5 array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10 array1[1] = 10 array1[2] = 10
```

```
Values on entering automaticArrayInit:
array2[0] = 1 array2[1] = 2 array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8
```

[Page 346]

Function `staticArrayInit` is called twice (lines 13 and 17). The `static` local array is initialized to zero by the compiler the first time the function is called. The function prints the array, adds 5 to each element and prints the array again. The second time the function is called, the `static` array contains the modified values stored during the first function call. Function `automaticArrayInit` also is called twice (lines 14 and 18). The elements of the automatic local array are initialized (line 47) with the values 1, 2 and 3. The function prints the array, adds 5 to each element and prints the array again. The second time the function is called, the array elements are reinitialized to 1, 2 and 3. The array has automatic storage class, so the array is recreated during each call to `automaticArrayInit`.

## Common Programming Error 7.10



Assuming that elements of a function's local `static` array are initialized every time the function is called can lead to logic errors in a program.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 347]

C++ passes arrays to functions by reference—the called functions can modify the element values in the callers' original arrays. The value of the name of the array is the address in the computer's memory of the first element of the array. Because the starting address of the array is passed, the called function knows precisely where the array is stored in memory. Therefore, when the called function modifies array elements in its function body, it is modifying the actual elements of the array in their original memory locations.

### Performance Tip 7.3



Passing arrays by reference makes sense for performance reasons. If arrays were passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would be time consuming and would require considerable storage for the copies of the array elements.

### Software Engineering Observation 7.3



It is possible to pass an array by value (by using a simple trick we explain in [Chapter 22](#))—this is rarely done.

Although entire arrays are passed by reference, individual array elements are passed by value exactly as simple variables are. Such simple single pieces of data are called **scalars** or **scalar quantities**. To pass an element of an array to a function, use the subscripted name of the array element as an argument in the function call. In [Chapter 6](#), we showed how to pass scalars (i.e., individual variables and array elements) by reference with references. In [Chapter 8](#), we show how to pass scalars by reference with pointers.

For a function to receive an array through a function call, the function's parameter list must specify that the function expects to receive an array. For example, the function header for function `modifyArray` might be written as

```
void modifyArray(int b[], int arraySize)
```

indicating that `modifyArray` expects to receive the address of an array of integers in parameter `b` and the number of array elements in parameter `arraySize`. The size of the array is not required between the array brackets. If it is included, the compiler ignores it. Because C++ passes arrays to functions by reference, when the called function uses the array name `b`, it will in fact be referring to the actual array in

the caller (i.e., array `hourlyTemperatures` discussed at the beginning of this section).

Note the strange appearance of the function prototype for `modifyArray`

```
void modifyArray(int [], int);
```

This prototype could have been written

```
void modifyArray(int anyArrayName[], int anyVariableName);
```

[Page 348]

but, as we learned in [Chapter 3](#), C++ compilers ignore variable names in prototypes. Remember, the prototype tells the compiler the number of arguments and the type of each argument (in the order in which the arguments are expected to appear).

The program in [Fig. 7.14](#) demonstrates the difference between passing an entire array and passing an array element. Lines 2223 print the five original elements of integer array `a`. Line 28 passes `a` and its size to function `modifyArray` (lines 4550), which multiplies each of `a`'s elements by 2 (through parameter `b`). Then, lines 3233 print array `a` again in `main`. As the output shows, the elements of `a` are indeed modified by `modifyArray`. Next, line 36 prints the value of scalar `a[ 3 ]`, then line 38 passes element `a[ 3 ]` to function `modifyElement` (lines 5458), which multiplies its parameter by 2 and prints the new value. Note that when line 39 again prints `a[ 3 ]` in `main`, the value has not been modified, because individual array elements are passed by value.

**Figure 7.14. Passing arrays and individual array elements to functions.**

(This item is displayed on pages 348 - 349 in the print version)

```

1 // Fig. 7.14: fig07_14.cpp
2 // Passing arrays and individual array elements to functions.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 void modifyArray(int [], int); // appears strange
11 void modifyElement(int);
12
13 int main()
```

```

14 {
15 const int arraySize = 5; // size of array a
16 int a[arraySize] = { 0, 1, 2, 3, 4 }; // initialize array a
17
18 cout << "Effects of passing entire array by reference:"
19 << "\n\nThe values of the original array are:\n";
20
21 // output original array elements
22 for (int i = 0; i < arraySize; i++)
23 cout << setw(3) << a[i];
24
25 cout << endl;
26
27 // pass array a to modifyArray by reference
28 modifyArray(a, arraySize);
29 cout << "The values of the modified array are:\n";
30
31 // output modified array elements
32 for (int j = 0; j < arraySize; j++)
33 cout << setw(3) << a[j];
34
35 cout << "\n\nEffects of passing array element by value:"
36 << "\n\na[3] before modifyElement: " << a[3] << endl;
37
38 modifyElement(a[3]); // pass array element a[3] by value
39 cout << "a[3] after modifyElement: " << a[3] << endl;
40
41 return 0; // indicates successful termination
42 } // end main
43
44 // in function modifyArray, "b" points to the original array "a" in memory
45 void modifyArray(int b[], int sizeOfArray)
46 {
47 // multiply each array element by 2
48 for (int k = 0; k < sizeOfArray; k++)
49 b[k] *= 2;
50 } // end function modifyArray
51
52 // in function modifyElement, "e" is a local copy of
53 // array element a[3] passed from main
54 void modifyElement(int e)
55 {
56 // multiply parameter by 2
57 cout << "Value of element in modifyElement: " << (e *= 2) << endl;
58 } // end function modifyElement

```

Effects of passing entire array by reference:

The values of the original array are:

0 1 2 3 4

The values of the modified array are:

0 2 4 6 8

Effects of passing array element by value:

a[3] before modifyElement: 6

Value of element in modifyElement: 12

a[3] after modifyElement: 6

[Page 349]

There may be situations in your programs in which a function should not be allowed to modify array elements. C++ provides the type qualifier `const` that can be used to prevent modification of array values in the caller by code in a called function. When a function specifies an array parameter that is preceded by the `const` qualifier, the elements of the array become constant in the function body, and any attempt to modify an element of the array in the function body results in a compilation error. This enables the programmer to prevent accidental modification of array elements in the function's body.

[Figure 7.15](#) demonstrates the `const` qualifier. Function `tryToModifyArray` (lines 2126) is defined with parameter `const int b[]`, which specifies that array `b` is constant and cannot be modified. Each of the three attempts by the function to modify array `b`'s elements (lines 2325) results in a compilation error. The Microsoft Visual C++ .NET compiler, for example, produces the error "l-value specifies const object." [Note: The C++ standard defines an "object" as any "region of storage," thus including variables or array elements of fundamental data types as well as instances of classes (what we've been calling objects).] This message indicates that using a `const` object (e.g., `b[ 0 ]`) as an lvalue is an error you cannot assign a new value to a `const` object by placing it on the left of an assignment operator. Note that compiler error messages vary between compilers (as shown in [Fig. 7.15](#)). The `const` qualifier will be discussed again in [Chapter 10](#).

[Page 351]

**Figure 7.15. `const` type qualifier applied to an array parameter.**

(This item is displayed on page 350 in the print version)

```

1 // Fig. 7.15: fig07_15.cpp
2 // Demonstrating the const type qualifier.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void tryToModifyArray(const int []); // function prototype
8
9 int main()
10 {
11 int a[] = { 10, 20, 30 };
12
13 tryToModifyArray(a);
14 cout << a[0] << ' ' << a[1] << ' ' << a[2] << '\n';
15
16 return 0; // indicates successful termination
17 } // end main
18
19 // In function tryToModifyArray, "b" cannot be used
20 // to modify the original array "a" in main.
21 void tryToModifyArray(const int b[])
22 {
23 b[0] /= 2; // error
24 b[1] /= 2; // error
25 b[2] /= 2; // error
26 } // end function tryToModifyArray

```

Borland C++ command-line compiler error message:

```

Error E2024 fig07_15.cpp 23: Cannot modify a const object
in function tryToModifyArray(const int * const)
Error E2024 fig07_15.cpp 24: Cannot modify a const object
in function tryToModifyArray(const int * const)
Error E2024 fig07_15.cpp 25: Cannot modify a const object
in function tryToModifyArray(const int * const)

```

Microsoft Visual C++ .NET compiler error message:

```
C:\cpphttp5_examples\ch07\fig07_15.cpp(23) : error C2166: l-value specifies
 const object
C:\cpphttp5_examples\ch07\fig07_15.cpp(24) : error C2166: l-value specifies
 const object
C:\cpphttp5_examples\ch07\fig07_15.cpp(25) : error C2166: l-value specifies
 const object
```

GNU C++ compiler error message:

```
fig07_15.cpp:23: error: assignment of read-only location
fig07_15.cpp:24: error: assignment of read-only location
fig07_15.cpp:25: error: assignment of read-only location
```

## Common Programming Error 7.11



Forgetting that arrays in the caller are passed by reference, and hence can be modified in called functions, may result in logic errors.

## Software Engineering Observation 7.4



Applying the `const` type qualifier to an array parameter in a function definition to prevent the original array from being modified in the function body is another example of the principle of least privilege. Functions should not be given the capability to modify an array unless it is absolutely necessary.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 355]

Note that the size of the array in line 29 of Fig. 7.16 is specified by `public const static data member students` (declared in line 13). This data member is `public` so that it is accessible to the clients of the class. We will soon see an example of a client program using this constant. Declaring `students` with the `const` qualifier indicates that this data member is constant its value cannot be changed after being initialized. Keyword `static` in this variable declaration indicates that the data member is shared by all objects of the class all `GradeBook` objects store grades for the same number of students. Recall from Section 3.6 that when each object of a class maintains its own copy of an attribute, the variable that represents the attribute is also known as a data member each object (instance) of the class has a separate copy of the variable in memory. There are variables for which each object of a class does not have a separate copy. That is the case with `static` data members, which are also known as [class variables](#). When objects of a class containing `static` data members are created, all the objects of that class share one copy of the class's `static` data members. A `static` data member can be accessed within the class definition and the member-function definitions just like any other data member. As you will soon see, a `public static` data member can also be accessed outside of the class, even when no objects of the class exist, using the class name followed by the binary scope resolution operator (`::`) and the name of the data member. You will learn more about `static` data members in Chapter 10.

The class's constructor (declared in line 16 of Fig. 7.16 and defined in lines 1724 of Fig. 7.17) has two parameters the name of the course and an array of grades. When a program creates a `GradeBook` object (e.g., line 13 of `fig07_18.cpp`), the program passes an existing `int` array to the constructor, which copies the values in the passed array to the data member `grades` (lines 2223 of Fig. 7.17). The grade values in the passed array could have been input from a user or read from a file on disk (as discussed in Chapter 17, File Processing). In our test program, we simply initialize an array with a set of grade values (Fig. 7.18, lines 1011). Once the grades are stored in data member `grades` of class `GradeBook`, all the class's member functions can access the `grades` array as needed to perform various calculations.

### Figure 7.17. `GradeBook` class member functions manipulating an array of grades.

(This item is displayed on pages 352 - 355 in the print version)

```

1 // Fig. 7.17: GradeBook.cpp
2 // Member-function definitions for class GradeBook that
3 // uses an array to store test grades.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12 using std::setw;
13
14 #include "GradeBook.h" // GradeBook class definition
15
16 // constructor initializes courseName and grades array
17 GradeBook::GradeBook(string name, const int gradesArray[])
18 {
19 setCourseName(name); // initialize courseName
20
21 // copy grades from gradeArray to grades data member
22 for (int grade = 0; grade < students; grade++)
23 grades[grade] = gradesArray[grade];
24 } // end GradeBook constructor
25
26 // function to set the course name
27 void GradeBook::setCourseName(string name)
28 {
29 courseName = name; // store the course name
30 } // end function setCourseName
31
32 // function to retrieve the course name
33 string GradeBook::getCourseName()
34 {
35 return courseName;
36 } // end function getCourseName
37
38 // display a welcome message to the GradeBook user
39 void GradeBook::displayMessage()
40 {
41 // this statement calls getCourseName to get the
42 // name of the course this GradeBook represents
43 cout << "Welcome to the grade book for\n" << getCourseName() << "!"
44 << endl;
45 } // end function displayMessage
46
47 // perform various operations on the data
48 void GradeBook::processGrades()
49 {

```

```

50 // output grades array
51 outputGrades();
52
53 // call function getAverage to calculate the average grade
54 cout << "\nClass average is " << setprecision(2) << fixed <<
55 getAverage() << endl;
56
57 // call functions getMinimum and getMaximum
58 cout << "Lowest grade is " << getMinimum() << "\nHighest grade is "
59 << getMaximum() << endl;
60
61 // call function outputBarChart to print grade distribution chart
62 outputBarChart();
63 } // end function processGrades
64
65 // find minimum grade
66 int GradeBook::getMinimum()
67 {
68 int lowGrade = 100; // assume lowest grade is 100
69
70 // loop through grades array
71 for (int grade = 0; grade < students; grade++)
72 {
73 // if current grade lower than lowGrade, assign it to lowGrade
74 if (grades[grade] < lowGrade)
75 lowGrade = grades[grade]; // new lowest grade
76 } // end for
77
78 return lowGrade; // return lowest grade
79 } // end function getMinimum
80
81 // find maximum grade
82 int GradeBook::getMaximum()
83 {
84 int highGrade = 0; // assume highest grade is 0
85
86 // loop through grades array
87 for (int grade = 0; grade < students; grade++)
88 {
89 // if current grade higher than highGrade, assign it to highGrade
90 if (grades[grade] > highGrade)
91 highGrade = grades[grade]; // new highest grade
92 } // end for
93
94 return highGrade; // return highest grade
95 } // end function getMaximum
96
97 // determine average grade for test
98 double GradeBook::getAverage()

```

```

99 {
100 int total = 0; // initialize total
101
102 // sum grades in array
103 for (int grade = 0; grade < students; grade++)
104 total += grades[grade];
105
106 // return average of grades
107 return static_cast< double >(total) / students;
108 } // end function getAverage
109
110 // output bar chart displaying grade distribution
111 void GradeBook::outputBarChart()
112 {
113 cout << "\nGrade distribution:" << endl;
114
115 // stores frequency of grades in each range of 10 grades
116 const int frequencySize = 11;
117 int frequency[frequencySize] = { 0 };
118
119 // for each grade, increment the appropriate frequency
120 for (int grade = 0; grade < students; grade++)
121 frequency[grades[grade] / 10]++;
122
123 // for each grade frequency, print bar in chart
124 for (int count = 0; count < frequencySize; count++)
125 {
126 // output bar labels ("0-9:", ..., "90-99:", "100:")
127 if (count == 0)
128 cout << " 0-9: ";
129 else if (count == 10)
130 cout << " 100: ";
131 else
132 cout << count * 10 << "-" << (count * 10) + 9 << ": ";
133
134 // print bar of asterisks
135 for (int stars = 0; stars < frequency[count]; stars++)
136 cout << '*';
137
138 cout << endl; // start a new line of output
139 } // end outer for
140 } // end function outputBarChart
141
142 // output the contents of the grades array
143 void GradeBook::outputGrades()
144 {
145 cout << "\nThe grades are:\n\n";
146
147 // output each student's grade

```

```

148 for (int student = 0; student < students; student++)
149 cout << "Student " << setw(2) << student + 1 << ":" << setw(3)
150 << grades[student] << endl;
151 } // end function outputGrades

```

[Page 356]

**Figure 7.18. Creates a GradeBook object using an array of grades, then invokes member function processGrades to analyze them.**

```

1 // Fig. 7.18: fig07_18.cpp
2 // Creates GradeBook object using an array of grades.
3
4 #include "GradeBook.h" // GradeBook class definition
5
6 // function main begins program execution
7 int main()
8 {
9 // array of student grades
10 int gradesArray[GradeBook::students] =
11 { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
12
13 GradeBook myGradeBook(
14 "CS101 Introduction to C++ Programming", gradesArray);
15 myGradeBook.displayMessage();
16 myGradeBook.processGrades();
17 return 0;
18 } // end main

```

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

The grades are:

Student 1: 87  
Student 2: 68  
Student 3: 94  
Student 4: 100  
Student 5: 83  
Student 6: 78  
Student 7: 85  
Student 8: 91  
Student 9: 76

```
Student 10: 87
```

```
Class average is 84.90
Lowest grade is 68
Highest grade is 100
```

```
Grade distribution:
```

```
0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
100: *
```

[Page 357]

Member function `processGrades` (declared in line 21 of Fig. 7.16 and defined in lines 4863 of Fig. 7.17) contains a series of member function calls that output a report summarizing the grades. Line 51 calls member function `outputGrades` to print the contents of the array `grades`. Lines 148150 in member function `outputGrades` use a `for` statement to output each student's grade. Although array indices start at 0, a professor would typically number students starting at 1. Thus, lines 149150 output `student + 1` as the student number to produce grade labels "Student 1:", "Student 2:", and so on.

Member function `processGrades` next calls member function `getAverage` (lines 5455) to obtain the average of the grades in the array. Member function `getAverage` (declared in line 24 of Fig. 7.16 and defined in lines 98108) uses a `for` statement to total the values in array `grades` before calculating the average. Note that the averaging calculation in line 107 uses `const static` data member `students` to determine the number of grades being averaged.

Lines 5859 in member function `processGrades` call member functions `getMinimum` and `getMaximum` to determine the lowest and highest grades of any student on the exam, respectively. Let us examine how member function `getMinimum` finds the lowest grade. Because the highest grade allowed is 100, we begin by assuming that 100 is the lowest grade (line 68). Then, we compare each of the elements in the array to the lowest grade, looking for smaller values. Lines 7176 in member function `getMinimum` loop through the array, and lines 7475 compare each grade to `lowGrade`. If a grade is less than `lowGrade`, `lowGrade` is set to that grade. When line 78 executes, `lowGrade` contains the lowest grade in the array. Member

function `getMaximum` (lines 8295) works similarly to member function `getMinimum`.

Finally, line 62 in member function `processGrades` calls member function `outputBarChart` to print a distribution chart of the grade data using a technique similar to that in Fig. 7.9. In that example, we manually calculated the number of grades in each category (i.e., 09, 1019, ..., 9099 and 100) by simply looking at a set of grades. In this example, lines 120121 use a technique similar to that in Fig. 7.10 and Fig. 7.11 to calculate the frequency of grades in each category. Line 117 declares and creates array `frequency` of 11 `ints` to store the frequency of grades in each grade category. For each grade in array `grades`, lines 120121 increment the appropriate element of the `frequency` array. To determine which element to increment, line 121 divides the current grade by 10 using integer division. For example, if grade is 85, line 121 increments `frequency[ 8 ]` to update the count of grades in the range 8089. Lines 124139 next print the bar chart (see Fig. 7.18) based on the values in array `frequency`. Like lines 2930 of Fig. 7.9, lines 135136 of Fig. 7.17 use a value in array `frequency` to determine the number of asterisks to display in each bar.

## Testing Class GradeBook

The program of Fig. 7.18 creates an object of class `GradeBook` (Figs. 7.167.17) using the `int` array `gradesArray` (declared and initialized in lines 1011). Note that we use the binary scope resolution operator (`::`) in the expression "GradeBook::students" (line 10) to access class `GradeBook`'s static constant `students`. We use this constant here to create an array that is the same size as array `grades` stored as a data member in class `GradeBook`. Lines 1314 pass a course name and `gradesArray` to the `GradeBook` constructor. Line 15 displays a welcome message, and line 16 invokes the `GradeBook` object's `processGrades` member function. The output reveals the summary of the 10 grades in `myGradeBook`.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 358]

## 7.7. Searching Arrays with Linear Search

Often a programmer will be working with large amounts of data stored in arrays. It may be necessary to determine whether an array contains a value that matches a certain **key value**. The process of finding a particular element of an array is called **searching**. In this section we discuss the simple linear search. Exercise 7.33 at the end of this chapter asks you to implement a recursive version of the linear search. In Chapter 20, Searching and Sorting, we present the more complex, yet more efficient, binary search.

### Linear Search

The **linear search** (Fig. 7.19, lines 3744) compares each element of an array with a **search key** (line 40). Because the array is not in any particular order, it is just as likely that the value will be found in the first element as the last. On average, therefore, the program must compare the search key with half the elements of the array. To determine that a value is not in the array, the program must compare the search key to every element in the array.

**Figure 7.19. Linear search of an array.**

(This item is displayed on pages 358 - 359 in the print version)

```

1 // Fig. 7.19: fig07_19.cpp
2 // Linear search of an array.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int linearSearch(const int [], int, int); // prototype
9
10 int main()
11 {
12 const int arraySize = 100; // size of array a
13 int a[arraySize]; // create array a
14 int searchKey; // value to locate in array a
15
16 for (int i = 0; i < arraySize; i++)

```

```

17 a[i] = 2 * i; // create some data
18
19 cout << "Enter integer search key: ";
20 cin >> searchKey;
21
22 // attempt to locate searchKey in array a
23 int element = linearSearch(a, searchKey, arraySize);
24
25 // display results
26 if (element != -1)
27 cout << "Found value in element " << element << endl;
28 else
29 cout << "Value not found" << endl;
30
31 return 0; // indicates successful termination
32 } // end main
33
34 // compare key to every element of array until location is
35 // found or until end of array is reached; return subscript of
36 // element if key or -1 if key not found
37 int linearSearch(const int array[], int key, int sizeOfArray)
38 {
39 for (int j = 0; j < sizeOfArray; j++)
40 if (array[j] == key) // if found,
41 return j; // return location of key
42
43 return -1; // key not found
44 } // end function linearSearch

```

Enter integer search key: 36  
 Found value in element 18

Enter integer search key: 37  
 Value not found

The linear searching method works well for small arrays or for unsorted arrays (i.e., arrays whose elements are in no particular order). However, for large arrays, linear searching is inefficient. If the array

is sorted (e.g., its elements are in ascending order), you can use the high-speed binary search technique that you will learn about in [Chapter 20](#), Searching and Sorting.

 PREV

NEXT 

**page footer**

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 360]

**Figure 7.20. Sorting an array with insertion sort.**

(This item is displayed on pages 360 - 361 in the print version)

```
1 // Fig. 7.20: fig07_20.cpp
2 // This program sorts an array's values into ascending order.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 const int arraySize = 10; // size of array a
13 int data[arraySize] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
14 int insert; // temporary variable to hold element to insert
15
16 cout << "Unsorted array:\n";
17
18 // output original array
19 for (int i = 0; i < arraySize; i++)
20 cout << setw(4) << data[i];
21
22 // insertion sort
23 // loop over the elements of the array
24 for (int next = 1; next < arraySize; next++)
25 {
26 insert = data[next]; // store the value in the current element
27
28 int moveItem = next; // initialize location to place element
29
30 // search for the location in which to put the current element
31 while ((moveItem > 0) && (data[moveItem - 1] > insert))
32 {
33 // shift element one slot to the right
34 data[moveItem] = data[moveItem - 1];
35 moveItem--;
36 } // end while
37 }
```

```

38 data[moveItem] = insert; // place inserted element into the array
39 } // end for
40
41 cout << "\nSorted array:\n";
42
43 // output sorted array
44 for (int i = 0; i < arraySize; i++)
45 cout << setw(4) << data[i];
46
47 cout << endl;
48 return 0; // indicates successful termination
49 } // end main

```

Unsorted array:  
 34 56 4 10 77 51 93 30 5 52  
 Sorted array:  
 4 5 10 30 34 51 52 56 77 93

Line 13 of Fig. 7.20 declares and initializes array `data` with the following values:

34 56 4 10 77 51 93 30 5 52

The program first looks at `data[ 0 ]` and `data[ 1 ]`, whose values are 34 and 56, respectively. These two elements are already in order, so the program continues if they were out of order, the program would swap them.

[Page 361]

In the second iteration, the program looks at the value of `data[ 2 ]`, 4. This value is less than 56, so the program stores 4 in a temporary variable and moves 56 one element to the right. The program then checks and determines that 4 is less than 34, so it moves 34 one element to the right. The program has now reached the beginning of the array, so it places 4 in `data[ 0 ]`. The array now is

4 34 56 10 77 51 93 30 5 52

In the third iteration, the program stores the value of `data[ 3 ]`, 10, in a temporary variable. Then the program compares 10 to 56 and moves 56 one element to the right because it is larger than 10. The

program then compares 10 to 34, moving 34 right one element. When the program compares 10 to 4, it observes that 10 is larger than 4 and places 10 in `data[ 1 ]`. The array now is

```
4 10 34 56 77 51 93 30 5 52
```

Using this algorithm, at the  $i^{\text{th}}$  iteration, the first  $i$  elements of the original array are sorted. They may not be in their final locations, however, because smaller values may be located later in the array.

The sorting is performed by the `for` statement in lines 2439 that loops over the elements of the array. In each iteration, line 26 temporarily stores in variable `insert` (declared in line 14) the value of the element that will be inserted into the sorted portion of the array. Line 28 declares and initializes the variable `moveItem`, which keeps track of where to insert the element. Lines 3136 loop to locate the correct position where the element should be inserted. The loop terminates either when the program reaches the front of the array or when it reaches an element that is less than the value to be inserted. Line 34 moves an element to the right, and line 35 decrements the position at which to insert the next element. After the while loop ends, line 38 inserts the element into place. When the `for` statement in lines 2439 terminates, the elements of the array are sorted.

The chief virtue of the insertion sort is that it is easy to program; however, it runs slowly. This becomes apparent when sorting large arrays. In the exercises, we will investigate some alternate algorithms for sorting an array. We investigate sorting and searching in greater depth in [Chapter 20](#).

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 363]

**Figure 7.22. Initializing multidimensional arrays.**

```
1 // Fig. 7.22: fig07_22.cpp
2 // Initializing multidimensional arrays.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void printArray(const int [][3]); // prototype
8
9 int main()
10 {
11 int array1[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
12 int array2[2][3] = { { 1, 2, 3, 4, 5 } };
13 int array3[2][3] = { { 1, 2 }, { 4 } };
14
15 cout << "Values in array1 by row are:" << endl;
16 printArray(array1);
17
18 cout << "\nValues in array2 by row are:" << endl;
19 printArray(array2);
20
21 cout << "\nValues in array3 by row are:" << endl;
22 printArray(array3);
23 return 0; // indicates successful termination
24 } // end main
25
26 // output array with two rows and three columns
27 void printArray(const int a[][3])
28 {
29 // loop through array's rows
30 for (int i = 0; i < 2; i++)
31 {
32 // loop through columns of current row
33 for (int j = 0; j < 3; j++)
34 cout << a[i][j] << ' ';
35
36 cout << endl; // start new line of output
37 } // end outer for
38 } // end function printArray
```

```
Values in array1 by row are:
1 2 3
4 5 6

Values in array2 by row are:
1 2 3
4 5 0

Values in array3 by row are:
1 2 0
4 0 0
```

The declaration of `array1` (line 11) provides six initializers in two sublists. The first sublist initializes row 0 of the array to the values 1, 2 and 3; and the second sublist initializes row 1 of the array to the values 4, 5 and 6. If the braces around each sublist are removed from the `array1` initializer list, the compiler initializes the elements of row 0 followed by the elements of row 1, yielding the same result.

---

[Page 364]

The declaration of `array2` (line 12) provides only five initializers. The initializers are assigned to row 0, then row 1. Any elements that do not have an explicit initializer are initialized to zero, so `array2[ 1 ][ 2 ]` is initialized to zero.

The declaration of `array3` (line 13) provides three initializers in two sublists. The sublist for row 0 explicitly initializes the first two elements of row 0 to 1 and 2; the third element is implicitly initialized to zero. The sublist for row 1 explicitly initializes the first element to 4 and implicitly initializes the last two elements to zero.

The program calls function `printArray` to output each array's elements. Notice that the function definition (lines 2738) specifies the parameter `const int a[][][ 3 ]`. When a function receives a one-dimensional array as an argument, the array brackets are empty in the function's parameter list. The size of the first dimension (i.e., the number of rows) of a two-dimensional array is not required either, but all subsequent dimension sizes are required. The compiler uses these sizes to determine the locations in memory of elements in multidimensional arrays. All array elements are stored consecutively in memory, regardless of the number of dimensions. In a two-dimensional array, row 0 is stored in memory followed by row 1. In a two-dimensional array, each row is a one-dimensional array. To locate an element in a particular row, the function must know exactly how many elements are in each row so it can skip the proper number of memory locations when accessing the array. Thus, when accessing `a[ 1 ][ 2 ]`, the function knows to skip row 0's three elements in memory to get to row 1. Then, the function accesses

element 2 of that row.

Many common array manipulations use `for` repetition statements. For example, the following `for` statement sets all the elements in row 2 of array `a` in Fig. 7.21 to zero:

```
for (column = 0; column < 4; column++)
 a[2][column] = 0;
```

The `for` statement varies only the second subscript (i.e., the column subscript). The preceding `for` statement is equivalent to the following assignment statements:

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

The following nested `for` statement determines the total of all the elements in array `a`:

```
total = 0;

for (row = 0; row < 3; row++)

 for (column = 0; column < 4; column++)
 total += a[row][column];
```

The `for` statement totals the elements of the array one row at a time. The outer `for` statement begins by setting `row` (i.e., the row subscript) to 0, so the elements of row 0 may be totaled by the inner `for` statement. The outer `for` statement then increments `row` to 1, so the elements of row 1 can be totaled. Then, the outer `for` statement increments `row` to 2, so the elements of row 2 can be totaled. When the nested `for` statement terminates, `total` contains the sum of all the array elements.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 369]

Member functions `getMinimum`, `getMaximum`, `outputBarChart` and `outputGrades` each loop through array `grades` by using nested `for` statements. For example, consider the nested `for` statement in member function `getMinimum` (lines 7079). The outer `for` statement begins by setting `student` (i.e., the row subscript) to 0, so the elements of row 0 can be compared with variable `lowGrade` in the body of the inner `for` statement. The inner `for` statement loops through the grades of a particular row and compares each grade with `lowGrade`. If a grade is less than `lowGrade`, `lowGrade` is set to that grade. The outer `for` statement then increments the row subscript to 1. The elements of row 1 are compared with variable `lowGrade`. The outer `for` statement then increments the row subscript to 2, and the elements of row 2 are compared with variable `lowGrade`. This repeats until all rows of `grades` have been traversed. When execution of the nested `for` statement is complete, `lowGrade` contains the smallest grade in the two-dimensional array. Member function `getMaximum` works similarly to member function `getMinimum`.

[Page 370]

Member function `outputBarChart` in Fig. 7.24 is nearly identical to the one in Fig. 7.17. However, to output the overall grade distribution for a whole semester, the member function uses a nested `for` statement (lines 127130) to create the one-dimensional array `frequency` based on all the grades in the two-dimensional array. The rest of the code in each of the two `outputBarChart` member functions that displays the chart is identical.

**Figure 7.24. GradeBook class member-function definitions manipulating a two-dimensional array of grades.**

(This item is displayed on pages 366 - 369 in the print version)

```

1 // Fig. 7.24: GradeBook.cpp
2 // Member-function definitions for class GradeBook that
3 // uses a two-dimensional array to store grades.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip> // parameterized stream manipulators
11 using std::setprecision; // sets numeric output precision
12 using std::setw; // sets field width
13
14 // include definition of class GradeBook from GradeBook.h
15 #include "GradeBook.h"
16

```

```

17 // two-argument constructor initializes courseName and grades array
18 GradeBook::GradeBook(string name, const int gradesArray[][tests])
19 {
20 setCourseName(name); // initialize courseName
21
22 // copy grades from gradeArray to grades
23 for (int student = 0; student < students; student++)
24
25 for (int test = 0; test < tests; test++)
26 grades[student][test] = gradesArray[student][test];
27 } // end two-argument GradeBook constructor
28
29 // function to set the course name
30 void GradeBook::setCourseName(string name)
31 {
32 courseName = name; // store the course name
33 } // end function setCourseName
34
35 // function to retrieve the course name
36 string GradeBook::getCourseName()
37 {
38 return courseName;
39 } // end function getCourseName
40
41 // display a welcome message to the GradeBook user
42 void GradeBook::displayMessage()
43 {
44 // this statement calls getCourseName to get the
45 // name of the course this GradeBook represents
46 cout << "Welcome to the grade book for\n" << getCourseName() << "!"
47 << endl;
48 } // end function displayMessage
49
50 // perform various operations on the data
51 void GradeBook::processGrades()
52 {
53 // output grades array
54 outputGrades();
55
56 // call functions getMinimum and getMaximum
57 cout << "\nLowest grade in the grade book is " << getMinimum()
58 << "\nHighest grade in the grade book is " << getMaximum() << endl;
59
60 // output grade distribution chart of all grades on all tests
61 outputBarChart();
62 } // end function processGrades
63
64 // find minimum grade
65 int GradeBook::getMinimum()
66 {
67 int lowGrade = 100; // assume lowest grade is 100

```

```

68
69 // loop through rows of grades array
70 for (int student = 0; student < students; student++)
71 {
72 // loop through columns of current row
73 for (int test = 0; test < tests; test++)
74 {
75 // if current grade less than lowGrade, assign it to lowGrade
76 if (grades[student][test] < lowGrade)
77 lowGrade = grades[student][test]; // new lowest grade
78 } // end inner for
79 } // end outer for
80
81 return lowGrade; // return lowest grade
82 } // end function getMinimum
83
84 // find maximum grade
85 int GradeBook::getMaximum()
86 {
87 int highGrade = 0; // assume highest grade is 0
88
89 // loop through rows of grades array
90 for (int student = 0; student < students; student++)
91 {
92 // loop through columns of current row
93 for (int test = 0; test < tests; test++)
94 {
95 // if current grade greater than lowGrade, assign it to highGrade
96 if (grades[student][test] > highGrade)
97 highGrade = grades[student][test]; // new highest grade
98 } // end inner for
99 } // end outer for
100
101 return highGrade; // return highest grade
102 } // end function getMaximum
103
104 // determine average grade for particular set of grades
105 double GradeBook::getAverage(const int setOfGrades[], const int grades)
106 {
107 int total = 0; // initialize total
108
109 // sum grades in array
110 for (int grade = 0; grade < grades; grade++)
111 total += setOfGrades[grade];
112
113 // return average of grades
114 return static_cast< double >(total) / grades;
115 } // end function getAverage
116
117 // output bar chart displaying grade distribution
118 void GradeBook::outputBarChart()

```

```

119 {
120 cout << "\nOverall grade distribution:" << endl;
121
122 // stores frequency of grades in each range of 10 grades
123 const int frequencySize = 11;
124 int frequency[frequencySize] = { 0 };
125
126 // for each grade, increment the appropriate frequency
127 for (int student = 0; student < students; student++)
128 {
129 for (int test = 0; test < tests; test++)
130 ++frequency[grades[student][test] / 10];
131
132 // for each grade frequency, print bar in chart
133 for (int count = 0; count < frequencySize; count++)
134 {
135 // output bar label ("0-9:", ..., "90-99:", "100:")
136 if (count == 0)
137 cout << " 0-9: ";
138 else if (count == 10)
139 cout << " 100: ";
140 else
141 cout << count * 10 << "-" << (count * 10) + 9 << ": ";
142
143 // print bar of asterisks
144 for (int stars = 0; stars < frequency[count]; stars++)
145 cout << '*';
146
147 cout << endl; // start a new line of output
148 } // end outer for
149 } // end function outputBarChart
150
151 // output the contents of the grades array
152 void GradeBook::outputGrades()
153 {
154 cout << "\nThe grades are:\n\n";
155 cout << " " ; // align column heads
156
157 // create a column heading for each of the tests
158 for (int test = 0; test < tests; test++)
159 cout << "Test " << test + 1 << " ";
160
161 cout << "Average" << endl; // student average column heading
162
163 // create rows/columns of text representing array grades
164 for (int student = 0; student < students; student++)
165 {
166 cout << "Student " << setw(2) << student + 1;
167
168 // output student's grades
169 for (int test = 0; test < tests; test++)

```

```

170 cout << setw(8) << grades[student][test];
171
172 // call member function getAverage to calculate student's average;
173 // pass row of grades and the value of tests as the arguments
174 double average = getAverage(grades[student], tests);
175 cout << setw(9) << setprecision(2) << fixed << average << endl;
176 } // end outer for
177 } // end function outputGrades

```

Member function `outputGrades` (lines 152177) also uses nested `for` statements to output values of the array `grades`, in addition to each student's semester average. The output in Fig. 7.25 shows the result, which resembles the tabular format of a professor's physical grade book. Lines 158159 print the column headings for each test. We use a counter-controlled `for` statement so that we can identify each test with a number. Similarly, the `for` statement in lines 164176 first outputs a row label using a counter variable to identify each student (line 166). Although array indices start at 0, note that lines 159 and 166 output `test + 1` and `student + 1`, respectively, to produce test and student numbers starting at 1 (see Fig. 7.25). The inner `for` statement in lines 169170 uses the outer `for` statement's counter variable `student` to loop through a specific row of array `grades` and output each student's test grade. Finally, line 174 obtains each student's semester average by passing the current row of `grades` (i.e., `grades[ student ]`) to member function `getAverage`.

---

[Page 371]

**Figure 7.25.** Creates a `GradeBook` object using a two-dimensional array of `grades`, then invokes member function `processGrades` to analyze them.

(This item is displayed on pages 370 - 371 in the print version)

```

1 // Fig. 7.25: fig07_25.cpp
2 // Creates GradeBook object using a two-dimensional array of grades.
3
4 #include "GradeBook.h" // GradeBook class definition
5
6 // function main begins program execution
7 int main()
8 {
9 // two-dimensional array of student grades
10 int gradesArray[GradeBook::students][GradeBook::tests] =
11 { { 87, 96, 70 },
12 { 68, 87, 90 },
13 { 94, 100, 90 },
14 { 100, 81, 82 },
15 { 83, 65, 85 },
16 { 78, 87, 65 },
17 { 85, 75, 83 },
18 { 91, 94, 100 },

```

```

19 { 76, 72, 84 },
20 { 87, 93, 73 } };
21
22 GradeBook myGradeBook(
23 "CS101 Introduction to C++ Programming", gradesArray);
24 myGradeBook.displayMessage();
25 myGradeBook.processGrades();
26 return 0; // indicates successful termination
27 } // end main

```

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

The grades are:

|            | Test 1 | Test 2 | Test 3 | Average |
|------------|--------|--------|--------|---------|
| Student 1  | 87     | 96     | 70     | 84.33   |
| Student 2  | 68     | 87     | 90     | 81.67   |
| Student 3  | 94     | 100    | 90     | 94.67   |
| Student 4  | 100    | 81     | 82     | 87.67   |
| Student 5  | 83     | 65     | 85     | 77.67   |
| Student 6  | 78     | 87     | 65     | 76.67   |
| Student 7  | 85     | 75     | 83     | 81.00   |
| Student 8  | 91     | 94     | 100    | 95.00   |
| Student 9  | 76     | 72     | 84     | 77.33   |
| Student 10 | 87     | 93     | 73     | 84.33   |

Lowest grade in the grade book is 65  
Highest grade in the grade book is 100

Overall grade distribution:

|        |       |
|--------|-------|
| 0-9:   |       |
| 10-19: |       |
| 20-29: |       |
| 30-39: |       |
| 40-49: |       |
| 50-59: |       |
| 60-69: | ***   |
| 70-79: | ***** |
| 80-89: | ***** |
| 90-99: | ***** |
| 100:   | ***   |

Member function `getAverage` (lines 105115) takes two argumentsa one-dimensional array of test results for a

particular student and the number of test results in the array. When line 174 calls `getAverage`, the first argument is `grades[ student ]`, which specifies that a particular row of the two-dimensional array `grades` should be passed to `getAverage`. For example, based on the array created in Fig. 7.25, the argument `grades[ 1 ]` represents the three values (a one-dimensional array of grades) stored in row 1 of the two-dimensional array `grades`. A two-dimensional array can be considered an array whose elements are one-dimensional arrays. Member function `getAverage` calculates the sum of the array elements, divides the total by the number of test results and returns the floating-point result as a `double` value (line 114).

---

[Page 372]

## Testing Class GradeBook

The program in Fig. 7.25 creates an object of class `GradeBook` (Figs. 7.237.24) using the two-dimensional array of `ints` named `gradesArray` (declared and initialized in lines 1020). Note that line 10 accesses class `GradeBook`'s static constants `students` and `tests` to indicate the size of each dimension of array `gradesArray`. Lines 2223 pass a course name and `gradesArray` to the `GradeBook` constructor. Lines 2425 then invoke `myGradeBook`'s `displayMessage` and `processGrades` member functions to display a welcome message and obtain a report summarizing the students' grades for the semester, respectively.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

---

[Page 373]

[Page 376]

Line 23 uses `vector` member function `size` to obtain the size (i.e., the number of elements) of `integers1`. Line 25 passes `integers1` to function `outputVector` (lines 88102), which uses square brackets (`[]`) to obtain the value in each element of the `vector` as a value that can be used for output. Note the resemblance of this notation to the notation used to access the value of an array element. Lines 28 and 30 perform the same tasks for `integers2`.

Member function `size` of class template `vector` returns the number of elements in a `vector` as a value of type `size_t` (which represents the type `unsigned int` on many systems). As a result, line 90 declares the control variable `i` to be of type `size_t`, too. On some compilers, declaring `i` as an `int` causes the compiler to issue a warning message, since the loop-continuation condition (line 92) would compare a `signed` value (i.e., `int i`) and an `unsigned` value (i.e., a value of type `size_t` returned by function `size`).

Lines 3435 pass `integers1` and `integers2` to function `inputVector` (lines 105109) to read values for each `vector`'s elements from the user. Function `inputVector` uses square brackets (`[]`) to obtain `lvalues` that can be used to store the input values in each element of the `vector`.

Line 46 demonstrates that `vector` objects can be compared directly with the `!=operator`. If the contents of two `vectors` are not equal, the operator returns `true`; otherwise, the operator returns `false`.

C++ Standard Library class template `vector` allows programmers to create a new `vector` object that is initialized with the contents of an existing `vector`. Line 51 creates a `vector` object (`integers3`) and initializes it with a copy of `integers1`. This invokes `vector`'s so-called copy constructor to perform the copy operation. You will learn about copy constructors in detail in [Chapter 11](#). Lines 53 and 55 output the size and contents of `integers3` to demonstrate that it was initialized correctly.

Line 59 assigns `integers2` to `integers1`, demonstrating that the assignment (`=`) operator can be used with `vector` objects. Lines 62 and 64 output the contents of both objects to show that they now contain identical values. Line 69 then compares `integers1` to `integers2` with the equality (`==`) operator to determine whether the contents of the two objects are equal after the assignment in line 59 (which they are).

Lines 73 and 77 demonstrate that a program can use square brackets (`[]`) to obtain a `vector` element as an **unmodifiable lvalue** and as a **modifiable lvalue**, respectively. An unmodifiable `lvalue` is an expression that identifies an object in memory (such as an element in a `vector`), but cannot be used to

modify that object. A modifiable lvalue also identifies an object in memory, but can be used to modify the object. As is the case with C-style pointer-based arrays, C++ does not perform any bounds checking when `vector` elements are accessed with square brackets. Therefore, the programmer must ensure that operations using `[ ]` do not accidentally attempt to manipulate elements outside the bounds of the `vector`. Standard class template `vector` does, however, provide bounds checking in its member function `at`, which "throws an exception" (see [Chapter 16](#), Exception Handling) if its argument is an invalid subscript. By default, this causes a C++ program to terminate. If the subscript is valid, function `at` returns the element at the specified location as a modifiable lvalue or an unmodifiable lvalue, depending on the context (`non-const` or `const`) in which the call appears. Line 83 demonstrates a call to function `at` with an invalid subscript.

---

[Page 377]

In this section, we demonstrated the C++ Standard Library class template `vector`, a robust, reusable class that can replace C-style pointer-based arrays. In [Chapter 11](#), you will see that `vector` achieves many of its capabilities by "overloading" C++'s built-in operators, and you will learn how to customize operators for use with your own classes in similar ways. For example, we create an `Array` class that, like class template `vector`, improves upon basic array capabilities. Our `Array` class also provides additional features, such as the ability to input and output entire arrays with operators `>>` and `<<`, respectively.



page footer



The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 378]

**Figure 7.27** lists the collaborations that can be derived from the requirements document. For each sending object, we list the collaborations in the order in which they are discussed in the requirements document. We list each collaboration involving a unique sender, message and recipient only once, even though the collaboration may occur several times during an ATM session. For example, the first row in [Fig. 7.27](#) indicates that the ATM collaborates with the Screen whenever the ATM needs to display a message to the user.

**Figure 7.27. Collaborations in the ATM system.**

| An object of class... | sends the message...                                                            | to an object of class...                                                    |
|-----------------------|---------------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| ATM                   | displayMessage<br>getInput<br>authenticateUser<br>execute<br>execute<br>execute | Screen<br>Keypad<br>BankDatabase<br>BalanceInquiry<br>Withdrawal<br>Deposit |
| BalanceInquiry        | getAvailableBalance<br>getTotalBalance<br>displayMessage                        | BankDatabase<br>BankDatabase<br>Screen                                      |

|              |                                                                                                         |                                                                                    |
|--------------|---------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| Withdrawal   | displayMessage<br>getInput<br>getAvailableBalance<br>isSufficientCashAvailable<br>debit<br>dispenseCash | Screen<br>Keypad<br>BankDatabase<br>CashDispenser<br>BankDatabase<br>CashDispenser |
| Deposit      | displayMessage<br>getInput<br>isEnvelopeReceived<br>credit                                              | Screen<br>Keypad<br>DepositSlot<br>BankDatabase                                    |
| BankDatabase | validatePIN<br>getAvailableBalance<br>getTotalBalance<br>debit<br>credit                                | Account<br>Account<br>Account<br>Account<br>Account                                |

Let's consider the collaborations in Fig. 7.27. Before allowing a user to perform any transactions, the ATM must prompt the user to enter an account number, then to enter a PIN. It accomplishes each of these tasks by sending a `displayMessage` message to the Screen. Both of these actions refer to the same collaboration between the ATM and the Screen, which is already listed in Fig. 7.27. The ATM obtains input in response to a prompt by sending a `getInput` message to the Keypad. Next, the ATM must determine whether the user-specified account number and PIN match those of an account in the database. It does so by sending an `authenticateUser` message to the BankDatabase. Recall that the BankDatabase cannot authenticate a user directly; only the user's Account (i.e., the Account that contains the account number specified by the user) can access the user's PIN to authenticate the user. Figure 7.27 therefore lists a collaboration in which the BankDatabase sends a `validatePIN` message to an Account.

After the user is authenticated, the ATM displays the main menu by sending a series of `displayMessage` messages to the Screen and obtains input containing a menu selection by sending a `getInput` message to the Keypad. We have already accounted for these collaborations. After the user chooses a type of transaction to perform, the ATM executes the transaction by sending an `execute` message to an object of the appropriate transaction class (i.e., a `BalanceInquiry`, a `Withdrawal` or a `Deposit`). For example, if the user chooses to perform a balance inquiry, the ATM sends an `execute` message to a `BalanceInquiry`.

Further examination of the requirements document reveals the collaborations involved in executing each transaction type. A `BalanceInquiry` retrieves the amount of money available in the user's account by sending a `getAvailableBalance` message to the `BankDatabase`, which responds by sending a `getAvailableBalance` message to the user's `Account`. Similarly, the `BalanceInquiry` retrieves the amount of money on deposit by sending a `getTotalBalance` message to the `BankDatabase`, which sends the same message to the user's `Account`. To display both measures of the user's balance at the same time, the `BalanceInquiry` sends a `displayMessage` message to the Screen.

A `Withdrawal` sends a series of `displayMessage` messages to the Screen to display a menu of standard withdrawal amounts (i.e., \$20, \$40, \$60, \$100, \$200). The `Withdrawal` sends a `getInput` message to the Keypad to obtain the user's menu selection. Next, the `Withdrawal` determines whether the requested withdrawal amount is less than or equal to the user's account balance. The `Withdrawal` can obtain the amount of money available in the user's account by sending a `getAvailableBalance` message to the `BankDatabase`. The `Withdrawal` then tests whether the cash dispenser contains enough cash by sending an `isSufficientCashAvailable` message to the `CashDispenser`. A `Withdrawal` sends a `debit` message to the `BankDatabase` to decrease the user's account balance. The `BankDatabase` in turn sends the same message to the appropriate `Account`. Recall that debiting funds from an `Account` decreases both the `totalBalance` and the `availableBalance`. To dispense the requested amount of cash, the `Withdrawal` sends a `dispenseCash` message to the `CashDispenser`. Finally, the `Withdrawal` sends a `displayMessage` message to the Screen, instructing the user to take the cash.

A `Deposit` responds to an `execute` message first by sending a `displayMessage` message to the Screen to prompt the user for a deposit amount. The `Deposit` sends a `getInput` message to the Keypad to obtain the user's input. The `Deposit` then sends a `displayMessage` message to the Screen to tell the user to insert a deposit envelope. To determine whether the deposit slot received an incoming deposit envelope, the `Deposit` sends an `isEnvelopeReceived` message to the `DepositSlot`. The `Deposit` updates the user's account by sending a `credit` message to the `BankDatabase`, which subsequently sends a `credit` message to the user's `Account`. Recall that crediting funds to an `Account` increases the `totalBalance` but not the `availableBalance`.

## Interaction Diagrams

Now that we have identified a set of possible collaborations between the objects in our ATM system, let us graphically model these interactions using the UML. The UML provides several types of [interaction diagrams](#) that model the behavior of a system by modeling how objects interact with one another. The [communication diagram](#) emphasizes which objects participate in collaborations. [Note: Communication diagrams were called [collaboration diagrams](#) in earlier versions of the UML.] Like the communication diagram, the [sequence diagram](#) shows collaborations among objects, but it emphasizes when messages are sent between objects over time.

[Page 380]

## Communication Diagrams

Figure 7.28 shows a communication diagram that models the ATM executing a `BalanceInquiry`. Objects are modeled in the UML as rectangles containing names in the form `objectName : ClassName`. In this example, which involves only one object of each type, we disregard the object name and list only a colon followed by the class name. [Note: Specifying the name of each object in a communication diagram is recommended when modeling multiple objects of the same type.] Communicating objects are connected with solid lines, and messages are passed between objects along these lines in the direction shown by arrows. The name of the message, which appears next to the arrow, is the name of an operation (i.e., a member function) belonging to the receiving object think of the name as a service that the receiving object provides to sending objects (its "clients").

**Figure 7.28. Communication diagram of the ATM executing a balance inquiry.**

[View full size image]



The solid filled arrow in Fig. 7.28 represents a message or [synchronous call](#) in the UML and a function call in C++. This arrow indicates that the flow of control is from the sending object (the ATM) to the receiving object (a `BalanceInquiry`). Since this is a synchronous call, the sending object may not send another message, or do anything at all, until the receiving object processes the message and returns control to the sending object. The sender just waits. For example, in Fig. 7.28, the ATM calls member function `execute` of a `BalanceInquiry` and may not send another message until `execute` has finished and returns control to the ATM. [Note: If this were an [asynchronous call](#), represented by a stick arrowhead, the sending object would not have to wait for the receiving object to return control it would continue sending additional messages immediately following the asynchronous call. Asynchronous calls often can be implemented in C++ using platform-specific libraries provided with your compiler. Such techniques are beyond the scope of this book.]

## Sequence of Messages in a Communication Diagram

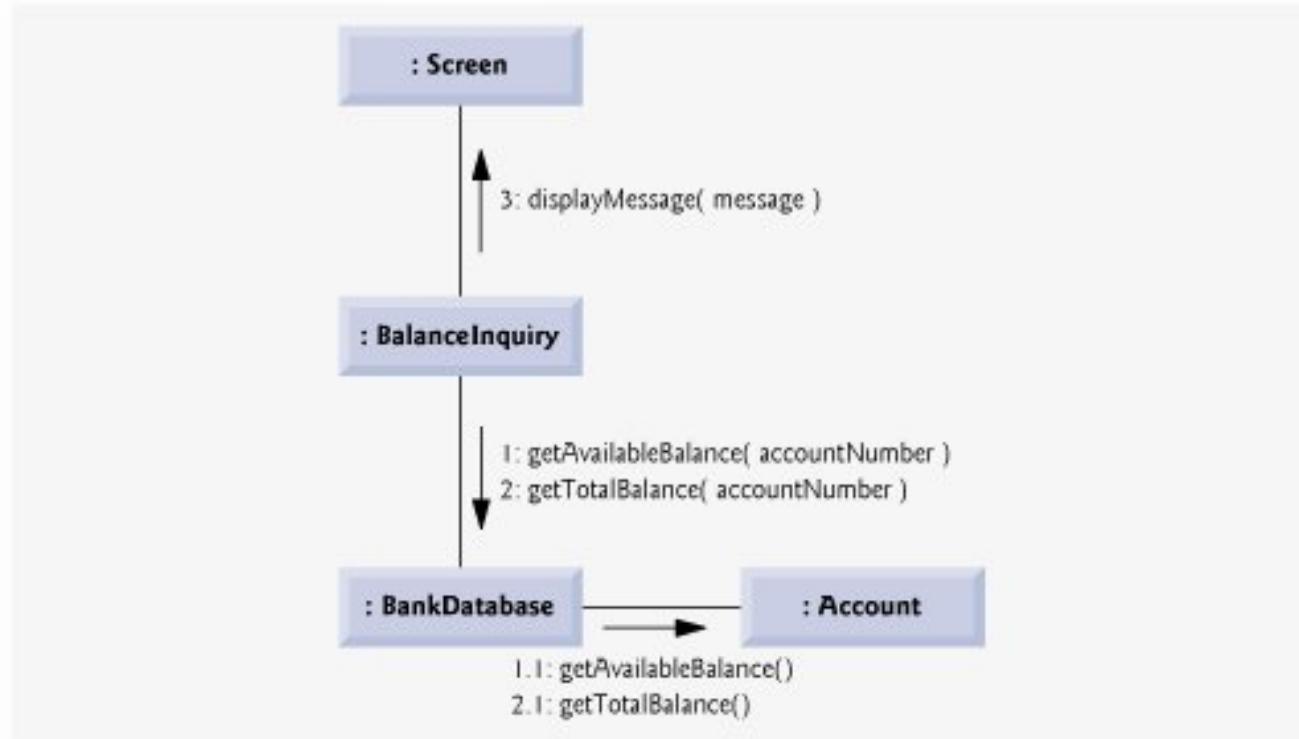
Figure 7.29 shows a communication diagram that models the interactions among objects in the system when an object of class BalanceInquiry executes. We assume that the object's accountNumber attribute contains the account number of the current user. The collaborations in Fig. 7.29 begin after the ATM sends an execute message to a BalanceInquiry (i.e., the interaction modeled in Fig. 7.28). The number to the left of a message name indicates the order in which the message is passed. The **sequence of messages** in a communication diagram progresses in numerical order from least to greatest. In this diagram, the numbering starts with message 1 and ends with message 3. The BalanceInquiry first sends a getAvailableBalance message to the BankDatabase (message 1), then sends a getTotalBalance message to the BankDatabase (message 2). Within the parentheses following a message name, we can specify a comma-separated list of the names of the parameters sent with the message (i.e., arguments in a C++ function call) the BalanceInquiry passes attribute accountNumber with its messages to the BankDatabase to indicate which Account's balance information to retrieve. Recall from Fig. 6.33 that operations getAvailableBalance and getTotalBalance of class BankDatabase each require a parameter to identify an account. The BalanceInquiry next displays the availableBalance and the totalBalance to the user by passing a displayMessage message to the Screen (message 3) that includes a parameter indicating the message to be displayed.

---

[Page 381]

**Figure 7.29. Communication diagram for executing a balance inquiry.**

[View full size image]



Note, however, that Fig. 7.29 models two additional messages passing from the BankDatabase to an Account (message 1.1 and message 2.1). To provide the ATM with the two balances of the user's

Account (as requested by messages 1 and 2), the BankDatabase must pass a getAvailableBalance and a getTotalBalance message to the user's Account. Such messages passed within the handling of another message are called [nested messages](#). The UML recommends using a decimal numbering scheme to indicate nested messages. For example, message 1.1 is the first message nested in message 1. The BankDatabase passes a getAvailableBalance message during BankDatabase's processing of a message by the same name. [Note: If the BankDatabase needed to pass a second nested message while processing message 1, the second message would be numbered 1.2.] A message may be passed only when all the nested messages from the previous message have been passed. For example, the BalanceInquiry passes message 3 only after messages 2 and 2.1 have been passed, in that order.

The nested numbering scheme used in communication diagrams helps clarify precisely when and in what context each message is passed. For example, if we numbered the messages in [Fig. 7.29](#) using a flat numbering scheme (i.e., 1, 2, 3, 4, 5), someone looking at the diagram might not be able to determine that BankDatabase passes the getAvailableBalance message (message 1.1) to an Account during the BankDatabase's processing of message 1, as opposed to after completing the processing of message 1. The nested decimal numbers make it clear that the second getAvailableBalance message (message 1.1) is passed to an Account within the handling of the first getAvailableBalance message (message 1) by the BankDatabase.

[Page 382]

## Sequence Diagrams

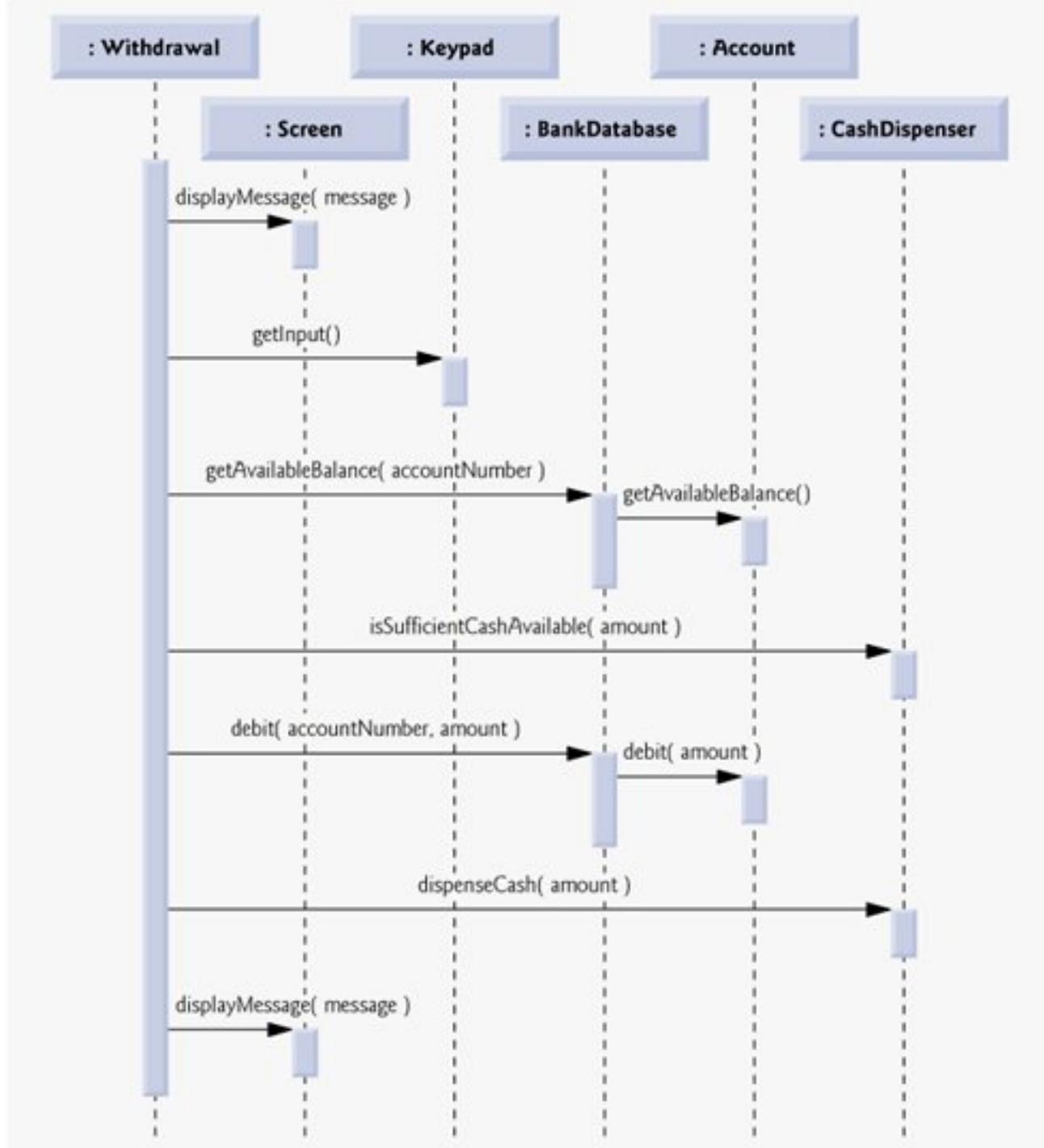
Communication diagrams emphasize the participants in collaborations but model their timing a bit awkwardly. A sequence diagram helps model the timing of collaborations more clearly. [Figure 7.30](#) shows a sequence diagram modeling the sequence of interactions that occur when a withdrawal executes. The dotted line extending down from an object's rectangle is that object's [lifeline](#), which represents the progression of time. Actions typically occur along an object's lifeline in chronological order from top to bottom: an action near the top typically happens before one near the bottom.

[Page 383]

**Figure 7.30. Sequence diagram that models a withdrawal executing.**

(This item is displayed on page 382 in the print version)

[\[View full size image\]](#)



Message passing in sequence diagrams is similar to message passing in communication diagrams. A solid arrow with a filled arrowhead extending from the sending object to the receiving object represents a message between two objects. The arrowhead points to an activation on the receiving object's lifeline. An **activation**, shown as a thin vertical rectangle, indicates that an object is executing. When an object returns control, a return message, represented as a dashed line with a stick arrowhead, extends from the activation of the object returning control to the activation of the object that initially sent the message. To eliminate clutter, we omit the return-message arrows—the UML allows this practice to make diagrams more readable. Like communication diagrams, sequence diagrams can indicate message parameters between the parentheses following a message name.

The sequence of messages in Fig. 7.30 begins when a Withdrawal prompts the user to choose a withdrawal amount by sending a displayMessage message to the Screen. The Withdrawal then sends a getInput message to the Keypad, which obtains input from the user. We have already modeled the control logic involved in a Withdrawal in the activity diagram of Fig. 5.28, so we do not show this logic in the sequence diagram of Fig. 7.30. Instead, we model the best-case scenario in which the balance of the user's account is greater than or equal to the chosen withdrawal amount, and the cash dispenser contains a sufficient amount of cash to satisfy the request. For information on how to model control logic in a sequence diagram, please refer to the Web resources and recommended readings listed at the end of Section 2.8.

After obtaining a withdrawal amount, the Withdrawal sends a getAvailableBalance message to the BankDatabase, which in turn sends a getAvailableBalance message to the user's Account. Assuming that the user's account has enough money available to permit the transaction, the Withdrawal next sends an isSufficientCashAvailable message to the CashDispenser. Assuming that there is enough cash available, the Withdrawal decreases the balance of the user's account (i.e., both the totalBalance and the availableBalance) by sending a debit message to the BankDatabase. The BankDatabase responds by sending a debit message to the user's Account. Finally, the Withdrawal sends a dispenseCash message to the CashDispenser and a displayMessage message to the Screen, telling the user to remove the cash from the machine.

We have identified the collaborations among objects in the ATM system and modeled some of these collaborations using UML interaction diagrams both communication diagrams and sequence diagrams. In the next "Software Engineering Case Study" section (Section 9.12), we enhance the structure of our model to complete a preliminary object-oriented design, then we begin implementing the ATM system.

## Software Engineering Case Study Self-Review Exercises

**7.1** A(n) \_\_\_\_\_ consists of an object of one class sending a message to an object of another class.

a.

association

b.

aggregation

c.

collaboration

d.

composition

- 7.2** Which form of interaction diagram emphasizes what collaborations occur? Which form emphasizes when collaborations occur?

---

[Page 384]

- 7.3** Create a sequence diagram that models the interactions among objects in the ATM system that occur when a Deposit executes successfully, and explain the sequence of messages modeled by the diagram.

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

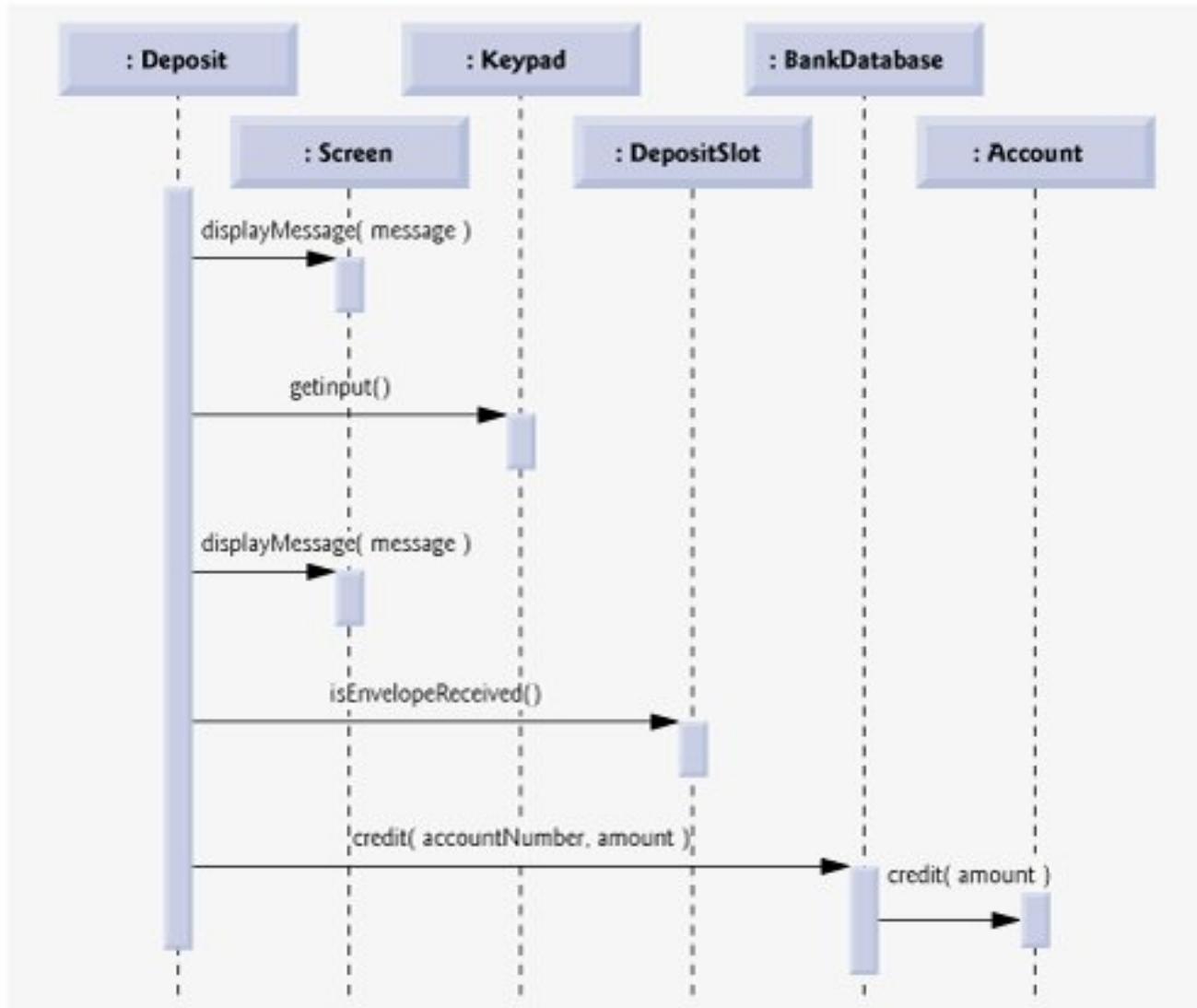
**7.1** c.

**7.2** Communication diagrams emphasize what collaborations occur. Sequence diagrams emphasize when collaborations occur.

**7.3** Figure 7.31 presents a sequence diagram that models the interactions between objects in the ATM system that occur when a Deposit executes successfully. Figure 7.31 indicates that a Deposit first sends a displayMessage message to the Screen to ask the user to enter a deposit amount. Next the Deposit sends a getInput message to the Keypad to receive input from the user. The Deposit then instructs the user to enter a deposit envelope by sending a displayMessage message to the Screen. The Deposit next sends an isEnvelopeReceived message to the DepositsSlot to confirm that the deposit envelope has been received by the ATM. Finally, the Deposit increases the totalBalance attribute (but not the availableBalance attribute) of the user's Account by sending a credit message to the BankDatabase. The BankDatabase responds by sending the same message to the user's Account.

**Figure 7.31. Sequence diagram that models a Deposit executing.**

[\[View full size image\]](#)



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 385]

## 7.13. Wrap-Up

This chapter began our introduction to data structures, exploring the use of arrays and `vectors` to store data in and retrieve data from lists and tables of values. The chapter examples demonstrated how to declare an array, initialize an array and refer to individual elements of an array. We also illustrated how to pass arrays to functions and how to use the `const` qualifier to enforce the principle of least privilege. Chapter examples also presented basic searching and sorting techniques. You learned how to declare and manipulate multidimensional arrays. Finally, we demonstrated the capabilities of C++ Standard Library class template `vector`, which provides a more robust alternative to arrays.

We continue our coverage of data structures in [Chapter 14](#), Templates, where we build a stack class template and in [Chapter 21](#), Data Structures, which introduces dynamic data structures, such as lists, queues, stacks and trees, that can grow and shrink as programs execute. [Chapter 23](#), Standard Template Library (STL), introduces several of the C++ Standard Library's predefined data structures, which programmers can use instead of building their own. [Chapter 23](#) presents the full functionality of class template `vector` and discusses many additional data structure classes, including `list` and `deque`, which are array-like data structures that can grow and shrink in response to a program's changing storage requirements.

We have now introduced the basic concepts of classes, objects, control statements, functions and arrays. In [Chapter 8](#), we present one of C++'s most powerful features—the pointer. Pointers keep track of where data and functions are stored in memory, which allows us to manipulate those items in interesting ways. After introducing basic pointer concepts, we examine in detail the close relationship among arrays, pointers and strings.

[◀ PREV](#)[NEXT ▶](#)**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 386]

- Arrays occupy space in memory. The programmer specifies the type of each element and the number of elements required by an array as follows:

```
type arrayName[arraySize] ;
```

and the compiler reserves the appropriate amount of memory.

- Arrays can be declared to contain any data type. For example, an array of type `char` can be used to store a character string.
- The elements of an array can be initialized in the array declaration by following the array name with an equals sign and an initializer lista comma-separated list (enclosed in braces) of constant initializers. When initializing an array with an initializer list, if there are fewer initializers than elements in the array, the remaining elements are initialized to zero.
- If the array size is omitted from a declaration with an initializer list, the compiler determines the number of elements in the array by counting the number of elements in the initializer list.
- If the array size and an initializer list are specified in an array declaration, the number of initializers must be less than or equal to the array size. Providing more initializers in an array initializer list than there are elements in the array is a compilation error.
- Constants must be initialized with a constant expression when they are declared and cannot be modified thereafter. Constants can be placed anywhere a constant expression is expected.
- C++ has no array bounds checking to prevent the computer from referring to an element that does not exist. Thus, an executing program can "walk off" either end of an array without warning. Programmers should ensure that all array references remain within the bounds of the array.
- A character array can be initialized using a string literal. The size of a character array is determined by the compiler based on the length of the string plus a special string-termination character called the null character (represented by the character constant '`\0`').
- All strings represented by character arrays end with the null character. A character array representing a string should always be declared large enough to hold the number of characters in the string and the terminating null character.
- Character arrays also can be initialized with individual character constants in an initializer list.
- Individual characters in a string can be accessed directly with array subscript notation.
- A string can be input directly into a character array from the keyboard using `cin` and `>>`.
- A character array representing a null-terminated string can be output with `cout` and `<<`.
- A static local variable in a function definition exists for the duration of the program but is visible only in the function body.
- A program initializes static local arrays when their declarations are first encountered. If a static array is not initialized explicitly by the programmer, each element of that array is initialized to zero by the compiler when the array is created.
- To pass an array argument to a function, specify the name of the array without any brackets. To pass an element of an array to a function, use the subscripted name of the array element as an argument in the function call.
- Arrays are passed to functions by referencethe called functions can modify the element values in the callers' original arrays. The value of the name of the array is the address in the computer's memory of the

first element of the array. Because the starting address of the array is passed, the called function knows precisely where the array is stored in memory.

- Individual array elements are passed by value exactly as simple variables are. Such simple single pieces of data are called scalars or scalar quantities.

[Page 387]

- To receive an array argument, a function's parameter list must specify that the function expects to receive an array. The size of the array is not required between the array brackets.
- C++ provides the type qualifier `const` that can be used to prevent modification of array values in the caller by code in a called function. When an array parameter is preceded by the `const` qualifier, the elements of the array become constant in the function body, and any attempt to modify an element of the array in the function body results in a compilation error.
- The linear search compares each element of an array with a search key. Because the array is not in any particular order, it is just as likely that the value will be found in the first element as the last. On average, therefore, a program must compare the search key with half the elements of the array. To determine that a value is not in the array, the program must compare the search key to every element in the array. The linear searching method works well for small arrays and is acceptable for unsorted arrays.
- An array can be sorted using insertion sort. The first iteration of this algorithm takes the second element and, if it is less than the first element, swaps it with the first element (i.e., the program inserts the second element in front of the first element). The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order. At the  $i^{\text{th}}$  iteration of this algorithm, the first  $i$  elements in the original array will be sorted. For small arrays, the insertion sort is acceptable, but for larger arrays it is inefficient compared to other more sophisticated sorting algorithms.
- Multidimensional arrays with two dimensions are often used to represent tables of values consisting of information arranged in rows and columns.
- Arrays that require two subscripts to identify a particular element are called two-dimensional arrays. An array with  $m$  rows and  $n$  columns is called an  $m$ -by- $n$  array.
- C++ Standard Library class template `vector` represents a more robust alternative to arrays featuring many capabilities that are not provided for C-style pointer-based arrays.
- By default, all the elements of an integer `vector` object are set to 0.
- A `vector` can be defined to store any data type using a declaration of the form

```
vector< type > name(size);
```

- Member function `size` of class template `vector` returns the number of elements in the `vector` on which it is invoked.
- The value of an element of a `vector` can be accessed or modified using square brackets (`[]`).
- Objects of standard class template `vector` can be compared directly with the equality (`==`) and inequality (`!=`) operators. The assignment (`=`) operator can also be used with `vector` objects.
- An unmodifiable lvalue is an expression that identifies an object in memory (such as an element in a `vector`), but cannot be used to modify that object. A modifiable lvalue also identifies an object in memory, but can be used to modify the object.
- Standard class template `vector` provides bounds checking in its member function `at`, which "throws an exception" if its argument is an invalid subscript. By default, this causes a C++ program to terminate.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 388]

constant variable

data structure

declare an array

element of an array

index

index zero

initialize an array

initializer

initializer list

insertion sort

key value

linear search of an array

magic number

m-by-n array

modifiable lvalue

multidimensional array

name of an array

named constant

null character ('\0')

off-by-one error

one-dimensional array

pass-by-reference

passing arrays to functions

position number

read-only variables

row of a two-dimensional array

row subscript

scalability

scalar

scalar quantity

search an array

search key

size member function of `vector`

sort an array

square brackets [ ]

`static` data member

string represented by a character array

subscript

table of values

tabular format

two-dimensional array

unmodifiable lvalue

value of an element

vector (C++ Standard Library class template)

"walk off" an array

zeroth element

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 389]

• Refer to array element 4.

• Assign the value 1.667 to array element 9.

• Assign the value 3.333 to the seventh element of the array.

• Print array elements 6 and 9 with two digits of precision to the right of the decimal point, and show the output that is actually displayed on the screen.

• Print all the array elements using a `for` statement. Define the integer variable `i` as a control variable for the loop. Show the output.

## 7.4

Answer the following questions regarding an array called `table`:

a.

Declare the array to be an integer array and to have 3 rows and 3 columns. Assume that the constant variable `arraySize` has been defined to be 3.

b.

How many elements does the array contain?

c.

Use a `for` repetition statement to initialize each element of the array to the sum of its subscripts.

Assume that the integer variables *i* and *j* are declared as control variables.

**d.**

Write a program segment to print the values of each element of array *table* in tabular format with 3 rows and 3 columns. Assume that the array was initialized with the declaration

```
int table[arraySize][arraySize] = { { 1, 8 }, { 2, 4, 6 }, { 5 } };
```

and the integer variables *i* and *j* are declared as control variables. Show the output.

## 7.5

Find the error in each of the following program segments and correct the error:

**a.**

```
#include <iostream>;
```

**b.**

```
arraySize = 10; // arraySize was declared const
```

**c.**

Assume that `int b[ 10 ] = { 0 };`

```
for (int i = 0; <= 10; i++)
 b[i] = 1;
```

**d.**

Assume that `int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`

```
a[1, 1] = 5;
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 390]

Output:

```
fractions[0] = 0.0
fractions[1] = 0.0
fractions[2] = 0.0
fractions[3] = 0.0
fractions[4] = 0.0
fractions[5] = 0.0
fractions[6] = 3.333
fractions[7] = 0.0
fractions[8] = 0.0
fractions[9] = 1.667
```

## 7.4

a.

```
int table[arraySize][arraySize];
```

b.

Nine.

c.

```
for (i = 0; i < arraySize; i++)

 for (j = 0; j < arraySize; j++)
 table[i][j] = i + j;
```

d.

```
cout << " [0] [1] [2]" << endl;

for (int i = 0; i < arraySize; i++) {
 cout << '[' << i << "] ";

 for (int j = 0; j < arraySize; j++)
 cout << setw(3) << table[i][j] << " ";
```

```
cout << endl;
```

Output:

|       |       |       |       |
|-------|-------|-------|-------|
|       | [ 0 ] | [ 1 ] | [ 2 ] |
| [ 0 ] | 1     | 8     | 0     |
| [ 1 ] | 2     | 4     | 6     |
| [ 2 ] | 5     | 0     | 0     |

## 7.5

a.

Error: Semicolon at end of #include preprocessor directive.

Correction: Eliminate semicolon.

b.

Error: Assigning a value to a constant variable using an assignment statement.

Correction: Initialize the constant variable in a const int arraySize declaration.

c.

Error: Referencing an array element outside the bounds of the array (b[10]).

Correction: Change the final value of the control variable to 9.

d.

Error: Array subscripting done incorrectly.

Correction: Change the statement to a[ 1 ][ 1 ] = 5;

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 391]

•

An m-by-n array contains \_\_\_\_\_ rows, \_\_\_\_\_ columns and \_\_\_\_\_ elements.

•

The name of the element in row 3 and column 5 of array d is \_\_\_\_\_.

## 7.7

Determine whether each of the following is true or false. If false, explain why.

a.

To refer to a particular location or element within an array, we specify the name of the array and the value of the particular element.

b.

An array declaration reserves space for the array.

c.

To indicate that 100 locations should be reserved for integer array p, the programmer writes the declaration

p[ 100 ];

d.

A for statement must be used to initialize the elements of a 15-element array to zero.

e.

Nested for statements must be used to total the elements of a two-dimensional array.

## 7.8

Write C++ statements to accomplish each of the following:

a.

Display the value of element 6 of character array f.

b.

Input a value into element 4 of one-dimensional floating-point array b.

c.

Initialize each of the 5 elements of one-dimensional integer array g to 8.

d.

Total and print the elements of floating-point array c of 100 elements.

e.

Copy array a into the first portion of array b. Assume double a[ 11 ], b[ 34 ];

f.

Determine and print the smallest and largest values contained in 99-element floating-point array w.

## 7.9

Consider a 2-by-3 integer array t.

a.

Write a declaration for t.

b.

How many rows does t have?

c.

How many columns does t have?

How many elements does t have?

e.

Write the names of all the elements in row 1 of t.

f.

Write the names of all the elements in column 2 of t.

g.

Write a single statement that sets the element of t in row 1 and column 2 to zero.

h.

Write a series of statements that initialize each element of t to zero. Do not use a loop.

i.

Write a nested for statement that initializes each element of t to zero.

j.

Write a statement that inputs the values for the elements of t from the terminal.

k.

Write a series of statements that determine and print the smallest value in array t.

l.

Write a statement that displays the elements in row 0 of t.

m.

Write a statement that totals the elements in column 3 of t.

n.

Write a series of statements that prints the array t in neat, tabular format. List the column subscripts as headings across the top and list the row subscripts at the left of each row.

**7.10**

Use a one-dimensional array to solve the following problem. A company pays its salespeople on a commission basis. The salespeople each receive \$200 per week plus 9 percent of their gross sales for that week. For example, a salesperson who grosses \$5000 in sales in a week receives \$200 plus 9 percent of \$5000, or a total of \$650. Write a program (using an array of counters) that determines how many of the salespeople earned salaries in each of the following ranges (assume that each salesperson's salary is truncated to an integer amount):

a.

\$200\$299

b.

\$300\$399

c.

\$400\$499

d.

\$500\$599

e.

\$600\$699

f.

\$700\$799

---

[Page 392]

g.

\$800\$899

h.

\$900\$999

\$1000 and over

## 7.11

(Bubble Sort) In the [bubble sort algorithm](#), smaller values gradually "bubble" their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom. The bubble sort makes several passes through the array. On each pass, successive pairs of elements are compared. If a pair is in increasing order (or the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array. Write a program that sorts an array of 10 integers using bubble sort.

## 7.12

The bubble sort described in [Exercise 7.11](#) is inefficient for large arrays. Make the following simple modifications to improve the performance of the bubble sort:

a.

After the first pass, the largest number is guaranteed to be in the highest-numbered element of the array; after the second pass, the two highest numbers are "in place," and so on. Instead of making nine comparisons on every pass, modify the bubble sort to make eight comparisons on the second pass, seven on the third pass, and so on.

b.

The data in the array may already be in the proper order or near-proper order, so why make nine passes if fewer will suffice? Modify the sort to check at the end of each pass if any swaps have been made. If none have been made, then the data must already be in the proper order, so the program should terminate. If swaps have been made, then at least one more pass is needed.

## 7.13

Write single statements that perform the following one-dimensional array operations:

a.

Initialize the 10 elements of integer array `counts` to zero.

b.

Add 1 to each of the 15 elements of integer array `bonus`.

c.

Read 12 values for double array `monthlyTemperatures` from the keyboard.

**d.**

Print the 5 values of integer array `bestScores` in column format.

## 7.14

Find the error(s) in each of the following statements:

**a.**

Assume that: `char str[ 5 ];`

```
cin >> str; // user types "hello"
```

**b.**

Assume that: `int a[ 3 ];`

```
cout << a[1] << " " << a[2] << " " << a[3] << endl;
```

**c.**

```
double f[3] = { 1.1, 10.01, 100.001, 1000.0001 };
```

**d.**

Assume that: `double d[ 2 ][ 10 ];`

```
d[1, 9] = 2.345;
```

## 7.15

Use a one-dimensional array to solve the following problem. Read in 20 numbers, each of which is between 10 and 100, inclusive. As each number is read, validate it and store it in the array only if it is not a duplicate of a number already read. After reading all the values, display only the unique values that the user entered. Provide for the "worst case" in which all 20 numbers are different. Use the smallest possible array to solve this problem.

## 7.16

Label the elements of a 3-by-5 one-dimensional array `sales` to indicate the order in which they are set to

zero by the following program segment:

```
for (row = 0; row < 3; row++)
 for (column = 0; column < 5; column++)
 sales[row][column] = 0;
```

## 7.17

Write a program that simulates the rolling of two dice. The program should use `rand` to roll the first die and should use `rand` again to roll the second die. The sum of the two values should then be calculated. [ Note: Each die can show an integer value from 1 to 6, so the sum of the two values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 being the least frequent sums.] [Figure 7.32](#) shows the 36 possible combinations of the two dice. Your program should roll the two dice 36,000 times. Use a one-dimensional array to tally the numbers of times each possible sum appears. Print the results in a tabular format. Also, determine if the totals are reasonable (i.e., there are six ways to roll a 7, so approximately one-sixth of all the rolls should be 7).

[Page 393]

**Figure 7.32. The 36 possible outcomes of rolling two dice.**

|   | 1 | 2 | 3 | 4  | 5  | 6  |
|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5  | 6  | 7  |
| 2 | 3 | 4 | 5 | 6  | 7  | 8  |
| 3 | 4 | 5 | 6 | 7  | 8  | 9  |
| 4 | 5 | 6 | 7 | 8  | 9  | 10 |
| 5 | 6 | 7 | 8 | 9  | 10 | 11 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |

## 7.18

What does the following program do?

```
1 // Ex. 7.18: Ex07_18.cpp
2 // What does this program do?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
```

```

6
7 int whatIsThis(int [], int); // function prototype
8
9 int main()
10 {
11 const int arraySize = 10;
12 int a[arraySize] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13
14 int result = whatIsThis(a, arraySize);
15
16 cout << "Result is " << result << endl;
17 return 0; // indicates successful termination
18 } // end main
19
20 // What does this function do?
21 int whatIsThis(int b[], int size)
22 {
23 if (size == 1) // base case
24 return b[0];
25 else // recursive step
26 return b[size - 1] + whatIsThis(b, size - 1);
27 } // end function whatIsThis

```

**7.19**

Modify the program of Fig. 6.11 to play 1000 games of craps. The program should keep track of the statistics and answer the following questions:

[Page 394]

**a.**

How many games are won on the 1<sup>st</sup> roll, 2<sup>nd</sup> roll, ..., 20<sup>th</sup> roll, and after the 20<sup>th</sup> roll?

**b.**

How many games are lost on the 1<sup>st</sup> roll, 2<sup>nd</sup> roll, ..., 20<sup>th</sup> roll, and after the 20<sup>th</sup> roll?

**c.**

What are the chances of winning at craps? [ Note: You should discover that craps is one of the fairest casino games. What do you suppose this means?]

**d.**

What is the average length of a game of craps?

e.

Do the chances of winning improve with the length of the game?

## 7.20

( Airline Reservations System) A small airline has just purchased a computer for its new automated reservations system. You have been asked to program the new system. You are to write a program to assign seats on each flight of the airline's only plane (capacity: 10 seats).

Your program should display the following menu of alternatives  
 Please type 1 for "First Class"  
 and Please type 2 for "Economy". If the person types 1, your program should assign a seat in the first class section (seats 1-5). If the person types 2, your program should assign a seat in the economy section (seats 6-10). Your program should print a boarding pass indicating the person's seat number and whether it is in the first class or economy section of the plane.

Use a one-dimensional array to represent the seating chart of the plane. Initialize all the elements of the array to 0 to indicate that all seats are empty. As each seat is assigned, set the corresponding elements of the array to 1 to indicate that the seat is no longer available.

Your program should, of course, never assign a seat that has already been assigned. When the first class section is full, your program should ask the person if it is acceptable to be placed in the economy section (and vice versa). If yes, then make the appropriate seat assignment. If no, then print the message "Next flight leaves in 3 hours".

## 7.21

What does the following program do?

```

1 // Ex. 7.21: Ex07_21.cpp
2 // What does this program do?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void someFunction(int [], int, int); // function prototype
8
9 int main()
10 {
11 const int arraySize = 10;
12 int a[arraySize] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13
14 cout << "The values in the array are:" << endl;

```

```
15 someFunction(a, 0, arraySize);
16 cout << endl;
17 return 0; // indicates successful termination
18 } // end main
19
20 // What does this function do?
21 void someFunction(int b[], int current, int size)
22 {
23 if (current < size)
24 {
25 someFunction(b, current + 1, size);
26 cout << b[current] << " ";
27 } // end if
28 } // end function someFunction
```

---

[Page 395]

## 7.22

Use a two-dimensional array to solve the following problem. A company has four salespeople (1 to 4) who sell five different products (1 to 5). Once a day, each salesperson passes in a slip for each different type of product sold. Each slip contains the following:

a.

The salesperson number

b.

The product number

c.

The total dollar value of that product sold that day

Thus, each salesperson passes in between 0 and 5 sales slips per day. Assume that the information from all of the slips for last month is available. Write a program that will read all this information for last month's sales and summarize the total sales by salesperson by product. All totals should be stored in the two-dimensional array sales. After processing all the information for last month, print the results in tabular format with each of the columns representing a particular salesperson and each of the rows representing a particular product. Cross total each row to get the total sales of each product for last month; cross total each column to get the total sales by salesperson for last month. Your tabular printout should include these cross totals to the right of the totaled rows and to the bottom of the totaled columns.

## 7.23

( Turtle Graphics ) The Logo language, which is popular among elementary school children, made the concept of turtle graphics famous. Imagine a mechanical turtle that walks around the room under the control of a C++ program. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves; while the pen is up, the turtle moves about freely without writing anything. In this problem, you will simulate the operation of the turtle and create a computerized sketchpad as well.

Use a 20-by-20 array `floor` that is initialized to zeros. Read commands from an array that contains them. Keep track of the current position of the turtle at all times and whether the pen is currently up or down. Assume that the turtle always starts at position (0, 0) of the floor with its pen up. The set of turtle commands your program must process are shown in Fig. 7.33.

**Figure 7.33. Turtle graphics commands.**

(This item is displayed on page 396 in the print version)

| Command | Meaning                                            |
|---------|----------------------------------------------------|
| 1       | Pen up                                             |
| 2       | Pen down                                           |
| 3       | Turn right                                         |
| 4       | Turn left                                          |
| 5,10    | Move forward 10 spaces (or a number other than 10) |
| 6       | Print the 20-by-20 array                           |
| 9       | End of data (sentinel)                             |

Suppose that the turtle is somewhere near the center of the floor. The following "program" would draw and print a 12-by-12 square and end with the pen in the up position:

```
2
5,12
3
5,12
3
5,12
3
```

```
5 ,12
1
6
9
```

As the turtle moves with the pen down, set the appropriate elements of array `floor` to 1's. When the `print` command (print) is given, wherever there is a 1 in the array, display an asterisk or some other character you choose. Wherever there is a zero, display a blank. Write a program to implement the turtle graphics capabilities discussed here. Write several turtle graphics programs to draw interesting shapes. Add other commands to increase the power of your turtle graphics language.

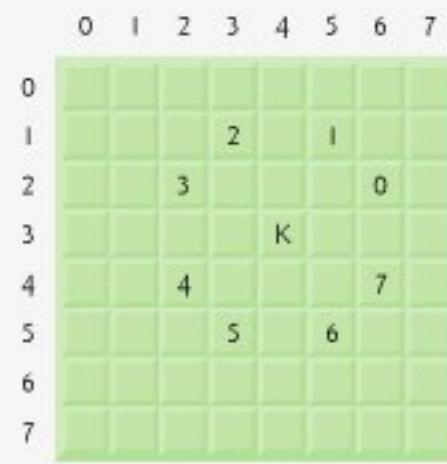
## 7.24

( Knight's Tour ) One of the more interesting puzzlers for chess buffs is the Knight's Tour problem. The question is this: Can the chess piece called the knight move around an empty chessboard and touch each of the 64 squares once and only once? We study this intriguing problem in depth in this exercise.

The knight makes L-shaped moves (over two in one direction and then over one in a perpendicular direction). Thus, from a square in the middle of an empty chessboard, the knight can make eight different moves (numbered 0 through 7) as shown in Fig. 7.34.

[Page 396]

**Figure 7.34. The eight possible moves of the knight.**



a.

Draw an 8-by-8 chessboard on a sheet of paper and attempt a Knight's Tour by hand. Put a 1 in

the first square you move to, a 2 in the second square, a 3 in the third, etc. Before starting the tour, estimate how far you think you will get, remembering that a full tour consists of 64 moves. How far did you get? Was this close to your estimate?

b.

Now let us develop a program that will move the knight around a chessboard. The board is represented by an 8-by-8 two-dimensional array `board`. Each of the squares is initialized to zero. We describe each of the eight possible moves in terms of both their horizontal and vertical components. For example, a move of type 0, as shown in Fig. 7.34, consists of moving two squares horizontally to the right and one square vertically upward. Move 2 consists of moving one square horizontally to the left and two squares vertically upward. Horizontal moves to the left and vertical moves upward are indicated with negative numbers. The eight moves may be described by two one-dimensional arrays, `horizontal` and `vertical`, as follows:

```
horizontal[0] = 2
horizontal[1] = 1
horizontal[2] = -1
horizontal[3] = -2
horizontal[4] = -2
```

---

[ Page 397 ]

```
horizontal[5] = -1
horizontal[6] = 1
horizontal[7] = 2
```

```
vertical[0] = -1
vertical[1] = -2
vertical[2] = -2
vertical[3] = -1
vertical[4] = 1
vertical[5] = 2
vertical[6] = 2
vertical[7] = 1
```

Let the variables `currentRow` and `currentColumn` indicate the row and column of the knight's current position. To make a move of type `moveNumber`, where `moveNumber` is between 0 and 7, your program uses the statements

```
currentRow += vertical[moveNumber];
currentColumn += horizontal[moveNumber];
```

Keep a counter that varies from 1 to 64. Record the latest count in each square the knight moves to. Remember to test each potential move to see if the knight has already visited that square, and, of course, test every potential move to make sure that the knight does not land off the chessboard. Now write a program to move the knight around the chessboard. Run the program.

How many moves did the knight make?

c.

After attempting to write and run a Knight's Tour program, you have probably developed some valuable insights. We will use these to develop a [heuristic](#) (or strategy) for moving the knight. Heuristics do not guarantee success, but a carefully developed heuristic greatly improves the chance of success. You may have observed that the outer squares are more troublesome than the squares nearer the center of the board. In fact, the most troublesome, or inaccessible, squares are the four corners.

Intuition may suggest that you should attempt to move the knight to the most troublesome squares first and leave open those that are easiest to get to, so when the board gets congested near the end of the tour, there will be a greater chance of success.

We may develop an "accessibility heuristic" by classifying each square according to how accessible it is and then always moving the knight to the square (within the knight's L-shaped moves, of course) that is most inaccessible. We label a two-dimensional array `accessibility` with numbers indicating from how many squares each particular square is accessible. On a blank chessboard, each center square is rated as 8, each corner square is rated as 2 and the other squares have accessibility numbers of 3, 4 or 6 as follows:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |

Now write a version of the Knight's Tour program using the accessibility heuristic. At any time, the knight should move to the square with the lowest accessibility number. In case of a tie, the knight may move to any of the tied squares. Therefore, the tour may begin in any of the four corners. [Note: As the knight moves around the chessboard, your program should reduce the accessibility numbers as more and more squares become occupied. In this way, at any given time during the tour, each available square's accessibility number will remain equal to precisely the number of squares from which that square may be reached.] Run this version of your program. Did you get a full tour? Now modify the program to run 64 tours, one starting from each square of the chessboard. How many full tours did you get?

[Page 398]

d.

Write a version of the Knight's Tour program which, when encountering a tie between two or more

squares, decides what square to choose by looking ahead to those squares reachable from the "tied" squares. Your program should move to the square for which the next move would arrive at a square with the lowest accessibility number.

## 7.25

(Knight's Tour: Brute Force Approaches) In [Exercise 7.24](#), we developed a solution to the Knight's Tour problem. The approach used, called the "accessibility heuristic," generates many solutions and executes efficiently.

As computers continue increasing in power, we will be able to solve more problems with sheer computer power and relatively unsophisticated algorithms. This is the "brute force" approach to problem solving.

a.

Use random number generation to enable the knight to walk around the chessboard (in its legitimate L-shaped moves, of course) at random. Your program should run one tour and print the final chessboard. How far did the knight get?

b.

Most likely, the preceding program produced a relatively short tour. Now modify your program to attempt 1000 tours. Use a one-dimensional array to keep track of the number of tours of each length. When your program finishes attempting the 1000 tours, it should print this information in neat tabular format. What was the best result?

c.

Most likely, the preceding program gave you some "respectable" tours, but no full tours. Now "pull all the stops out" and simply let your program run until it produces a full tour. [Caution: This version of the program could run for hours on a powerful computer.] Once again, keep a table of the number of tours of each length, and print this table when the first full tour is found. How many tours did your program attempt before producing a full tour? How much time did it take?

d.

Compare the brute force version of the Knight's Tour with the accessibility heuristic version. Which required a more careful study of the problem? Which algorithm was more difficult to develop? Which required more computer power? Could we be certain (in advance) of obtaining a full tour with the accessibility heuristic approach? Could we be certain (in advance) of obtaining a full tour with the brute force approach? Argue the pros and cons of brute force problem solving in general.

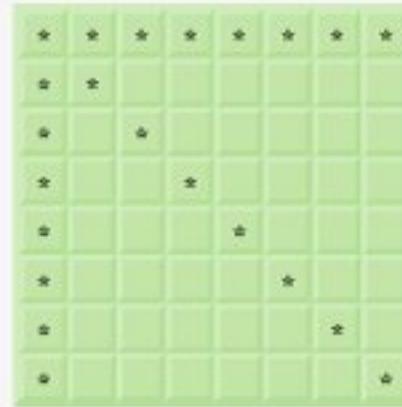
## 7.26

(Eight Queens) Another puzzler for chess buffs is the Eight Queens problem. Simply stated: Is it possible

to place eight queens on an empty chessboard so that no queen is "attacking" any other, i.e., no two queens are in the same row, the same column, or along the same diagonal? Use the thinking developed in [Exercise 7.24](#) to formulate a heuristic for solving the Eight Queens problem. Run your program. [Hint: It is possible to assign a value to each square of the chessboard indicating how many squares of an empty chessboard are "eliminated" if a queen is placed in that square. Each of the corners would be assigned the value 22, as in [Fig. 7.35](#).] Once these "elimination numbers" are placed in all 64 squares, an appropriate heuristic might be: Place the next queen in the square with the smallest elimination number. Why is this strategy intuitively appealing?

**Figure 7.35. The 22 squares eliminated by placing a queen in the upper-left corner.**

(This item is displayed on page 399 in the print version)



## 7.27

(Eight Queens: Brute Force Approaches) In this exercise, you will develop several brute-force approaches to solving the Eight Queens problem introduced in [Exercise 7.26](#).

a.

Solve the Eight Queens exercise, using the random brute force technique developed in [Exercise 7.25](#).

b.

Use an exhaustive technique, i.e., try all possible combinations of eight queens on the chessboard.

Why do you suppose the exhaustive brute force approach may not be appropriate for solving the Knight's Tour problem?

d.

Compare and contrast the random brute force and exhaustive brute force approaches in general.

## 7.28

(Knight's Tour: Closed-Tour Test) In the Knight's Tour, a full tour occurs when the knight makes 64 moves touching each square of the chess board once and only once. A closed tour occurs when the 64th move is one move away from the location in which the knight started the tour. Modify the Knight's Tour program you wrote in [Exercise 7.24](#) to test for a closed tour if a full tour has occurred.

## 7.29

(The Sieve of Eratosthenes) A prime integer is any integer that is evenly divisible only by itself and 1. The Sieve of Eratosthenes is a method of finding prime numbers. It operates as follows:

a.

Create an array with all elements initialized to 1 (true). Array elements with prime subscripts will remain 1. All other array elements will eventually be set to zero. You will ignore elements 0 and 1 in this exercise.

b.

Starting with array subscript 2, every time an array element is found whose value is 1, loop through the remainder of the array and set to zero every element whose subscript is a multiple of the subscript for the element with value 1. For array subscript 2, all elements beyond 2 in the array that are multiples of 2 will be set to zero (subscripts 4, 6, 8, 10, etc.); for array subscript 3, all elements beyond 3 in the array that are multiples of 3 will be set to zero (subscripts 6, 9, 12, 15, etc.); and so on.

When this process is complete, the array elements that are still set to one indicate that the subscript is a prime number. These subscripts can then be printed. Write a program that uses an array of 1000 elements to determine and print the prime numbers between 2 and 999. Ignore element 0 of the array.

## 7.30

(Bucket Sort) A [bucket sort](#) begins with a one-dimensional array of positive integers to be sorted and a two-dimensional array of integers with rows subscripted from 0 to 9 and columns subscripted from 0 to n1, where n is the number of values in the array to be sorted. Each row of the two-dimensional array is referred to as a bucket. Write a function `bucketSort` that takes an integer array and the array size as

arguments and performs as follows:

a.

Place each value of the one-dimensional array into a row of the bucket array based on the value's ones digit. For example, 97 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0. This is called a "distribution pass."

b.

Loop through the bucket array row by row, and copy the values back to the original array. This is called a "gathering pass." The new order of the preceding values in the one-dimensional array is 100, 3 and 97.

c.

Repeat this process for each subsequent digit position (tens, hundreds, thousands, etc.).

---

[Page 400]

On the second pass, 100 is placed in row 0, 3 is placed in row 0 (because 3 has no tens digit) and 97 is placed in row 9. After the gathering pass, the order of the values in the one-dimensional array is 100, 3 and 97. On the third pass, 100 is placed in row 1, 3 is placed in row zero and 97 is placed in row zero (after the 3). After the last gathering pass, the original array is now in sorted order.

Note that the two-dimensional array of buckets is 10 times the size of the integer array being sorted. This sorting technique provides better performance than a insertion sort, but requires much more memory. The insertion sort requires space for only one additional element of data. This is an example of the spacetime trade-off: The bucket sort uses more memory than the insertion sort, but performs better. This version of the bucket sort requires copying all the data back to the original array on each pass. Another possibility is to create a second two-dimensional bucket array and repeatedly swap the data between the two bucket arrays.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 400 (continued)]

## Recursion Exercises

- 7.31** (Selection Sort) A [selection sort](#) searches an array looking for the smallest element. Then, the smallest element is swapped with the first element of the array. The process is repeated for the subarray beginning with the second element of the array. Each pass of the array results in one element being placed in its proper location. This sort performs comparably to the insertion sort for an array of  $n$  elements,  $n - 1$  passes must be made, and for each subarray,  $n - 1$  comparisons must be made to find the smallest value. When the subarray being processed contains one element, the array is sorted. Write recursive function `selectionSort` to perform this algorithm.
- 7.32** (Palindromes) A palindrome is a string that is spelled the same way forward and backward. Some examples of palindromes are "radar," "able was i ere i saw elba" and (if blanks are ignored) "a man a plan a canal panama." Write a recursive function `testPalindrome` that returns `true` if the string stored in the array is a palindrome, and `false` otherwise. The function should ignore spaces and punctuation in the string.
- 7.33** (Linear Search) Modify the program in [Fig. 7.19](#) to use recursive function `linearSearch` to perform a linear search of the array. The function should receive an integer array and the size of the array as arguments. If the search key is found, return the array subscript; otherwise, return 1.
- 7.34** (Eight Queens) Modify the Eight Queens program you created in [Exercise 7.26](#) to solve the problem recursively.
- 7.35** (Print an array) Write a recursive function `printArray` that takes an array, a starting subscript and an ending subscript as arguments and returns nothing. The function should stop processing and return when the starting subscript equals the ending subscript.

- 7.36** (Print a string backward) Write a recursive function `stringReverse` that takes a character array containing a string and a starting subscript as arguments, prints the string backward and returns nothing. The function should stop processing and return when the terminating null character is encountered.
- 7.37** (Find the minimum value in an array) Write a recursive function `recursiveMinimum` that takes an integer array, a starting subscript and an ending subscript as arguments, and returns the smallest element of the array. The function should stop processing and return when the starting subscript equals the ending subscript.

[!\[\]\(3817e646180c41d601a84fc654a1c93c\_img.jpg\) PREV](#)[!\[\]\(87f7e2197f2c9cedcac6c69156ead63e\_img.jpg\) NEXT](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 400 (continued)]

## vector Exercises

- 7.38** Use a `vector` of integers to solve the problem described in [Exercise 7.10](#).
- 7.39** Modify the dice-rolling program you created in [Exercise 7.17](#) to use a `vector` to store the numbers of times each possible sum of the two dice appears.
- 7.40** (Find the minimum value in a `vector`) Modify your solution to [Exercise 7.37](#) to find the minimum value in a `vector` instead of an array.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 402]

## Outline

### [8.1 Introduction](#)

### [8.2 Pointer Variable Declarations and Initialization](#)

### [8.3 Pointer Operators](#)

### [8.4 Passing Arguments to Functions by Reference with Pointers](#)

### [8.5 Using const with Pointers](#)

### [8.6 Selection Sort Using Pass-by-Reference](#)

### [8.7 sizeof Operators](#)

### [8.8 Pointer Expressions and Pointer Arithmetic](#)

### [8.9 Relationship Between Pointers and Arrays](#)

### [8.10 Arrays of Pointers](#)

### [8.11 Case Study: Card Shuffling and Dealing Simulation](#)

### [8.12 Function Pointers](#)

### [8.13 Introduction to Pointer-Based String Processing](#)

#### [8.13.1 Fundamentals of Characters and Pointer-Based Strings](#)

#### [8.13.2 String Manipulation Functions of the String-Handling Library](#)

### [8.14 Wrap-Up](#)

## [Summary](#)

## Terminology

### Self-Review Exercises

### Answers to Self-Review Exercises

## Exercises

### Special Section: Building Your Own Computer

### More Pointer Exercises

### String-Manipulation Exercises

### Special Section: Advanced String-Manipulation Exercises

### A Challenging String-Manipulation Project

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 402 (continued)]

## 8.1. Introduction

This chapter discusses one of the most powerful features of the C++ programming language, the pointer. In [Chapter 6](#), we saw that references can be used to perform pass-by-reference. Pointers also enable pass-by-reference and can be used to create and manipulate dynamic data structures (i.e., data structures that can grow and shrink), such as linked lists, queues, stacks and trees. This chapter explains basic pointer concepts and reinforces the intimate relationship among arrays and pointers. The view of arrays as pointers derives from the C programming language. As we saw in [Chapter 7](#), C++ Standard Library class `vector` provides an implementation of arrays as full-fledged objects.

Similarly, C++ actually offers two types of strings: `string` class objects (which we have been using since [Chapter 3](#)) and C-style, `char *` pointer-based strings. This chapter on pointers discusses `char *` strings to deepen your knowledge of pointers. In fact, the null-terminated strings that we introduced in [Section 7.4](#) and used in [Fig. 7.12](#) are `char *` pointer-based strings. This chapter also includes a substantial collection of string-processing exercises that use `char *` strings. C-style, `char *` pointer-based strings are widely used in legacy C and C++ systems. So, if you work with legacy C or C++ systems, you may be required to manipulate these `char *` pointer-based strings.

We will examine the use of pointers with classes in [Chapter 13](#), Object-Oriented Programming: Polymorphism, where we will see that the so-called "polymorphic processing" of object-oriented programming is performed with pointers and references. [Chapter 21](#), Data Structures, presents examples of creating and using dynamic data structures that are implemented with pointers.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 404]

## Error-Prevention Tip 8.1



Initialize pointers to prevent pointing to unknown or uninitialized areas of memory.

[PREV](#)

[\*\*NEXT\*\*](#)

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 405]

prints the value of variable `y`, namely, 5, just as the statement

```
cout << y << endl;
```

would. Using `*` in this manner is called **dereferencing a pointer**. Note that a dereferenced pointer may also be used on the left side of an assignment statement, as in

```
*yPtr = 9;
```

which would assign 9 to `y` in Fig. 8.3. The dereferenced pointer may also be used to receive an input value as in

```
cin >> *yPtr;
```

which places the input value in `y`. The dereferenced pointer is an lvalue.

### Common Programming Error 8.2



Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion, possibly with incorrect results.

### Common Programming Error 8.3



An attempt to dereference a variable that is not a pointer is a compilation error.

### Common Programming Error 8.4



Dereferencing a null pointer is normally a fatal execution-time error.

The program in Fig. 8.4 demonstrates the `&` and `*` pointer operators. Memory locations are output by `<<` in this example as hexadecimal (i.e., base-16) integers. (See Appendix D, Number Systems, for more information on hexadecimal integers.) Note that the hexadecimal memory addresses output by this program are compiler and operating-system dependent, so you may get different results when you run the program.

**Figure 8.4. Pointer operators `&` and `*`.**

(This item is displayed on page 406 in the print version)

```

1 // Fig. 8.4: fig08_04.cpp
2 // Using the & and * operators.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int a; // a is an integer
10 int *aPtr; // aPtr is an int * -- pointer to an integer
11
12 a = 7; // assigned 7 to a
13 aPtr = &a; // assign the address of a to aPtr
14
15 cout << "The address of a is " << &a
16 << "\n\nThe value of aPtr is " << aPtr;
17 cout << "\n\nThe value of a is " << a
18 << "\n\nThe value of *aPtr is " << *aPtr;
19 cout << "\n\nShowing that * and & are inverses of "
20 << "each other.\n&aPtr = " << &aPtr
21 << "\n*aPtr = " << *aPtr << endl;
22 return 0; // indicates successful termination
23 } // end main

```

```
The address of a is 0012F580
The value of aPtr is 0012F580

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&aPtr = 0012F580
*&aPtr = 0012F580
```

## Portability Tip 8.1



The format in which a pointer is output is compiler dependent. Some systems output pointer values as hexadecimal integers, while others use decimal integers.

Notice that the address of a (line 15) and the value of aPtr (line 16) are identical in the output, confirming that the address of a is indeed assigned to the pointer variable aPtr. The & and \* operators are inverses of one another when they are both applied consecutively to aPtr in either order, they "cancel one another out" and the same result (the value in aPtr) is printed.

---

[Page 406]

Figure 8.5 lists the precedence and associativity of the operators introduced to this point. Note that the address operator (&) and the dereferencing operator (\*) are unary operators on the third level of precedence in the chart.

### **Figure 8.5. Operator precedence and associativity.**

---

[Page 407]

| Operators | Associativity | Type |
|-----------|---------------|------|
|           |               |      |

|     |     |                                                 |    |    |    |   |               |                      |
|-----|-----|-------------------------------------------------|----|----|----|---|---------------|----------------------|
| ( ) | [ ] |                                                 |    |    |    |   | left to right | highest              |
| ++  | --  | <code>static_cast&lt;type&gt;( operand )</code> |    |    |    |   | 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                |

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 408]

**Figure 8.6. Pass-by-value used to cube a variable's value.**

```

1 // Fig. 8.6: fig08_06.cpp
2 // Cube a variable using pass-by-value.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int cubeByValue(int); // prototype
8
9 int main()
10 {
11 int number = 5;
12
13 cout << "The original value of number is " << number;
14
15 number = cubeByValue(number); // pass number by value to cubeByValue
16 cout << "\nThe new value of number is " << number << endl;
17 return 0; // indicates successful termination
18 } // end main
19
20 // calculate and return cube of integer argument
21 int cubeByValue(int n)
22 {
23 return n * n * n; // cube local variable n and return result
24 } // end function cubeByValue

```

The original value of number is 5  
 The new value of number is 125

**Figure 8.7. Pass-by-reference with a pointer argument used to cube a variable's value.**

(This item is displayed on page 409 in the print version)

```

1 // Fig. 8.7: fig08_07.cpp
2 // Cube a variable using pass-by-reference with a pointer argument.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void cubeByReference(int *); // prototype
8
9 int main()
10 {
11 int number = 5;
12
13 cout << "The original value of number is " << number;
14
15 cubeByReference(&number); // pass number address to cubeByReference
16
17 cout << "\nThe new value of number is " << number << endl;
18 return 0; // indicates successful termination
19 } // end main
20
21 // calculate cube of *nPtr; modifies variable number in main
22 void cubeByReference(int *nPtr)
23 {
24 *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
25 } // end function cubeByReference

```

The original value of number is 5  
The new value of number is 125

Figure 8.7 passes the variable `number` to function `cubeByReference` using pass-by-reference with a pointer argument (line 15) the address of `number` is passed to the function. Function `cubeByReference` (lines 22-25) specifies parameter `nPtr` (a pointer to `int`) to receive its argument. The function dereferences the pointer and cubes the value to which `nPtr` points (line 24). This directly changes the value of `number` in `main`.

Common Programming Error 8.5



Not dereferencing a pointer when it is necessary to do so to obtain the value to which the pointer points is an error.

A function receiving an address as an argument must define a pointer parameter to receive the address. For example, the header for function `cubeByReference` (line 22) specifies that `cubeByReference` receives the address of an `int` variable (i.e., a pointer to an `int`) as an argument, stores the address locally in `nPtr` and does not return a value.

The function prototype for `cubeByReference` (line 7) contains `int *` in parentheses. As with other variable types, it is not necessary to include names of pointer parameters in function prototypes. Parameter names included for documentation purposes are ignored by the compiler.

---

[Page 409]

Figures 8.88.9 analyze graphically the execution of the programs in Fig. 8.6 and Fig. 8.7, respectively.

**Figure 8.8. Pass-by-value analysis of the program of Fig. 8.6.**

(This item is displayed on page 410 in the print version)

[\[View full size image\]](#)

Step 1: Before main calls cubeByValue:

```
int main()
{
 int number = 5;
 number = cubeByValue(number);
}
```

number

5

```
int cubeByValue(int n)
{
 return n * n * n;
}
```

n

undefined

Step 2: After cubeByValue receives the call:

```
int main()
{
 int number = 5;
 number = cubeByValue(number);
}
```

number

5

```
int cubeByValue(int n)
{
 return n * n * n;
}
```

n

5

Step 3: After cubeByValue cubes parameter n and before cubeByValue returns to main:

```
int main()
{
 int number = 5;
 number = cubeByValue(number);
}
```

number

5

```
int cubeByValue(int n)
{
 125
 return n * n * n;
}
```

n

5

Step 4: After cubeByValue returns to main and before assigning the result to number:

```
int main()
{
 int number = 5;
 125
 number = cubeByValue(number);
}
```

number

5

```
int cubeByValue(int n)
{
 return n * n * n;
}
```

n

undefined

Step 5: After main completes the assignment to number:

```
int main()
{
 int number = 5;
 125
 125
 number = cubeByValue(number);
}
```

number

125

```
int cubeByValue(int n)
{
 return n * n * n;
}
```

n

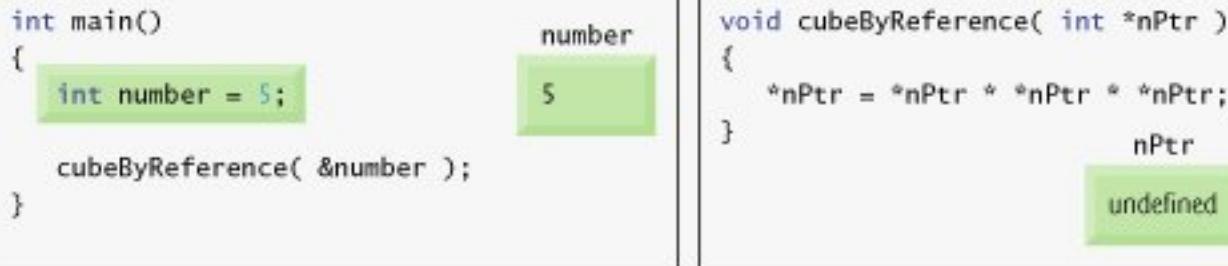
undefined

**Figure 8.9. Pass-by-reference analysis (with a pointer argument) of the program of Fig. 8.7.**

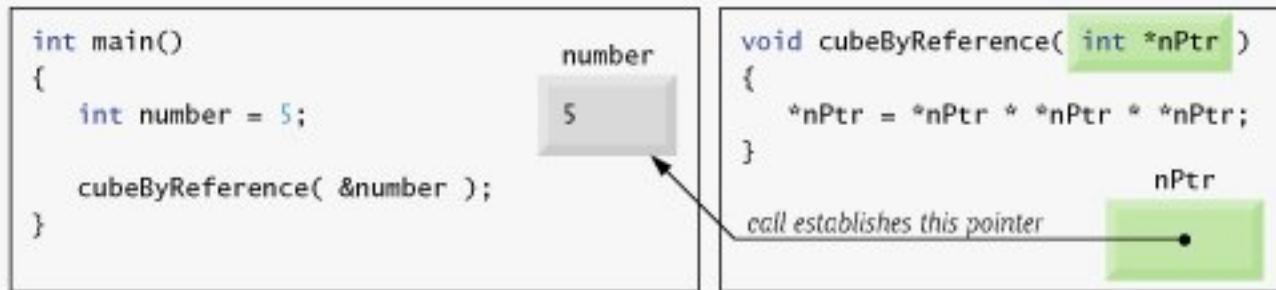
(This item is displayed on page 411 in the print version)

[\[View full size image\]](#)

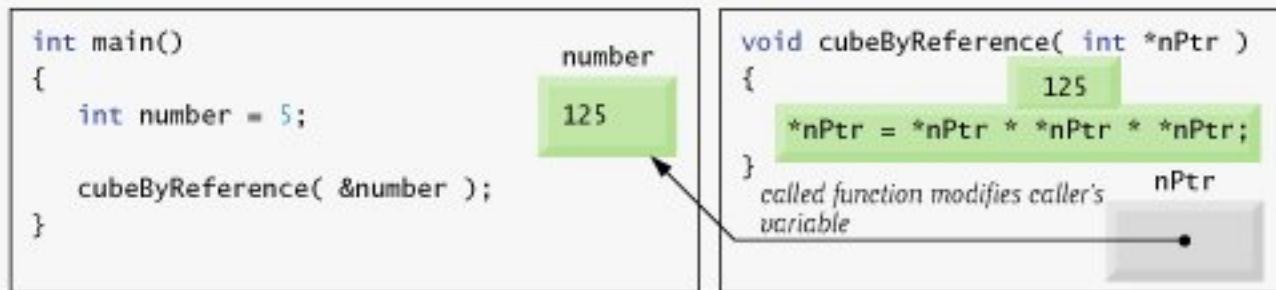
Step 1: Before main calls cubeByReference:



Step 2: After cubeByReference receives the call and before \*nPtr is cubed:



Step 3: After \*nPtr is cubed and before program control returns to main:

**Software Engineering Observation 8.1**

Use pass-by-value to pass arguments to a function unless the caller explicitly requires that the called function directly modify the value of the argument variable in the caller. This is another example of the principle of least privilege.

In the function header and in the prototype for a function that expects a one-dimensional array as an argument, the pointer notation in the parameter list of cubeByReference may be used. The compiler

does not differentiate between a function that receives a pointer and a function that receives a one-dimensional array. This, of course, means that the function must "know" when it is receiving an array or simply a single variable for which it is to perform pass-by-reference. When the compiler encounters a function parameter for a one-dimensional array of the form `int b[ ]`, the compiler converts the parameter to the pointer notation `int *b` (pronounced "b is a pointer to an integer"). Both forms of declaring a function parameter as a one-dimensional array are interchangeable.

 PREVNEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 412]

**Chapter 6** explained that when a function is called using pass-by-value, a copy of the argument (or arguments) in the function call is made and passed to the function. If the copy is modified in the function, the original value is maintained in the caller without change. In many cases, a value passed to a function is modified so the function can accomplish its task. However, in some instances, the value should not be altered in the called function, even though the called function manipulates only a copy of the original value.

For example, consider a function that takes a one-dimensional array and its size as arguments and subsequently prints the array. Such a function should loop through the array and output each array element individually. The size of the array is used in the function body to determine the highest subscript of the array so the loop can terminate when the printing completes. The size of the array does not change in the function body, so it should be declared `const`. Of course, because the array is only being printed, it, too, should be declared `const`. This is especially important because an entire array is always passed by reference and could easily be changed in the called function.

#### Software Engineering Observation 8.2



If a value does not (or should not) change in the body of a function to which it is passed, the parameter should be declared `const` to ensure that it is not accidentally modified.

If an attempt is made to modify a `const` value, a warning or an error is issued, depending on the particular compiler.

#### Error-Prevention Tip 8.2



Before using a function, check its function prototype to determine the parameters that it can modify.

There are four ways to pass a pointer to a function: a nonconstant pointer to nonconstant data ([Fig. 8.10](#)), a nonconstant pointer to constant data ([Fig. 8.11](#) and [Fig. 8.12](#)), a constant pointer to nonconstant data ([Fig. 8.13](#)) and a constant pointer to constant data ([Fig. 8.14](#)). Each combination provides a different level of access privileges.

**Figure 8.10. Converting a string to uppercase.**

(This item is displayed on page 413 in the print version)

```

1 // Fig. 8.10: fig08_10.cpp
2 // Converting lowercase letters to uppercase letters
3 // using a non-constant pointer to non-constant data.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <cctype> // prototypes for islower and toupper
9 using std::islower;
10 using std::toupper;
11
12 void convertToUppercase(char *);
13
14 int main()
15 {
16 char phrase[] = "characters and $32.98";
17
18 cout << "The phrase before conversion is: " << phrase;
19 convertToUppercase(phrase);
20 cout << "\nThe phrase after conversion is: " << phrase << endl;
21 return 0; // indicates successful termination
22 } // end main
23
24 // convert string to uppercase letters
25 void convertToUppercase(char *sPtr)
26 {
27 while (*sPtr != '\0') // loop while current character is not '\0'
28 {
29 if (islower(*sPtr)) // if character is lowercase,
30 *sPtr = toupper(*sPtr); // convert to uppercase
31
32 sPtr++; // move sPtr to next character in string
33 } // end while
34 } // end function convertToUppercase

```

The phrase before conversion is: characters and \$32.98  
 The phrase after conversion is: CHARACTERS AND \$32.98

Figure 8.11. Printing a string one character at a time using a nonconstant pointer to constant data.

(This item is displayed on page 414 in the print version)

```
1 // Fig. 8.11: fig08_11.cpp
2 // Printing a string one character at a time using
3 // a non-constant pointer to constant data.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 void printCharacters(const char *); // print using pointer to const data
9
10 int main()
11 {
12 const char phrase[] = "print characters of a string";
13
14 cout << "The string is:\n";
15 printCharacters(phrase); // print characters in phrase
16 cout << endl;
17 return 0; // indicates successful termination
18 } // end main
19
20 // sPtr can be modified, but it cannot modify the character to which
21 // it points, i.e., sPtr is a "read-only" pointer
22 void printCharacters(const char *sPtr)
23 {
24 for (; *sPtr != '\0'; sPtr++) // no initialization
25 cout << *sPtr; // display character without modification
26 } // end function printCharacters
```

The string is:  
print characters of a string

**Figure 8.12. Attempting to modify data through a nonconstant pointer to constant data.**

(This item is displayed on page 415 in the print version)

```

1 // Fig. 8.12: fig08_12.cpp
2 // Attempting to modify data through a
3 // non-constant pointer to constant data.
4
5 void f(const int *); // prototype
6
7 int main()
8 {
9 int y;
10
11 f(&y); // f attempts illegal modification
12 return 0; // indicates successful termination
13 } // end main
14
15 // xPtr cannot modify the value of constant variable to which it points
16 void f(const int *xPtr)
17 {
18 *xPtr = 100; // error: cannot modify a const object
19 } // end function f

```

Borland C++ command-line compiler error message:

```
Error E2024 fig08_12.cpp 18:
 Cannot modify a const object in function f(const int *)
```

Microsoft Visual C++ compiler error message:

```
c:\cpphttp5_examples\ch08\Fig08_12\fig08_12.cpp(18) :
 error C2166: l-value specifies const object
```

GNU C++ compiler error message:

```
fig08_12.cpp: In function `void f(const int*)':
fig08_12.cpp:18: error: assignment of read-only location
```

**Figure 8.13. Attempting to modify a constant pointer to nonconstant data.**

(This item is displayed on page 416 in the print version)

```

1 // Fig. 8.13: fig08_13.cpp
2 // Attempting to modify a constant pointer to non-constant data.
3
4 int main()
5 {
6 int x, y;
7
8 // ptr is a constant pointer to an integer that can
9 // be modified through ptr, but ptr always points to the
10 // same memory location.
11 int * const ptr = &x; // const pointer must be initialized
12
13 *ptr = 7; // allowed: *ptr is not const
14 ptr = &y; // error: ptr is const; cannot assign to it a new address
15 return 0; // indicates successful termination
16 } // end main

```

Borland C++ command-line compiler error message:

```
Error E2024 fig08_13.cpp 14: Cannot modify a const object in function main()
```

Microsoft Visual C++ compiler error message:

```
c:\cpphttp5e_examples\ch08\Fig08_13\fig08_13.cpp(14) : error C2166:
 l-value specifies const object
```

GNU C++ compiler error message:

```
fig08_13.cpp: In function `int main()':
fig08_13.cpp:14: error: assignment of read-only variable `ptr'
```

**Figure 8.14. Attempting to modify a constant pointer to constant data.**

(This item is displayed on pages 417 - 418 in the print version)

```

1 // Fig. 8.14: fig08_14.cpp
2 // Attempting to modify a constant pointer to constant data.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int x = 5, y;
10
11 // ptr is a constant pointer to a constant integer.
12 // ptr always points to the same location; the integer
13 // at that location cannot be modified.
14 const int *const ptr = &x;
15
16 cout << *ptr << endl;
17
18 *ptr = 7; // error: *ptr is const; cannot assign new value
19 ptr = &y; // error: ptr is const; cannot assign new address
20
21 } // end main

```

Borland C++ command-line compiler error message:

```
Error E2024 fig08_14.cpp 18: Cannot modify a const object in function main()
Error E2024 fig08_14.cpp 19: Cannot modify a const object in function main()
```

Microsoft Visual C++ compiler error message:

```
c:\cpphttp5e_examples\ch08\Fig08_14\fig08_14.cpp(18) : error C2166:
 l-value specifies const object
c:\cpphttp5e_examples\ch08\Fig08_14\fig08_14.cpp(19) : error C2166:
 l-value specifies const object
```

GNU C++ compiler error message:

```
fig08_14.cpp: In function `int main()':
fig08_14.cpp:18: error: assignment of read-only location
fig08_14.cpp:19: error: assignment of read-only variable `ptr'
```

## Nonconstant Pointer to Nonconstant Data

The highest access is granted by a **nonconstant pointer to nonconstant data**the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data. The declaration for a nonconstant pointer to nonconstant data does not include `const`. Such a pointer can be used to receive a null-terminated string in a function that changes the pointer value to process (and possibly modify) each character in the string. Recall from [Section 7.4](#) that a null-terminated string can be placed in a character array that contains the characters of the string and a null character indicating where the string ends.

In [Fig. 8.10](#), function `convertToUppercase` (lines 2534) declares parameter `sPtr` (line 25) to be a nonconstant pointer to nonconstant data (again, `const` is not used). The function processes one character at a time from the null-terminated string stored in character array `phrase` (lines 2733). Keep in mind that a character array's name is really equivalent to a pointer to the first character of the array, so passing `phrase` as an argument to `convertToUppercase` is possible. Function `islower` (line 29) takes a character argument and returns true if the character is a lowercase letter and false otherwise. Characters in the range '`a`' through '`z`' are converted to their corresponding uppercase letters by function `toupper` (line 30); others remain unchangedfunction `toupper` takes one character as an argument. If the character is a lowercase letter, the corresponding uppercase letter is returned; otherwise, the original character is returned. Function `toupper` and function `islower` are part of the character-handling library `<cctype>` (see [Chapter 22](#), Bits, Characters, C-Strings and structs). After processing one character, line 32 increments `sPtr` by 1 (this would not be possible if `sPtr` were declared `const`). When operator `++` is applied to a pointer that points to an array, the memory address stored in the pointer is modified to point to the next element of the array (in this case, the next character in the string). Adding one to a pointer is one valid operation in **pointer arithmetic**, which is covered in detail in [Section 8.8](#) and [Section 8.9](#).

---

[Page 413]

---

[Page 414]

## Nonconstant Pointer to Constant Data

A **nonconstant pointer to constant data** is a pointer that can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified through that pointer. Such a pointer might be used to receive an array argument to a function that will process each element of the array, but

should not be allowed to modify the data. For example, function `printCharacters` (lines 2226 of Fig. 8.11) declares parameter `sPtr` (line 22) to be of type `const char *`, so that it can receive a null-terminated pointer-based string. The declaration is read from right to left as "`sPtr` is a pointer to a character constant." The body of the function uses a `for` statement (lines 2425) to output each character in the string until the null character is encountered. After each character is printed, pointer `sPtr` is incremented to point to the next character in the string (this works because the pointer is not `const`). Function `main` creates `char` array `phrase` to be passed to `printCharacters`. Again, we can pass the array `phrase` to `printCharacters` because the name of the array is really a pointer to the first character in the array.

[Figure 8.12](#) demonstrates the compilation error messages produced when attempting to compile a function that receives a nonconstant pointer to constant data, then tries to use that pointer to modify the data. [Note: Remember that compiler error messages vary among compilers.]

---

[Page 415]

As we know, arrays are aggregate data types that store related data items of the same type under one name. When a function is called with an array as an argument, the array is passed to the function by reference. However, objects are always passed by valuea copy of the entire object is passed. This requires the execution-time overhead of making a copy of each data item in the object and storing it on the function call stack. When an object must be passed to a function, we can use a pointer to constant data (or a reference to constant data) to get the performance of pass-by-reference and the protection of pass-by-value. When a pointer to an object is passed, only a copy of the address of the object must be made; the object itself is not copied. On a machine with four-byte addresses, a copy of four bytes of memory is made rather than a copy of a possibly large object.

#### Performance Tip 8.1



If they do not need to be modified by the called function, pass large objects using pointers to constant data or references to constant data, to obtain the performance benefits of pass-by-reference.

---

[Page 416]

#### Software Engineering Observation 8.3



Pass large objects using pointers to constant data, or references to constant data, to obtain the security of pass-by-value.

## Constant Pointer to Nonconstant Data

A **constant pointer to nonconstant data** is a pointer that always points to the same memory location; the data at that location can be modified through the pointer. This is the default for an array name. An array name is a constant pointer to the beginning of the array. All data in the array can be accessed and changed by using the array name and array subscripting. A constant pointer to nonconstant data can be used to receive an array as an argument to a function that accesses array elements using array subscript notation. Pointers that are declared `const` must be initialized when they are declared. (If the pointer is a function parameter, it is initialized with a pointer that is passed to the function.) The program of Fig. 8.13 attempts to modify a constant pointer. Line 11 declares pointer `ptr` to be of type `int * const`. The declaration in the figure is read from right to left as "ptr is a constant pointer to a nonconstant integer." The pointer is initialized with the address of integer variable `x`. Line 14 attempts to assign the address of `y` to `ptr`, but the compiler generates an error message. Note that no error occurs when line 13 assigns the value 7 to `*ptr`—the nonconstant value to which `ptr` points can be modified using the dereferenced `ptr`, even though `ptr` itself has been declared `const`.

[Page 417]

## Common Programming Error 8.6



Not initializing a pointer that is declared `const` is a compilation error.

## Constant Pointer to Constant Data

The least amount of access privilege is granted by a **constant pointer to constant data**. Such a pointer always points to the same memory location, and the data at that memory location cannot be modified using the pointer. This is how an array should be passed to a function that only reads the array, using array subscript notation, and does not modify the array. The program of Fig. 8.14 declares pointer variable `ptr` to be of type `const int * const` (line 14). This declaration is read from right to left as "ptr is a constant pointer to an integer constant." The figure shows the error messages generated when an attempt is made to modify the data to which `ptr` points (line 18) and when an attempt is made to modify the address stored in the pointer variable (line 19). Note that no errors occur when the program attempts to dereference `ptr`, or when the program attempts to output the value to which `ptr` points (line 16), because neither the pointer nor the data it points to is being modified in this statement.

[Page 418]

PREV

page footer

NEXT

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 419]

**Figure 8.15** implements selection sort using two functions `selectionSort` and `swap`. Function `selectionSort` (lines 3653) sorts the array. Line 38 declares the variable `smallest`, which will store the index of the smallest element in the remaining array. Lines 4152 loop size - 1 times. Line 43 sets the index of the smallest element to the current index. Lines 4649 loop over the remaining elements in the array. For each of these elements, line 48 compares its value to the value of the smallest element. If the current element is smaller than the smallest element, line 49 assigns the current element's index to `smallest`. When this loop finishes, `smallest` will contain the index of the smallest element in the remaining array. Line 51 calls function `swap` (lines 5762) to place the smallest remaining element in the next spot in the array (i.e., exchange the array elements `array[ i ]` and `array[ smallest ]`).

**Figure 8.15. Selection sort with pass-by-reference.**

(This item is displayed on pages 419 - 420 in the print version)

```

1 // Fig. 8.15: fig08_15.cpp
2 // This program puts values into an array, sorts the values into
3 // ascending order and prints the resulting array.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 void selectionSort(int * const, const int); // prototype
12 void swap(int * const, int * const); // prototype
13
14 int main()
15 {
16 const int arraySize = 10;
17 int a[arraySize] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19 cout << "Data items in original order\n";
20
21 for (int i = 0; i < arraySize; i++)
22 cout << setw(4) << a[i];
23
24 selectionSort(a, arraySize); // sort the array
25
26 cout << "\nData items in ascending order\n";

```

```

27
28 for (int j = 0; j < arraySize; j++)
29 cout << setw(4) << a[j];
30
31 cout << endl;
32 return 0; // indicates successful termination
33 } // end main
34
35 // function to sort an array
36 void selectionSort(int * const array, const int size)
37 {
38 int smallest; // index of smallest element
39
40 // loop over size - 1 elements
41 for (int i = 0; i < size - 1; i++)
42 {
43 smallest = i; // first index of remaining array
44
45 // loop to find index of smallest element
46 for (int index = i + 1; index < size; index++)
47
48 if (array[index] < array[smallest])
49 smallest = index;
50
51 swap(&array[i], &array[smallest]);
52 } // end if
53 } // end function selectionSort
54
55 // swap values at memory locations to which
56 // element1Ptr and element2Ptr point
57 void swap(int * const element1Ptr, int * const element2Ptr)
58 {
59 int hold = *element1Ptr;
60 *element1Ptr = *element2Ptr;
61 *element2Ptr = hold;
62 } // end function swap

```

Data items in original order  
2 6 4 8 10 12 89 68 45 37  
Data items in ascending order  
2 4 6 8 10 12 37 45 68 89

Let us now look more closely at function `swap`. Remember that C++ enforces information hiding between functions, so `swap` does not have access to individual array elements in `selectionSort`. Because `selectionSort` wants `swap` to have access to the array elements to be swapped, `selectionSort` passes each of these elements to `swap` by reference—the address of each array element is passed explicitly. Although entire arrays are passed by reference, individual array elements are scalars and are ordinarily passed by value. Therefore, `selectionSort` uses the address operator (`&`) on each array element in the `swap` call (line 51) to effect pass-by-reference. Function `swap` (lines 5762) receives `&array[ i ]` in pointer variable `element1Ptr`. Information hiding prevents `swap` from "knowing" the name `array[ i ]`, but `swap` can use `*element1Ptr` as a synonym for `array[ i ]`. Thus, when `swap` references `*element1Ptr`, it is actually referencing `array[ i ]` in `selectionSort`. Similarly, when `swap` references `*element2Ptr`, it is actually referencing `array[ smallest ]` in `selectionSort`.

---

[Page 420]

Even though `swap` is not allowed to use the statements

```
hold = array[i];
array[i] = array[smallest];
array[smallest] = hold;
```

---

[Page 421]

precisely the same effect is achieved by

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

in the `swap` function of [Fig. 8.15](#).

Several features of function `selectionSort` should be noted. The function header (line 36) declares `array` as `int * const array`, rather than `int array[]`, to indicate that function `selectionSort` receives a one-dimensional array as an argument. Both parameter `array`'s pointer and parameter `size` are declared `const` to enforce the principle of least privilege. Although parameter `size` receives a copy of a value in `main` and modifying the copy cannot change the value in `main`, `selectionSort` does not need to alter `size` to accomplish its task—the array size remains fixed during the execution of `selectionSort`. Therefore, `size` is declared `const` to ensure that it is not modified. If the size of the array were to be modified during the sorting process, the sorting algorithm would not run correctly.

Note that function `selectionSort` receives the size of the array as a parameter, because the function

must have that information to sort the array. When a pointer-based array is passed to a function, only the memory address of the first element of the array is received by the function; the array size must be passed separately to the function.

By defining function `selectionSort` to receive the array size as a parameter, we enable the function to be used by any program that sorts one-dimensional `int` arrays of arbitrary size. The size of the array could have been programmed directly into the function, but this would restrict the function to processing an array of a specific size and reduce the function's reusability only programs processing one-dimensional `int` arrays of the specific size "hard coded" into the function could use the function.

#### Software Engineering Observation 8.4



When passing an array to a function, also pass the size of the array (rather than building into the function knowledge of the array size). This makes the function more reusable.

[◀ PREV](#)[NEXT ▶](#)**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 422]

[Note: When the Borland C++ compiler is used to compile Fig. 8.16, the compiler generates the warning message "Parameter 'ptr' is never used in function get-Size(double \*)." This warning occurs because sizeof is actually a compile-time operator; thus, variable ptr is not used in the function's body at execution time. Many compilers issue warnings like this to let you know that a variable is not being used so that you can either remove it from your code or modify your code to use the variable properly. Similar messages occur in Fig. 8.17 with various compilers.]

**Figure 8.17. sizeof operator used to determine standard data type sizes.**

(This item is displayed on page 423 in the print version)

```

1 // Fig. 8.17: fig08_17.cpp
2 // Demonstrating the sizeof operator.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 char c; // variable of type char
10 short s; // variable of type short
11 int i; // variable of type int
12 long l; // variable of type long
13 float f; // variable of type float
14 double d; // variable of type double
15 long double ld; // variable of type long double
16 int array[20]; // array of int
17 int *ptr = array; // variable of type int *
18
19 cout << "sizeof c = " << sizeof c
20 << "\nsizeof(char) = " << sizeof(char)
21 << "\nsizeof s = " << sizeof s
22 << "\nsizeof(short) = " << sizeof(short)
23 << "\nsizeof i = " << sizeof i
24 << "\nsizeof(int) = " << sizeof(int)
25 << "\nsizeof l = " << sizeof l
26 << "\nsizeof(long) = " << sizeof(long)
27 << "\nsizeof f = " << sizeof f
28 << "\nsizeof(float) = " << sizeof(float)
29 << "\nsizeof d = " << sizeof d

```

```

30 << "\tsizeof(double) = " << sizeof(double)
31 << "\nsizeof ld = " << sizeof ld
32 << "\tsizeof(long double) = " << sizeof(long double)
33 << "\nsizeof array = " << sizeof array
34 << "\nsizeof ptr = " << sizeof ptr << endl;
35 return 0; // indicates successful termination
36 } // end main

```

```

sizeof c = 1 sizeof(char) = 1
sizeof s = 2 sizeof(short) = 2
sizeof i = 4 sizeof(int) = 4
sizeof l = 4 sizeof(long) = 4
sizeof f = 4 sizeof(float) = 4
sizeof d = 8 sizeof(double) = 8
sizeof ld = 8 sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4

```

The number of elements in an array also can be determined using the results of two `sizeof` operations. For example, consider the following array declaration:

```
double realArray[22];
```

If variables of data type `double` are stored in eight bytes of memory, array `realArray` contains a total of 176 bytes. To determine the number of elements in the array, the following expression can be used:

```
sizeof realArray / sizeof(double) // calculate number of elements
```

The expression determines the number of bytes in array `realArray` (176) and divides that value by the number of bytes used in memory to store a `double` value (8); the result is the number of elements in `realArray` (22).

The program of Fig. 8.17 uses the `sizeof` operator to calculate the number of bytes used to store most of the standard data types. Notice that, in the output, the types `double` and `long double` have the same size. Types may have different sizes based on the system the program is run on. On another system, for example, `double` and `long double` may be defined to be of different sizes.

[Page 424]

### Portability Tip 8.3



The number of bytes used to store a particular data type may vary between systems. When writing programs that depend on data type sizes, and that will run on several computer systems, use `sizeof` to determine the number of bytes used to store the data types.

Operator `sizeof` can be applied to any variable name, type name or constant value. When `sizeof` is applied to a variable name (which is not an array name) or a constant value, the number of bytes used to store the specific type of variable or constant is returned. Note that the parentheses used with `sizeof` are required only if a type name (e.g., `int`) is supplied as its operand. The parentheses used with `sizeof` are not required when `sizeof`'s operand is a variable name or constant. Remember that `sizeof` is an operator, not a function, and that it has its effect at compile time, not execution time.

### Common Programming Error 8.8



Omitting the parentheses in a `sizeof` operation when the operand is a type name is a compilation error.

### Performance Tip 8.2



Because `sizeof` is a compile-time unary operator, not an execution-time operator, using `sizeof` does not negatively impact execution performance.

### Error-Prevention Tip 8.3



To avoid errors associated with omitting the parentheses around the operand of operator `sizeof`, many programmers include parentheses around every `sizeof` operand.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 425]

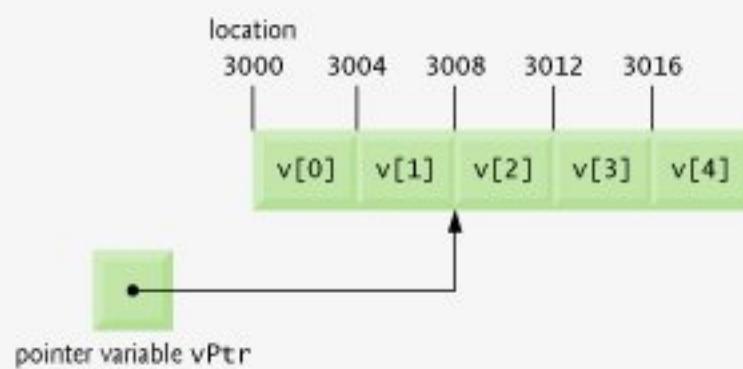
In conventional arithmetic, the addition  $3000 + 2$  yields the value 3002. This is normally not the case with pointer arithmetic. When an integer is added to, or subtracted from, a pointer, the pointer is not simply incremented or decremented by that integer, but by that integer times the size of the object to which the pointer refers. The number of bytes depends on the object's data type. For example, the statement

```
vPtr += 2;
```

would produce 3008 ( $3000 + 2 * 4$ ), assuming that an int is stored in four bytes of memory. In the array v, vPtr would now point to v[ 2 ] (Fig. 8.19). If an integer is stored in two bytes of memory, then the preceding calculation would result in memory location 3004 ( $3000 + 2 * 2$ ). If the array were of a different data type, the preceding statement would increment the pointer by twice the number of bytes it takes to store an object of that data type. When performing pointer arithmetic on a character array, the results will be consistent with regular arithmetic, because each character is one byte long.

**Figure 8.19. Pointer vPtr after pointer arithmetic.**

[[View full size image](#)]



If vPtr had been incremented to 3016, which points to v[ 4 ], the statement

```
vPtr -= 4;
```

would set vPtr back to 3000 the beginning of the array. If a pointer is being incremented or decremented by one, the increment (++) and decrement (--) operators can be used. Each of the

## statements

```
++vPtr;
vPtr++;
```

[Page 426]

increments the pointer to point to the next element of the array. Each of the statements

```
--vPtr;
vPtr--;
```

decrements the pointer to point to the previous element of the array.

Pointer variables pointing to the same array may be subtracted from one another. For example, if vPtr contains the location 3000 and v2Ptr contains the address 3008, the statement

```
x = v2Ptr - vPtr;
```

would assign to x the number of array elements from vPtr to v2Ptr in this case, 2. Pointer arithmetic is meaningless unless performed on a pointer that points to an array. We cannot assume that two variables of the same type are stored contiguously in memory unless they are adjacent elements of an array.

### Common Programming Error 8.9



Using pointer arithmetic on a pointer that does not refer to an array of values is a logic error.

### Common Programming Error 8.10



Subtracting or comparing two pointers that do not refer to elements of the same array is a logic error.

### Common Programming Error 8.11



Using pointer arithmetic to increment or *decrement a pointer* such that the pointer refers to an element past the end of the array or before the beginning of the array is normally a logic error.

A pointer can be assigned to another pointer if both pointers are of the same type. Otherwise, a cast operator must be used to convert the value of the pointer on the right of the assignment to the pointer type on the left of the assignment. The exception to this rule is the pointer to `void` (i.e., `void *`), which is a generic pointer capable of representing any pointer type. All pointer types can be assigned to a pointer of type `void *` without casting. However, a pointer of type `void *` cannot be assigned directly to a pointer of another type—the pointer of type `void *` must first be cast to the proper pointer type.

### Software Engineering Observation 8.5



Nonconstant pointer arguments can be passed to constant pointer parameters. This is helpful when the body of a program uses a nonconstant pointer to access data, but does not want that data to be modified by a function called in the body of the program.

A `void *` pointer cannot be dereferenced. For example, the compiler "knows" that a pointer to `int` refers to four bytes of memory on a machine with four-byte integers, but a pointer to `void` simply contains a memory address for an unknown data type—the precise number of bytes to which the pointer refers and the type of the data are not known by the compiler. The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer for a pointer to `void`, this number of bytes cannot be determined from the type.

[Page 427]

### Common Programming Error 8.12



Assigning a pointer of one type to a pointer of another (other than `void *`) without casting the first pointer to the type of the second pointer is a compilation error.

### Common Programming Error 8.13



All operations on a `void *` pointer are compilation errors, except comparing `void *` pointers with other pointers, casting `void *` pointers to valid pointer types and assigning addresses to `void *` pointers.

Pointers can be compared using equality and relational operators. Comparisons using relational operators are meaningless unless the pointers point to members of the same array. Pointer comparisons compare the addresses stored in the pointers. A comparison of two pointers pointing to the same array could show, for example, that one pointer points to a higher numbered element of the array than the other pointer does. A common use of pointer comparison is determining whether a pointer is 0 (i.e., the pointer is a null pointer or it does not point to anything).

 PREVNEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 428]

The array name can be treated as a pointer and used in pointer arithmetic. For example, the expression

```
* (b + 3)
```

also refers to the array element `b[ 3 ]`. In general, all subscripted array expressions can be written with a pointer and an offset. In this case, pointer/offset notation was used with the name of the array as a pointer. Note that the preceding expression does not modify the array name in any way; `b` still points to the first element in the array.

Pointers can be subscripted exactly as arrays can. For example, the expression

```
bPtr[1]
```

refers to the array element `b[ 1 ]`; this expression uses [pointer/subscript notation](#).

Remember that an array name is a constant pointer; it always points to the beginning of the array. Thus, the expression

```
b += 3
```

causes a compilation error, because it attempts to modify the value of the array name (a constant) with pointer arithmetic.

#### Common Programming Error 8.14



Although array names are pointers to the beginning of the array and pointers can be modified in arithmetic expressions, array names cannot be modified in arithmetic expressions, because array names are constant pointers.

#### Good Programming Practice 8.2



For clarity, use array notation instead of pointer notation when manipulating arrays.

Figure 8.20 uses the four notations discussed in this section for referring to array elementsarray subscript notation, pointer/offset notation with the array name as a pointer, pointer subscript notation and pointer/offset notation with a pointerto accomplish the same task, namely printing the four elements of the integer array b.

### Figure 8.20. Referencing array elements with the array name and with pointers.

(This item is displayed on pages 428 - 429 in the print version)

```

1 // Fig. 8.20: fig08_20.cpp
2 // Using subscripting and pointer notations with arrays.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int b[] = { 10, 20, 30, 40 }; // create 4-element array b
10 int *bPtr = b; // set bPtr to point to array b
11
12 // output array b using array subscript notation
13 cout << "Array b printed with:\n\nArray subscript notation\n";
14
15 for (int i = 0; i < 4; i++)
16 cout << "b[" << i << "] = " << b[i] << '\n';
17
18 // output array b using the array name and pointer/offset notation
19 cout << "\nPointer/offset notation where "
20 << "the pointer is the array name\n";
21
22 for (int offset1 = 0; offset1 < 4; offset1++)
23 cout << "*(" << b + offset1 << ") = " << *(b + offset1) << '\n';
24
25 // output array b using bPtr and array subscript notation
26 cout << "\nPointer subscript notation\n";
27
28 for (int j = 0; j < 4; j++)
29 cout << "bPtr[" << j << "] = " << bPtr[j] << '\n';
30
31 cout << "\nPointer/offset notation\n";
32

```

```

33 // output array b using bPtr and pointer/offset notation
34 for (int offset2 = 0; offset2 < 4; offset2++)
35 cout << "*("bPtr + " " << offset2 << ") = "
36 << *(bPtr + offset2) << '\n';
37
38 return 0; // indicates successful termination
39 } // end main

```

Array b printed with:

Array subscript notation

```

b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

```

Pointer/offset notation where the pointer is the array name

```

*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

```

Pointer subscript notation

```

bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

```

Pointer/offset notation

```

*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40

```

[Page 430]

To further illustrate the interchangeability of arrays and pointers, let us look at the two string-copying functions `copy1` and `copy2` in the program of Fig. 8.21. Both functions copy a string into a character array. After a comparison of the function prototypes for `copy1` and `copy2`, the functions appear identical (because of the interchangeability of arrays and pointers). These functions accomplish the same task, but

they are implemented differently.

**Figure 8.21. String copying using array notation and pointer notation.**

```

1 // Fig. 8.21: fig08_21.cpp
2 // Copying a string using array notation and pointer notation.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void copy1(char *, const char *); // prototype
8 void copy2(char *, const char *); // prototype
9
10 int main()
11 {
12 char string1[10];
13 char *string2 = "Hello";
14 char string3[10];
15 char string4[] = "Good Bye";
16
17 copy1(string1, string2); // copy string2 into string1
18 cout << "string1 = " << string1 << endl;
19
20 copy2(string3, string4); // copy string4 into string3
21 cout << "string3 = " << string3 << endl;
22 return 0; // indicates successful termination
23 } // end main
24
25 // copy s2 to s1 using array notation
26 void copy1(char * s1, const char * s2)
27 {
28 // copying occurs in the for header
29 for (int i = 0; (s1[i] = s2[i]) != '\0'; i++)
30 ; // do nothing in body
31 } // end function copy1
32
33 // copy s2 to s1 using pointer notation
34 void copy2(char *s1, const char *s2)
35 {
36 // copying occurs in the for header
37 for (; (*s1 = *s2) != '\0'; s1++, s2++)
38 ; // do nothing in body
39 } // end function copy2

```

```
string1 = Hello
string3 = Good Bye
```

[Page 431]

Function `copy1` (lines 2631) uses array subscript notation to copy the string in `s2` to the character array `s1`. The function declares an integer counter variable `i` to use as the array subscript. The `for` statement header (line 29) performs the entire copy operation its body is the empty statement. The header specifies that `i` is initialized to zero and incremented by one on each iteration of the loop. The condition in the `for, ( s1[ i ] = s2[ i ] ) != '\0'`, performs the copy operation character by character from `s2` to `s1`. When the null character is encountered in `s2`, it is assigned to `s1`, and the loop terminates, because the null character is equal to `'\0'`. Remember that the value of an assignment statement is the value assigned to its left operand.

Function `copy2` (lines 3439) uses pointers and pointer arithmetic to copy the string in `s2` to the character array `s1`. Again, the `for` statement header (line 37) performs the entire copy operation. The header does not include any variable initialization. As in function `copy1`, the condition `( *s1 = *s2 ) != '\0'` performs the copy operation. Pointer `s2` is dereferenced, and the resulting character is assigned to the dereferenced pointer `s1`. After the assignment in the condition, the loop increments both pointers, so they point to the next element of array `s1` and the next character of string `s2`, respectively. When the loop encounters the null character in `s2`, the null character is assigned to the dereferenced pointer `s1` and the loop terminates. Note that the "increment portion" of this `for` statement has two increment expressions separated by a comma operator.

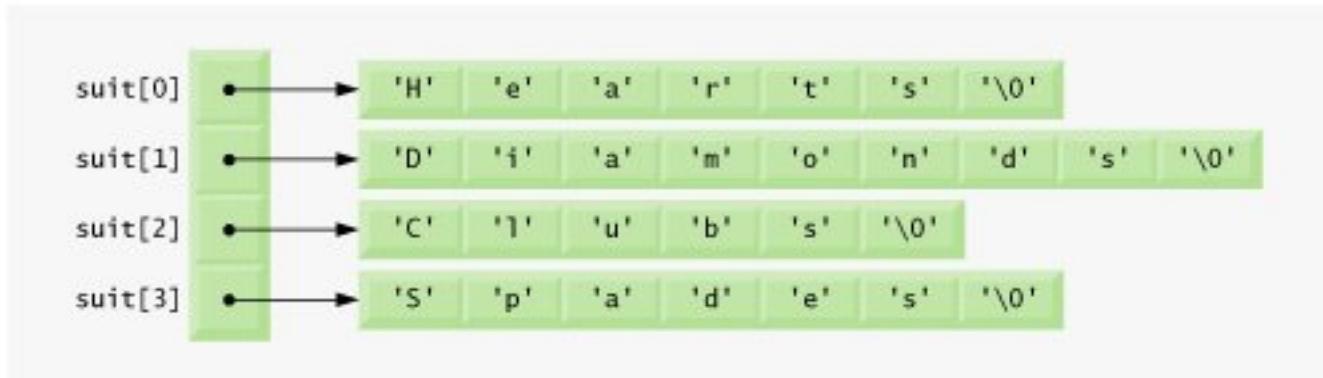
The first argument to both `copy1` and `copy2` must be an array large enough to hold the string in the second argument. Otherwise, an error may occur when an attempt is made to write into a memory location beyond the bounds of the array (recall that when using pointer-based arrays, there is no "built-in" bounds checking). Also, note that the second parameter of each function is declared as `const char *` (a pointer to a character constant i.e., a constant string). In both functions, the second argument is copied into the first argument characters are copied from the second argument one at a time, but the characters are never modified. Therefore, the second parameter is declared to point to a constant value to enforce the principle of least privilege neither function needs to modify the second argument, so neither function is allowed to modify the second argument.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 432]

**Figure 8.22. Graphical representation of the suit array.**

[\[View full size image\]](#)



The suit strings could be placed into a two-dimensional array, in which each row represents one suit and each column represents one of the letters of a suit name. Such a data structure must have a fixed number of columns per row, and that number must be as large as the largest string. Therefore, considerable memory is wasted when we store a large number of strings, of which most are shorter than the longest string. We use arrays of strings to help represent a deck of cards in the next section.

String arrays are commonly used with **command-line arguments** that are passed to function `main` when a program begins execution. Such arguments follow the program name when a program is executed from the command line. A typical use of command-line arguments is to pass options to a program. For example, from the command line on a Windows computer, the user can type

```
dir /P
```

to list the contents of the current directory and pause after each screen of information. When the `dir` command executes, the option `/P` is passed to `dir` as a command-line argument. Such arguments are placed in a string array that `main` receives as an argument. We discuss command-line arguments in [Appendix E, C Legacy Code Topics](#).

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 433]

**Figure 8.23. Two-dimensional array representation of a deck of cards.**

[\[View full size image\]](#)

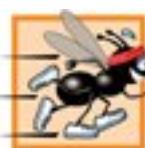
|          | Ace | Two | Three | Four | Five | Six | Seven | Eight | Nine | Ten | Jack | Queen | King |
|----------|-----|-----|-------|------|------|-----|-------|-------|------|-----|------|-------|------|
|          | 0   | 1   | 2     | 3    | 4    | 5   | 6     | 7     | 8    | 9   | 10   | 11    | 12   |
| Hearts   | 0   |     |       |      |      |     |       |       |      |     |      |       |      |
| Diamonds | 1   |     |       |      |      |     |       |       |      |     |      |       |      |
| Clubs    | 2   |     |       |      |      |     |       |       |      |     |      |       |      |
| Spades   | 3   |     |       |      |      |     |       |       |      |     |      |       |      |

deck[2][12] represents the King of Clubs  
 Clubs      King

This simulated deck of cards may be shuffled as follows. First the array `deck` is initialized to zeros. Then, a `row` (03) and a `column` (012) are each chosen at random. The number 1 is inserted in array element `deck[ row ][ column ]` to indicate that this card is going to be the first one dealt from the shuffled deck. This process continues with the numbers 2, 3, ..., 52 being randomly inserted in the `deck` array to indicate which cards are to be placed second, third, ..., and 52nd in the shuffled deck. As the `deck` array begins to fill with card numbers, it is possible that a card will be selected twice (i.e., `deck[ row ][ column ]` will be nonzero when it is selected). This selection is simply ignored, and other `rows` and `columns` are repeatedly chosen at random until an unselected card is found. Eventually, the numbers 1 through 52 will occupy the 52 slots of the `deck` array. At this point, the deck of cards is fully shuffled.

This shuffling algorithm could execute for an indefinitely long period if cards that have already been shuffled are repeatedly selected at random. This phenomenon is known as **indefinite postponement** (also called **starvation**). In the exercises, we discuss a better shuffling algorithm that eliminates the possibility of indefinite postponement.

### Performance Tip 8.3



Sometimes algorithms that emerge in a "natural" way can contain subtle performance problems such as indefinite postponement. Seek algorithms that avoid indefinite postponement.

To deal the first card, we search the array for the element `deck[ row ][ column ]` that matches 1. This is accomplished with a nested `for` statement that varies `row` from 0 to 3 and `column` from 0 to 12. What card does that slot of the array correspond to? The `suit` array has been preloaded with the four suits, so to get the suit, we print the character string `suit[ row ]`. Similarly, to get the face value of the card, we print the character string `face[ column ]`. We also print the character string " of ". Printing this information in the proper order enables us to print each card in the form "King of Clubs", "Ace of Diamonds" and so on.

---

[Page 434]

Let us proceed with the top-down, stepwise-refinement process. The top is simply

Shuffle and deal 52 cards

Our first refinement yields

Initialize the suit array  
Initialize the face array  
Initialize the deck array  
Shuffle the deck  
Deal 52 cards

"Shuffle the deck" may be expanded as follows:

For each of the 52 cards  
    Place card number in randomly selected unoccupied slot of deck

"Deal 52 cards" may be expanded as follows:

For each of the 52 cards  
    Find card number in deck array and print face and suit of card

Incorporating these expansions yields our complete second refinement:

Initialize the suit array  
Initialize the face array  
Initialize the deck array

For each of the 52 cards

    Place card number in randomly selected unoccupied slot of deck

For each of the 52 cards

    Find card number in deck array and print face and suit of card

"Place card number in randomly selected unoccupied slot of deck" may be expanded as follows:

Choose slot of deck randomly

While chosen slot of deck has been previously chosen

    Choose slot of deck randomly

Place card number in chosen slot of deck

"Find card number in deck array and print face and suit of card" may be expanded as follows:

For each slot of the deck array

    If slot contains card number

        Print the face and suit of the card

Incorporating these expansions yields our third refinement (Fig. 8.24):

**Figure 8.24. Pseudocode algorithm for card shuffling and dealing program.**

(This item is displayed on page 435 in the print version)

- 1 Initialize the suit array
- 2 Initialize the face array
- 3 Initialize the deck array
- 4
- 5 For each of the 52 cards
- 6     Choose slot of deck randomly
- 7
- 8     While slot of deck has been previously chosen
- 9         Choose slot of deck randomly
- 10
- 11         Place card number in chosen slot of deck
- 12
- 13 For each of the 52 cards

```

14 For each slot of deck array
15 If slot contains desired card number
16 Print the face and suit of the card

```

This completes the refinement process. Figures 8.258.27 contain the card shuffling and dealing program and a sample execution. Lines 6167 of function `deal` (Fig. 8.26) implement lines 12 of Fig. 8.24. The constructor (lines 2235 of Fig. 8.26) implements lines 13 of Fig. 8.24. Function `shuffle` (lines 3855 of Fig. 8.26) implements lines 511 of Fig. 8.24. Function `deal` (lines 5888 of Fig. 8.26) implements lines 1316 of Fig. 8.24. Note the output formatting used in function `deal` (lines 8183 of Fig. 8.26).

[Page 435]

**Figure 8.25. DeckOfCards header file.**

```

1 // Fig. 8.25: DeckOfCards.h
2 // Definition of class DeckOfCards that
3 // represents a deck of playing cards.
4
5 // DeckOfCards class definition
6 class DeckOfCards
7 {
8 public:
9 DeckOfCards(); // constructor initializes deck
10 void shuffle(); // shuffles cards in deck
11 void deal(); // deals cards in deck
12 private:
13 int deck[4][13]; // represents deck of cards
14 } // end class DeckOfCards

```

**Figure 8.26. Definitions of member functions for shuffling and dealing.**

(This item is displayed on pages 435 - 437 in the print version)

```

1 // Fig. 8.26: DeckOfCards.cpp
2 // Member-function definitions for class DeckOfCards that simulates
3 // the shuffling and dealing of a deck of playing cards.
4 #include <iostream>
5 using std::cout;
6 using std::left;
7 using std::right;
8
9 #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // prototypes for rand and srand
13 using std::rand;
14 using std::srand;
15
16 #include <ctime> // prototype for time
17 using std::time;
18
19 #include "DeckOfCards.h" // DeckOfCards class definition
20
21 // DeckOfCards default constructor initializes deck
22 DeckOfCards::DeckOfCards()
23 {
24 // loop through rows of deck
25 for (int row = 0; row <= 3; row++)
26 {
27 // loop through columns of deck for current row
28 for (int column = 0; column <= 12; column++)
29 {
30 deck[row][column] = 0; // initialize slot of deck to 0
31 } // end inner for
32 } // end outer for
33
34 srand(time(0)); // seed random number generator
35 } // end DeckOfCards default constructor
36
37 // shuffle cards in deck
38 void DeckOfCards::shuffle()
39 {
40 int row; // represents suit value of card
41 int column; // represents face value of card
42
43 // for each of the 52 cards, choose a slot of the deck randomly
44 for (int card = 1; card <= 52; card++)
45 {
46 do // choose a new random location until unoccupied slot is found
47 {
48 row = rand() % 4; // randomly select the row

```

```

49 column = rand() % 13; // randomly select the column
50 } while(deck[row][column] != 0); // end do...while
51
52 // place card number in chosen slot of deck
53 deck[row][column] = card;
54 } // end for
55 } // end function shuffle
56
57 // deal cards in deck
58 void DeckOfCards::deal()
59 {
60 // initialize suit array
61 static const char *suit[4] =
62 { "Hearts", "Diamonds", "Clubs", "Spades" };
63
64 // initialize face array
65 static const char *face[13] =
66 { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
67 "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
68
69 // for each of the 52 cards
70 for (int card = 1; card <= 52; card++)
71 {
72 // loop through rows of deck
73 for (int row = 0; row <= 3; row++)
74 {
75 // loop through columns of deck for current row
76 for (int column = 0; column <= 12; column++)
77 {
78 // if slot contains current card, display card
79 if (deck[row][column] == card)
80 {
81 cout << setw(5) << right << face[column]
82 << " of " << setw(8) << left << suit[row]
83 << (card % 2 == 0 ? '\n' : '\t');
84 } // end if
85 } // end innermost for
86 } // end inner for
87 } // end outer for
88 } // end function deal

```

**Figure 8.27. Card shuffling and dealing program.**

(This item is displayed on pages 437 - 438 in the print version)

```
1 // Fig. 8.27: fig08_27.cpp
2 // Card shuffling and dealing program.
3 #include "DeckOfCards.h" // DeckOfCards class definition
4
5 int main()
6 {
7 DeckOfCards deckOfCards; // create DeckOfCards object
8
9 deckOfCards.shuffle(); // shuffle the cards in the deck
10 deckOfCards.deal(); // deal the cards in the deck
11 return 0; // indicates successful termination
12 } // end main
```

|                   |                   |
|-------------------|-------------------|
| Nine of Spades    | Seven of Clubs    |
| Five of Spades    | Eight of Clubs    |
| Queen of Diamonds | Three of Hearts   |
| Jack of Spades    | Five of Diamonds  |
| Jack of Diamonds  | Three of Diamonds |
| Three of Clubs    | Six of Clubs      |
| Ten of Clubs      | Nine of Diamonds  |
| Ace of Hearts     | Queen of Hearts   |
| Seven of Spades   | Deuce of Spades   |
| Six of Hearts     | Deuce of Clubs    |
| Ace of Clubs      | Deuce of Diamonds |
| Nine of Hearts    | Seven of Diamonds |
| Six of Spades     | Eight of Diamonds |
| Ten of Spades     | King of Hearts    |
| Four of Clubs     | Ace of Spades     |
| Ten of Hearts     | Four of Spades    |
| Eight of Hearts   | Eight of Spades   |
| Jack of Hearts    | Ten of Diamonds   |
| Four of Diamonds  | King of Diamonds  |
| Seven of Hearts   | King of Spades    |
| Queen of Spades   | Four of Hearts    |
| Nine of Clubs     | Six of Diamonds   |
| Deuce of Hearts   | Jack of Clubs     |
| King of Clubs     | Three of Spades   |
| Queen of Clubs    | Five of Clubs     |
| Five of Hearts    | Ace of Diamonds   |

The output statement outputs the face right justified in a field of five characters and outputs the suit left justified in a field of eight characters (Fig. 8.27). The output is printed in two-column format if the card being output is in the first column, a tab is output after the card to move to the second column (line 83); otherwise, a newline is output.

There is also a weakness in the dealing algorithm. Once a match is found, even if it is found on the first try, the two inner `for` statements continue searching the remaining elements of `deck` for a match. In the exercises, we correct this deficiency.

 PREVNEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 439]

This parameter specifies a pointer to a function. The keyword `bool` indicates that the function being pointed to returns a `bool` value. The text `( *compare )` indicates the name of the pointer to the function (the `*` indicates that parameter `compare` is a pointer). The text `( int, int )` indicates that the function pointed to by `compare` takes two integer arguments. Parentheses are needed around `*compare` to indicate that `compare` is a pointer to a function. If we had not included the parentheses, the declaration would have been

```
bool *compare(int, int)
```

which declares a function that receives two integers as parameters and returns a pointer to a `bool` value.

[Page 441]

The corresponding parameter in the function prototype of `selectionSort` is

```
bool (*)(int, int)
```

Note that only types have been included. As always, for documentation purposes, the programmer can include names that the compiler will ignore.

The function passed to `selectionSort` is called in line 71 as follows:

```
(*compare)(work[smallestOrLargest], work[index])
```

Just as a pointer to a variable is dereferenced to access the value of the variable, a pointer to a function is dereferenced to execute the function. The parentheses around `*compare` are again necessary if they were left out, the `*` operator would attempt to dereference the value returned from the function call. The call to the function could have been made without dereferencing the pointer, as in

```
compare(work[smallestOrLargest], work[index])
```

which uses the pointer directly as the function name. We prefer the first method of calling a function through a pointer, because it explicitly illustrates that `compare` is a pointer to a function that is dereferenced to call the function. The second method of calling a function through a pointer makes it

appear as though `compare` is the name of an actual function in the program. This may be confusing to a user of the program who would like to see the definition of function `compare` and finds that it is not defined in the file.

---

[Page 442]

## Arrays of Pointers to Functions

One use of function pointers is in menu-driven systems. For example, a program might prompt a user to select an option from a menu by entering an integer values. The user's choice can be used as a subscript into an array of function pointers, and the pointer in the array can be used to call the function.

**Figure 8.29** provides a mechanical example that demonstrates declaring and using an array of pointers to functions. The program defines three functions `function0`, `function1` and `function2` that each take an integer argument and do not return a value. Line 17 stores pointers to these three functions in array `f`. In this case, all the functions to which the array points must have the same return type and same parameter types. The declaration in line 17 is read beginning in the leftmost set of parentheses as, "`f` is an array of three pointers to functions that each take an `int` as an argument and return `void`." The array is initialized with the names of the three functions (which, again, are pointers). The program prompts the user to enter a number between 0 and 2, or 3 to terminate. When the user enters a value between 0 and 2, the value is used as the subscript into the array of pointers to functions. Line 29 invokes one of the functions in array `f`. In the call, `f[ choice ]` selects the pointer at location `choice` in the array. The pointer is dereferenced to call the function, and `choice` is passed as the argument to the function. Each function prints its argument's value and its function name to indicate that the function is called correctly. In the exercises, you will develop a menu-driven system. We will see in [Chapter 13, Object-Oriented Programming: Polymorphism](#), that arrays of pointers to functions are used by compiler developers to implement the mechanisms that support `virtual` functions—the key technology behind polymorphism.

---

[Page 443]

### Figure 8.29. Array of pointers to functions.

(This item is displayed on pages 442 - 443 in the print version)

```

1 // Fig. 8.29: fig08_29.cpp
2 // Demonstrating an array of pointers to functions.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // function prototypes -- each function performs similar actions
9 void function0(int);
10 void function1(int);
11 void function2(int);
12
13 int main()
14 {
15 // initialize array of 3 pointers to functions that each
16 // take an int argument and return void
17 void (*f[3])(int) = { function0, function1, function2 };
18
19 int choice;
20
21 cout << "Enter a number between 0 and 2, 3 to end: ";
22 cin >> choice;
23
24 // process user's choice
25 while ((choice >= 0) && (choice < 3))
26 {
27 // invoke the function at location choice in
28 // the array f and pass choice as an argument
29 (*f[choice])(choice);
30
31 cout << "Enter a number between 0 and 2, 3 to end: ";
32 cin >> choice;
33 } // end while
34
35 cout << "Program execution completed." << endl;
36 return 0; // indicates successful termination
37 } // end main
38
39 void function0(int a)
40 {
41 cout << "You entered " << a << " so function0 was called\n\n";
42 } // end function function0
43
44 void function1(int b)
45 {
46 cout << "You entered " << b << " so function1 was called\n\n";
47 } // end function function1
48

```

```
49 void function2(int c)
50 {
51 cout << "You entered " << c << " so function2 was called\n\n";
52 } // end function function2
```

Enter a number between 0 and 2, 3 to end: 0  
You entered 0 so function0 was called

Enter a number between 0 and 2, 3 to end: 1  
You entered 1 so function1 was called

Enter a number between 0 and 2, 3 to end: 2  
You entered 2 so function2 was called

Enter a number between 0 and 2, 3 to end: 3  
Program execution completed.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 444]

### 8.13.1. Fundamentals of Characters and Pointer-Based Strings

Characters are the fundamental building blocks of C++ source programs. Every program is composed of a sequence of characters that when grouped together meaningfully is interpreted by the compiler as a series of instructions used to accomplish a task. A program may contain **character constants**. A character constant is an integer value represented as a character in single quotes. The value of a character constant is the integer value of the character in the machine's character set. For example, 'z' represents the integer value of z (122 in the ASCII character set; see [Appendix B](#)), and '\n' represents the integer value of newline (10 in the ASCII character set).

A string is a series of characters treated as a single unit. A string may include letters, digits and various **special characters** such as +, -, \*, / and \$. **String literals**, or **string constants**, in C++ are written in double quotation marks as follows:

|                          |                      |
|--------------------------|----------------------|
| "John Q. Doe"            | (a name)             |
| "9999 Main Street"       | (a street address)   |
| "Maynard, Massachusetts" | (a city and state)   |
| "(201) 555-1212"         | (a telephone number) |

A pointer-based string in C++ is an array of characters ending in the null character ('\0'), which marks where the string terminates in memory. A string is accessed via a pointer to its first character. The value of a string is the address of its first character. Thus, in C++, it is appropriate to say that a string is a constant pointer in fact, a pointer to the string's first character. In this sense, strings are like arrays, because an array name is also a pointer to its first element.

A string literal may be used as an initializer in the declaration of either a character array or a variable of type `char *`. The declarations

```
char color[] = "blue";
const char *colorPtr = "blue";
```

each initialize a variable to the string "blue". The first declaration creates a five-element array `color` containing the characters 'b', 'l', 'u', 'e' and '\0'. The second declaration creates pointer variable `colorPtr` that points to the letter b in the string "blue" (which ends in '\0') somewhere in memory. String literals have `static` storage class (they exist for the duration of the program) and may or may not

be shared if the same string literal is referenced from multiple locations in a program. Also, string literals in C++ are constant their characters cannot be modified.

The declaration `char color[ ] = "blue";` could also be written

```
char color[] = { 'b', 'l', 'u', 'e', '\0' };
```

When declaring a character array to contain a string, the array must be large enough to store the string and its terminating null character. The preceding declaration determines the size of the array, based on the number of initializers provided in the initializer list.

[Page 445]

### Common Programming Error 8.15



Not allocating sufficient space in a character array to store the null character that terminates a string is an error.

### Common Programming Error 8.16



Creating or using a C-style string that does not contain a terminating null character can lead to logic errors.

### Error-Prevention Tip 8.4



When storing a string of characters in a character array, be sure that the array is large enough to hold the largest string that will be stored. C++ allows strings of any length to be stored. If a string is longer than the character array in which it is to be stored, characters beyond the end of the array will overwrite data in memory following the array, leading to logic errors.

A string can be read into a character array using stream extraction with `cin`. For example, the following statement can be used to read a string into character array `word[ 20 ]`:

```
cin >> word;
```

The string entered by the user is stored in `word`. The preceding statement reads characters until a white-space character or end-of-file indicator is encountered. Note that the string should be no longer than 19 characters to leave room for the terminating null character. The `setw` stream manipulator can be used to ensure that the string read into `word` does not exceed the size of the array. For example, the statement

```
cin >> setw(20) >> word;
```

specifies that `cin` should read a maximum of 19 characters into array `word` and save the 20th location in the array to store the terminating null character for the string. The `setw` stream manipulator applies only to the next value being input. If more than 19 characters are entered, the remaining characters are not saved in `word`, but will be read in and can be stored in another variable.

In some cases, it is desirable to input an entire line of text into an array. For this purpose, C++ provides the function `cin.getline` in header file `<iostream>`. In [Chapter 3](#) you were introduced to the similar function `getline` from header file `<string>`, which read input until a newline character was entered, and stored the input (without the newline character) into a `string` specified as an argument. The `cin.getline` function takes three arguments a character array in which the line of text will be stored, a length and a delimiter character. For example, the program segment

```
char sentence[80];
cin.getline(sentence, 80, '\n');
```

declares array `sentence` of 80 characters and reads a line of text from the keyboard into the array. The function stops reading characters when the delimiter character '`'\n'`' is encountered, when the end-of-file indicator is entered or when the number of characters read so far is one less than the length specified in the second argument. (The last character in the array is reserved for the terminating null character.) If the delimiter character is encountered, it is read and discarded. The third argument to `cin.getline` has '`'\n'`' as a default value, so the preceding function call could have been written as follows:

```
cin.getline(sentence, 80);
```

[Page 446]

[Chapter 15](#), Stream Input/Output, provides a detailed discussion of `cin.getline` and other input/output functions.

Common Programming Error 8.17



Processing a single character as a `char *` string can lead to a fatal runtime error. A `char *` string is a pointer probably a respectable large integer. However, a character is a small integer (ASCII values range 0–255). On many systems, dereferencing a `char` value causes an error, because low memory addresses are reserved for special purposes such as operating system interrupt handlers so "memory access violations" occur.

## Common Programming Error 8.18



Passing a string as an argument to a function when a character is expected is a compilation error.

### 8.13.2. String Manipulation Functions of the String-Handling Library

The string-handling library provides many useful functions for manipulating string data, comparing strings, searching strings for characters and other strings, tokenizing strings (separating strings into logical pieces such as the separate words in a sentence) and determining the length of strings. This section presents some common string-manipulation functions of the string-handling library (from the C++ standard library). The functions are summarized in Fig. 8.30; then each is used in a live-code example. The prototypes for these functions are located in header file `<cstring>`.

**Figure 8.30. String-manipulation functions of the string-handling library.**

---

[Page 447]

| Function prototype                                                | Function description                                                                                                                                        |
|-------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>char *strcpy( char *s1, const char *s2 );</code>            | Copies the string <code>s2</code> into the character array <code>s1</code> . The value of <code>s1</code> is returned.                                      |
| <code>char *strncpy( char *s1, const char *s2, size_t n );</code> | Copies at most <code>n</code> characters of the string <code>s2</code> into the character array <code>s1</code> . The value of <code>s1</code> is returned. |
| <code>char *strcat( char *s1, const char *s2 );</code>            |                                                                                                                                                             |

|                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                       | Appends the string <code>s2</code> to <code>s1</code> . The first character of <code>s2</code> overwrites the terminating null character of <code>s1</code> . The value of <code>s1</code> is returned.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>char *strncat( char *s1, const char *s2, size_t n );</code>     | Appends at most <code>n</code> characters of string <code>s2</code> to string <code>s1</code> . The first character of <code>s2</code> overwrites the terminating null character of <code>s1</code> . The value of <code>s1</code> is returned.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>int strcmp( const char *s1, const char *s2 );</code>            | Compares the string <code>s1</code> with the string <code>s2</code> . The function returns a value of zero, less than zero (usually -1) or greater than zero (usually 1) if <code>s1</code> is equal to, less than or greater than <code>s2</code> , respectively.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>int strncmp( const char *s1, const char *s2, size_t n );</code> | Compares up to <code>n</code> characters of the string <code>s1</code> with the string <code>s2</code> . The function returns zero, less than zero or greater than zero if the <code>n</code> -character portion of <code>s1</code> is equal to, less than or greater than the corresponding <code>n</code> -character portion of <code>s2</code> , respectively.                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <code>char *strtok( char *s1, const char *s2 );</code>                | A sequence of calls to <code>strtok</code> breaks string <code>s1</code> into "tokens" logical pieces such as words in a line of text. The string is broken up based on the characters contained in string <code>s2</code> . For instance, if we were to break the string "this:is:a:string" into tokens based on the character ':', the resulting tokens would be "this", "is", "a" and "string". Function <code>strtok</code> returns only one token at a time, however. The first call contains <code>s1</code> as the first argument, and subsequent calls to continue tokenizing the same string contain <code>NULL</code> as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, <code>NULL</code> is returned. |
| <code>size_t strlen( const char *s );</code>                          | Determines the length of string <code>s</code> . The number of characters preceding the terminating null character is returned.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

[Page 447]

Note that several functions in Fig. 8.30 contain parameters with data type `size_t`. This type is defined in the header file `<cstring>` to be an unsigned integral type such as `unsigned int` or `unsigned long`.

Common Programming Error 8.19



Forgetting to include the `<cstring>` header file when using functions from the string-handling library causes compilation errors.

## Copying Strings with `strcpy` and `strncpy`

Function `strcpy` copies its second argument a string into its first argument a character array that must be large enough to store the string and its terminating null character, (which is also copied). Function `strncpy` is equivalent to `strcpy`, except that `strncpy` specifies the number of characters to be copied from the string into the array. Note that function `strncpy` does not necessarily copy the terminating null character of its second argument a terminating null character is written only if the number of characters to be copied is at least one more than the length of the string. For example, if "test" is the second argument, a terminating null character is written only if the third argument to `strncpy` is at least 5 (four characters in "test" plus one terminating null character). If the third argument is larger than 5, null characters are appended to the array until the total number of characters specified by the third argument is written.

[Page 448]

### Common Programming Error 8.20



When using `strncpy`, the terminating null character of the second argument (`char * string`) will not be copied if the number of characters specified by `strncpy`'s third argument is not greater than the second argument's length. In that case, a fatal error may occur if the programmer does not manually terminate the resulting `char * string` with a null character.

Figure 8.31 uses `strcpy` (line 17) to copy the entire string in array `x` into array `y` and uses `strncpy` (line 23) to copy the first 14 characters of array `x` into array `z`. Line 24 appends a null character ('`\0`') to array `z`, because the call to `strncpy` in the program does not write a terminating null character. (The third argument is less than the string length of the second argument plus one.)

**Figure 8.31. `strcpy` and `strncpy`.**

```

1 // Fig. 8.31: fig08_31.cpp
2 // Using strcpy and strncpy.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototypes for strcpy and strncpy
8 using std::strcpy;
9 using std::strncpy;
10
11 int main()
12 {
13 char x[] = "Happy Birthday to You"; // string length 21
14 char y[25];
15 char z[15];
16
17 strcpy(y, x); // copy contents of x into y
18
19 cout << "The string in array x is: " << x
20 << "\nThe string in array y is: " << y << '\n';
21
22 // copy first 14 characters of x into z
23 strncpy(z, x, 14); // does not copy null character
24 z[14] = '\0'; // append '\0' to z's contents
25
26 cout << "The string in array z is: " << z << endl;
27 return 0; // indicates successful termination
28 } // end main

```

The string in array x is: Happy Birthday to You  
 The string in array y is: Happy Birthday to You  
 The string in array z is: Happy Birthday

## Concatenating Strings with **strcat** and **strncat**

Function **strcat** appends its second argument (a string) to its first argument (a character array containing a string). The first character of the second argument replaces the null character ('\0') that terminates the string in the first argument. The programmer must ensure that the array used to store the first string is large enough to store the combination of the first string, the second string and the terminating null character (copied from the second string). Function **strncat** appends a specified number of characters

from the second string to the first string and appends a terminating null character to the result. The program of Fig. 8.32 demonstrates function `strcat` (lines 19 and 29) and function `strncat` (line 24).

[Page 449]

**Figure 8.32. `strcat` and `strncat`.**

```
1 // Fig. 8.32: fig08_32.cpp
2 // Using strcat and strncat.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototypes for strcat and strncat
8 using std::strcat;
9 using std::strncat;
10
11 int main()
12 {
13 char s1[20] = "Happy "; // length 6
14 char s2[] = "New Year "; // length 9
15 char s3[40] = "";
16
17 cout << "s1 = " << s1 << "\ns2 = " << s2;
18
19 strcat(s1, s2); // concatenate s2 to s1 (length 15)
20
21 cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1 << "\ns2 = " << s2;
22
23 // concatenate first 6 characters of s1 to s3
24 strncat(s3, s1, 6); // places '\0' after last character
25
26 cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
27 << "\ns3 = " << s3;
28
29 strcat(s3, s1); // concatenate s1 to s3
30 cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
31 << "\ns3 = " << s3 << endl;
32 return 0; // indicates successful termination
33 } // end main
```

```
s1 = Happy
s2 = New Year

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy

After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
```

[Page 450]

## Comparing Strings with `strcmp` and `strncmp`

Figure 8.33 compares three strings using `strcmp` (lines 21, 22 and 23) and `strncmp` (lines 26, 27 and 28). Function `strcmp` compares its first string argument with its second string argument character by character. The function returns zero if the strings are equal, a negative value if the first string is less than the second string and a positive value if the first string is greater than the second string. Function `strncmp` is equivalent to `strcmp`, except that `strncmp` compares up to a specified number of characters. Function `strncmp` stops comparing characters if it reaches the null character in one of its string arguments. The program prints the integer value returned by each function call.

**Figure 8.33. `strcmp` and `strncmp`.**

(This item is displayed on pages 450 - 451 in the print version)

```

1 // Fig. 8.33: fig08_33.cpp
2 // Using strcmp and strncmp.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstring> // prototypes for strcmp and strncmp
11 using std::strcmp;
12 using std::strncmp;
13
14 int main()
15 {
16 char *s1 = "Happy New Year";
17 char *s2 = "Happy New Year";
18 char *s3 = "Happy Holidays";
19
20 cout << "s1 = " << s1 << "\ns2 = " << s2 << "\ns3 = " << s3
21 << "\n\nstrcmp(s1, s2) = " << setw(2) << strcmp(s1, s2)
22 << "\nstrcmp(s1, s3) = " << setw(2) << strcmp(s1, s3)
23 << "\nstrcmp(s3, s1) = " << setw(2) << strcmp(s3, s1);
24
25 cout << "\n\nstrncmp(s1, s3, 6) = " << setw(2)
26 << strncmp(s1, s3, 6) << "\nstrncmp(s1, s3, 7) = " << setw(2)
27 << strncmp(s1, s3, 7) << "\nstrncmp(s3, s1, 7) = " << setw(2)
28 << strncmp(s3, s1, 7) << endl;
29 return 0; // indicates successful termination
30 } // end main

```

```

s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) = 0
strcmp(s1, s3) = 1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) = 0
strncmp(s1, s3, 7) = 1
strncmp(s3, s1, 7) = -1

```

## Common Programming Error 8.21



Assuming that `strcmp` and `strncmp` return one (a true value) when their arguments are equal is a logic error. Both functions return zero (C++'s false value) for equality. Therefore, when testing two strings for equality, the result of the `strcmp` or `strncmp` function should be compared with zero to determine whether the strings are equal.

To understand just what it means for one string to be "greater than" or "less than" another string, consider the process of alphabetizing a series of last names. The reader would, no doubt, place "Jones" before "Smith," because the first letter of "Jones" comes before the first letter of "Smith" in the alphabet. But the alphabet is more than just a list of 26 letters; it is an ordered list of characters. Each letter occurs in a specific position within the list. "Z" is more than just a letter of the alphabet; "Z" is specifically the 26th letter of the alphabet.

---

[Page 451]

How does the computer know that one letter comes before another? All characters are represented inside the computer as numeric codes; when the computer compares two strings, it actually compares the numeric codes of the characters in the strings.

In an effort to standardize character representations, most computer manufacturers have designed their machines to utilize one of two popular coding schemes **ASCII** or **EBCDIC**. Recall that ASCII stands for "American Standard Code for Information Interchange." EBCDIC stands for "Extended Binary Coded Decimal Interchange Code." There are other coding schemes, but these two are the most popular.

ASCII and EBCDIC are called **character codes**, or character sets. Most readers of this book will be using desktop or notebook computers that use the ASCII character set. IBM mainframe computers use the EBCDIC character set. As Internet and World Wide Web usage becomes pervasive, the newer Unicode character set is growing rapidly in popularity. For more information on Unicode, visit [www.unicode.org](http://www.unicode.org). String and character manipulations actually involve the manipulation of the appropriate numeric codes and not the characters themselves. This explains the interchangeability of characters and small integers in C++. Since it is meaningful to say that one numeric code is greater than, less than or equal to another numeric code, it becomes possible to relate various characters or strings to one another by referring to the character codes. Appendix B contains the ASCII character codes.

### Portability Tip 8.5



The internal numeric codes used to represent characters may be different on different computers, because these computers may use different character sets.

## Portability Tip 8.6



Do not explicitly test for ASCII codes, as in `if ( rating == 65 )`; rather, use the corresponding character constant, as in `if ( rating == 'A' )`.

[Note: With some compilers, functions `strcmp` and `strncmp` always return -1, 0 or 1, as in the sample output of Fig. 8.33. With other compilers, these functions return 0 or the difference between the numeric codes of the first characters that differ in the strings being compared. For example, when `s1` and `s3` are compared, the first characters that differ between them are the first character of the second word in each string (numeric code 78) in `s1` and H (numeric code 72) in `s3`, respectively. In this case, the return value will be 6 (or -6 if `s3` is compared to `s1`).]

[Page 452]

## Tokenizing a String with `strtok`

Function `strtok` breaks a string into a series of **tokens**. A token is a sequence of characters separated by **delimiting characters** (usually spaces or punctuation marks). For example, in a line of text, each word can be considered a token, and the spaces separating the words can be considered delimiters.

Multiple calls to `strtok` are required to break a string into tokens (assuming that the string contains more than one token). The first call to `strtok` contains two arguments, a string to be tokenized and a string containing characters that separate the tokens (i.e., delimiters). Line 19 in Fig. 8.34 assigns to `tokenPtr` a pointer to the first token in `sentence`. The second argument, " ", indicates that tokens in `sentence` are separated by spaces. Function `strtok` searches for the first character in `sentence` that is not a delimiting character (space). This begins the first token. The function then finds the next delimiting character in the string and replaces it with a null ('\0') character. This terminates the current token. Function `strtok` saves (in a static variable) a pointer to the next character following the token in `sentence` and returns a pointer to the current token.

[Page 453]

### Figure 8.34. `strtok`.

(This item is displayed on pages 452 - 453 in the print version)

```

1 // Fig. 8.34: fig08_34.cpp
2 // Using strtok.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototype for strtok
8 using std::strtok;
9
10 int main()
11 {
12 char sentence[] = "This is a sentence with 7 tokens";
13 char *tokenPtr;
14
15 cout << "The string to be tokenized is:\n" << sentence
16 << "\n\nThe tokens are:\n\n";
17
18 // begin tokenization of sentence
19 tokenPtr = strtok(sentence, " ");
20
21 // continue tokenizing sentence until tokenPtr becomes NULL
22 while (tokenPtr != NULL)
23 {
24 cout << tokenPtr << '\n';
25 tokenPtr = strtok(NULL, " "); // get next token
26 } // end while
27
28 cout << "\nAfter strtok, sentence = " << sentence << endl;
29 return 0; // indicates successful termination
30 } // end main

```

The string to be tokenized is:  
 This is a sentence with 7 tokens

The tokens are:

This  
 is  
 a  
 sentence  
 with  
 7  
 tokens

After strtok, sentence = This

Subsequent calls to `strtok` to continue tokenizing `sentence` contain `NULL` as the first argument (line 25). The `NULL` argument indicates that the call to `strtok` should continue tokenizing from the location in `sentence` saved by the last call to `strtok`. Note that `strtok` maintains this saved information in a manner that is not visible to the programmer. If no tokens remain when `strtok` is called, `strtok` returns `NULL`. The program of Fig. 8.34 uses `strtok` to tokenize the string "This is a sentence with 7 tokens". The program prints each token on a separate line. Line 28 outputs `sentence` after tokenization. Note that `strtok` modifies the input string; therefore, a copy of the string should be made if the program requires the original after the calls to `strtok`. When `sentence` is output after tokenization, note that only the word "This" prints, because `strtok` replaced each blank in `sentence` with a null character ('`\0`') during the tokenization process.

### Common Programming Error 8.22



Not realizing that `strtok` modifies the string being tokenized and then attempting to use that string as if it were the original unmodified string is a logic error.

## Determining String Lengths

Function `strlen` takes a string as an argument and returns the number of characters in the string—the terminating null character is not included in the length. The length is also the index of the null character. The program of Fig. 8.35 demonstrates function `strlen`.

**Figure 8.35. `strlen` returns the length of a `char * string`.**

(This item is displayed on pages 453 - 454 in the print version)

```
1 // Fig. 8.35: fig08_35.cpp
2 // Using strlen.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototype for strlen
8 using std::strlen;
9
10 int main()
11 {
12 char *string1 = "abcdefghijklmnopqrstuvwxyz";
13 char *string2 = "four";
14 char *string3 = "Boston";
15
16 cout << "The length of \" " << string1 << "\" is " << strlen(string1)
17 << "\nThe length of \" " << string2 << "\" is " << strlen(string2)
18 << "\nThe length of \" " << string3 << "\" is " << strlen(string3)
19 << endl;
20
21 return 0; // indicates successful termination
22 } // end main
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 454]

## 8.14. Wrap-Up

In this chapter we provided a detailed introduction into pointers, or variables that contain memory addresses as their values. We began by demonstrating how to declare and initialize pointers. You saw how to use the address operator (`&`) to assign the address of a variable to a pointer and the indirection operator (`*`) to access the data stored in the variable indirectly referenced by a pointer. We discussed passing arguments by reference using both pointer arguments and reference arguments.

You learned how to use `const` with pointers to enforce the principle of least privilege. We demonstrated using nonconstant pointers to nonconstant data, nonconstant pointers to constant data, constant pointers to nonconstant data and constant pointers to constant data. We then used selection sort to demonstrate passing arrays and individual array elements by reference. We discussed the `sizeof` operator, which can be used to determine the size of a data type in bytes during program compilation.

We continued by demonstrating how to use pointers in arithmetic and comparison expressions. You saw that pointer arithmetic can be used to jump from one element of an array to another. You learned how to use arrays of pointers, and more specifically string arrays (an array of strings). We then continued by discussing function pointers, which enable programmers to pass functions as parameters. We concluded the chapter with a discussion of several C++ functions that manipulate pointer-based strings. You learned string processing capabilities such as copying strings, tokenizing strings and determining the length of strings.

In the next chapter, we begin our in-depth treatment of classes. You will learn about the scope of a class's members, and how to keep objects in a consistent state. You will also learn about using special member functions called constructors and destructors, which execute when an object is created and destroyed, respectively.

[◀ PREV](#)[NEXT ▶](#)**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 456]

- Pointers can be compared using the equality and relational operators. Comparisons using relational operators are meaningful only if the pointers point to members of the same array.
- Pointers that point to arrays can be subscripted exactly as array names can.
- In pointer/offset notation, if the pointer points to the first element of the array, the offset is the same as an array subscript.
- All subscripted array expressions can be written with a pointer and an offset, using either the name of the array as a pointer or using a separate pointer that points to the array.
- Arrays may contain pointers.
- A pointer to a function is the address where the code for the function resides.
- Pointers to functions can be passed to functions, returned from functions, stored in arrays and assigned to other pointers.
- A common use of function pointers is in so-called menu-driven systems. The function pointers are used to select which function to call for a particular menu item.
- Function `strcpy` copies its second argument a string into its first argument a character array. The programmer must ensure that the target array is large enough to store the string and its terminating null character.
- Function `strncpy` is equivalent to `strcpy`, except that a call to `strncpy` specifies the number of characters to be copied from the string into the array. The terminating null character will be copied only if the number of characters to be copied is at least one more than the length of the string.
- Function `strcat` appends its second string argument including the terminating null character to its first string argument. The first character of the second string replaces the null ('\0') character of the first string. The programmer must ensure that the target array used to store the first string is large enough to store both the first string and the second string.
- Function `strncat` is equivalent to `strcat`, except that a call to `strncat` appends a specified number of characters from the second string to the first string. A terminating null character is appended to the result.
- Function `strcmp` compares its first string argument with its second string argument character by character. The function returns zero if the strings are equal, a negative value if the first string is less than the second string and a positive value if the first string is greater than the second string.
- Function `strncmp` is equivalent to `strcmp`, except that `strncmp` compares a specified number of characters. If the number of characters in one of the strings is less than the number of characters specified, `strncmp` compares characters until the null character in the shorter string is encountered.
- A sequence of calls to `strtok` breaks a string into tokens that are separated by characters contained in a second string argument. The first call specifies the string to be tokenized as the first argument, and subsequent calls to continue tokenizing the same string specify `NULL` as the first argument. The function returns a pointer to the current token from each call. If there are no more tokens when `strtok` is called, `NULL` is returned.
- Function `strlen` takes a string as an argument and returns the number of characters in the string the terminating null character is not included in the length of the string.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 457]

character code

character constant

command-line arguments

comparing strings

concatenating strings

const with function parameters

constant pointer

constant pointer to constant data

constant pointer to nonconstant data

copying strings

decrement a pointer

delimiter character

dereference a 0 pointer

dereference a pointer

dereferencing operator (\*)

directly reference a value

EBCDIC (Extended Binary Coded Decimal Interchange Code)

function pointer

getline function of cin

increment a pointer

indefinite postponement

indirection

indirection (\*) operator

indirectly reference a value

interchangeability of arrays and pointers

islower function (<cctype>)

modify a constant pointer

modify address stored in pointer variable

nonconstant pointer to constant data

nonconstant pointer to nonconstant data

null character ('\0')

null pointer

null-terminated string

offset to a pointer

pass-by-reference with pointer arguments

pass-by-reference with reference arguments

pointer arithmetic

pointer-based strings

pointer dereference (\*) operator

pointer subtraction

pointer to a function

reference to constant data

referencing array elements

selection sort algorithm

`size_t` type

`sizeof` operator

special characters

starvation

`strcat` function of header file `<cstring>`

`strcmp` function of header file `<cstring>`

`strcpy` function of header file `<cstring>`

string array

string being tokenized

string constant

string copying

`strlen` function of header file `<cstring>`

`strncat` function of header file `<cstring>`

[strcmp function of header file <cstring>](#)

[strncpy function of header file <cstring>](#)

[strtok function of header file <cstring>](#)

[terminating null character](#)

[token](#)

[tokenizing strings](#)

[toupper function \(<cctype>\)](#)

[!\[\]\(28ede2d64848c82143e764129d9bf1e6\_img.jpg\) PREV](#)

[\*\*NEXT\*\* !\[\]\(90c351dd01807fbffc396df094560450\_img.jpg\)](#)

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 458]

•

Declare an array of type `double` called `numbers` with 10 elements, and initialize the elements to the values 0.0, 1.1, 2.2, ..., 9.9. Assume that the symbolic constant `SIZE` has been defined as 10.

•

Declare a pointer `nPtr` that points to a variable of type `double`.

•

Use a `for` statement to print the elements of array `numbers` using array subscript notation. Print each number with one position of precision to the right of the decimal point.

•

Write two separate statements that each assign the starting address of array `numbers` to the pointer variable `nPtr`.

•

Use a `for` statement to print the elements of array `numbers` using pointer/offset notation with pointer `nPtr`.

•

Use a `for` statement to print the elements of array `numbers` using pointer/offset notation with the array name as the pointer.

•

Use a `for` statement to print the elements of array `numbers` using pointer/subscript notation with pointer `nPtr`.

•

Refer to the fourth element of array `numbers` using array subscript notation, pointer/offset notation with the array name as the pointer, pointer subscript notation with `nPtr` and pointer/offset notation with `nPtr`.

• Assuming that `nPtr` points to the beginning of array `numbers`, what address is referenced by `nPtr + 8`? What value is stored at that location?

•

Assuming that `nPtr` points to `numbers[ 5 ]`, what address is referenced by `nPtr` after `nPtr -= 4` is executed? What is the value stored at that location?

## 8.4

For each of the following, write a single statement that performs the specified task. Assume that floating-point variables `number1` and `number2` have been declared and that `number1` has been initialized to 7.3. Assume that variable `ptr` is of type `char *`. Assume that arrays `s1` and `s2` are each 100-element char arrays that are initialized with string literals.

a.

Declare the variable `fPtr` to be a pointer to an object of type `double`.

b.

Assign the address of variable `number1` to pointer variable `fPtr`.

c.

Print the value of the object pointed to by `fPtr`.

d.

Assign the value of the object pointed to by `fPtr` to variable `number2`.

e.

Print the value of `number2`.

f.

Print the address of `number1`.

g.

Print the address stored in `fPtr`. Is the value printed the same as the address of `number1`?

**h.**

Copy the string stored in array `s2` into array `s1`.

**i.**

Compare the string in `s1` with the string in `s2`, and print the result.

**j.**

Append the first 10 characters from the string in `s2` to the string in `s1`.

**k.**

Determine the length of the string in `s1`, and print the result.

**l.**

Assign to `ptr` the location of the first token in `s2`. The tokens delimiters are commas (,).

## 8.5

Perform the task specified by each of the following statements:

**a.**

Write the function header for a function called `exchange` that takes two pointers to double-precision, floating-point numbers `x` and `y` as parameters and does not return a value.

**b.**

Write the function prototype for the function in part (a).

**c.**

Write the function header for a function called `evaluate` that returns an integer and that takes as parameters integer `x` and a pointer to function `poly`. Function `poly` takes an integer parameter and returns an integer.

Write the function prototype for the function in part (c).

e.

Write two statements that each initialize character array vowel with the string of vowels, "AEIOU".

## 8.6

Find the error in each of the following program segments. Assume the following declarations and statements:

[Page 459]

```
int *zPtr; // zPtr will reference array z
int *aPtr = 0;
void *sPtr = 0;
int number;
int z[5] = { 1, 2, 3, 4, 5 };
```

a.

```
++zPtr;
```

b.

```
// use pointer to get first value of array
number = zPtr;
```

c.

```
// assign array element 2 (the value 3) to number
number = *zPtr[2];
```

d.

```
// print entire array z
for (int i = 0; i <= 5; i++)
 cout << zPtr[i] << endl;
```

e.

```
// assign the value pointed to by sPtr to number
number = *sPtr;
```

**f.**

```
++z;
```

**g.**

```
char s[10];
cout << strncpy(s, "hello", 5) << endl;
```

**h.**

```
char s[12];
strcpy(s, "Welcome Home");
```

**i.**

```
if (strcmp(string1, string2))
 cout << "The strings are equal" << endl;
```

## 8.7

What (if anything) prints when each of the following statements is performed? If the statement contains an error, describe the error and indicate how to correct it. Assume the following variable declarations:

```
char s1[50] = "jack";
char s2[50] = "jill";
char s3[50];
```

**a.**

```
cout << strcpy(s3, s2) << endl;
```

**b.**

```
cout << strcat(strcat(strcpy(s3, s1), " and "), s2)
 << endl;
```

**c.**

```
cout << strlen(s1) + strlen(s2) << endl;
```

d.

```
cout << strlen(s3) << endl;
```

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 460]

•

```
nPtr = numbers;
nPtr = &numbers[0];
```

•

```
cout << fixed << showpoint << setprecision(1);
for (int j = 0; j < SIZE; j++)
 cout << *(nPtr + j) << ' ';
```

•

```
cout << fixed << showpoint << setprecision(1);
for (int k = 0; k < SIZE; k++)
 cout << *(numbers + k) << ' ';
```

•

```
cout << fixed << showpoint << setprecision(1);
for (int m = 0; m < SIZE; m++)
 cout << nPtr[m] << ' ';
```

•

```
numbers[3]
*(numbers + 3)
nPtr[3]
*(nPtr + 3)
```

•

The address is  $1002500 + 8 * 8 = 1002564$ . The value is 8.8.

•

The address of numbers[ 5 ] is  $1002500 + 5 * 8 = 1002540$ .

The address of nPtr -= 4 is  $1002540 - 4 * 8 = 1002508$ .

The value at that location is 1.1.

## 8.4

a.

```
double *fPtr;
```

b.

```
fPtr = &number1;
```

c.

```
cout << "The value of *fPtr is " << *fPtr << endl;
```

d.

```
number2 = *fPtr;
```

e.

```
cout << "The value of number2 is " << number2 << endl;
```

f.

```
cout << "The address of number1 is " << &number1 << endl;
```

g.

```
cout << "The address stored in fPtr is " << fPtr << endl;
```

Yes, the value is the same.

h.

```
strcpy(s1, s2);
```

i.

```
cout << "strcmp(s1, s2) = " << strcmp(s1, s2) << endl;
```

j.

```
strncat(s1, s2, 10);
```

k.

```
cout << "strlen(s1) = " << strlen(s1) << endl;
```

l.

```
ptr = strtok(s2, ", ");
```

## 8.5

a.

```
void exchange(double *x, double *y)
```

b.

```
void exchange(double *, double *);
```

c.

```
int evaluate(int x, int (*poly)(int))
```

d.

```
int evaluate(int, int (*) (int));
```

e.

```
char vowel[] = "AEIOU";
```

```
char vowel[] = { 'A', 'E', 'I', 'O', 'U', '\0' };
```

## 8.6

a.

Error: `zPtr` has not been initialized.

Correction: Initialize `zPtr` with `zPtr = z;`

**b.**

Error: The pointer is not dereferenced.

Correction: Change the statement to `number = *zPtr;`

**c.**

Error: `zPtr[ 2 ]` is not a pointer and should not be dereferenced.

Correction: Change `*zPtr[ 2 ]` to `zPtr[ 2 ].`

**d.**

Error: Referring to an array element outside the array bounds with pointer subscripting.

Correction: To prevent this, change the relational operator in the for statement to `<.`

**e.**

Error: Dereferencing a void pointer.

Correction: To dereference the void pointer, it must first be cast to an integer pointer. Change the statement to `number = *static_cast< int * >( sPtr );`

---

[Page 461]

**f.**

Error: Trying to modify an array name with pointer arithmetic.

Correction: Use a pointer variable instead of the array name to accomplish pointer arithmetic, or subscript the array name to refer to a specific element.

**g.**

Error: Function `strncpy` does not write a terminating null character to array `s`, because its third argument is equal to the length of the string "hello".

Correction: Make 6 the third argument of `strncpy` or assign '\0' to `s[ 5 ]` to ensure that the terminating null character is added to the string.

**h.**

Error: Character array `s` is not large enough to store the terminating null character.

Correction: Declare the array with more elements.

**i.**

Error: Function `strcmp` will return 0 if the strings are equal; therefore, the condition in the `if` statement will be false, and the output statement will not be executed.

Correction: Explicitly compare the result of `strcmp` with 0 in the condition of the `if` statement.

## 8.7

**a.**

jill

**b.**

jack and jill

**c.**

8

**d.**

13

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 462]

•

Print the value of `value2`.

•

Print the address of `value1`.

•

Print the address stored in `longPtr`. Is the value printed the same as `value1`'s address?

## 8.11

Perform the task specified by each of the following statements:

a.

Write the function header for function `zero` that takes a long integer array parameter `bigIntegers` and does not return a value.

b.

Write the function prototype for the function in part (a).

c.

Write the function header for function `add1AndSum` that takes an integer array parameter `oneTooSmall` and returns an integer.

d.

Write the function prototype for the function described in part (c).

Note: [Exercise 8.12](#) through [Exercise 8.15](#) are reasonably challenging. Once you have solved these problems, you ought to be able to implement many popular card games.

## 8.12

Modify the program in [Fig. 8.27](#) so that the card dealing function deals a five-card poker hand. Then write functions to accomplish each of the following:

a.

Determine whether the hand contains a pair.

b.

Determine whether the hand contains two pairs.

c.

Determine whether the hand contains three of a kind (e.g., three jacks).

d.

Determine whether the hand contains four of a kind (e.g., four aces).

e.

Determine whether the hand contains a flush (i.e., all five cards of the same suit).

f.

Determine whether the hand contains a straight (i.e., five cards of consecutive face values).

### 8.13

Use the functions developed in [Exercise 8.12](#) to write a program that deals two five-card poker hands, evaluates each hand and determines which is the better hand.

### 8.14

Modify the program developed in [Exercise 8.13](#) so that it can simulate the dealer. The dealer's five-card hand is dealt "face down" so the player cannot see it. The program should then evaluate the dealer's hand, and, based on the quality of the hand, the dealer should draw one, two or three more cards to replace the corresponding number of unneeded cards in the original hand. The program should then reevaluate the dealer's hand. [Caution: This is a difficult problem!]

### 8.15

Modify the program developed in [Exercise 8.14](#) so that it handles the dealer's hand, but the player is allowed to decide which cards of the player's hand to replace. The program should then evaluate both hands and determine who wins. Now use this new program to play 20 games against the computer. Who wins more games, you or the computer? Have one of your friends play 20 games against the computer. Who wins more games? Based on the results of these games, make appropriate modifications to refine your poker-playing program. [Note: This, too, is a difficult problem.] Play 20 more games. Does your modified program play a better game?

## 8.16

In the card shuffling and dealing program of [Figs. 8.258.27](#), we intentionally used an inefficient shuffling algorithm that introduced the possibility of indefinite postponement. In this problem, you will create a high-performance shuffling algorithm that avoids indefinite postponement.

Modify [Figs. 8.258.27](#) as follows. Initialize the `deck` array as shown in [Fig. 8.36](#). Modify the `shuffle` function to loop row by row and column by column through the array, touching every element once. Each element should be swapped with a randomly selected element of the array. Print the resulting array to determine whether the deck is satisfactorily shuffled (as in [Fig. 8.37](#), for example). You may want your program to call the `shuffle` function several times to ensure a satisfactory shuffle.

**Figure 8.36. Unshuffled deck array.**

(This item is displayed on page 463 in the print version)

| Unshuffled deck array |    |    |    |    |    |    |    |    |    |    |    |    |    |
|-----------------------|----|----|----|----|----|----|----|----|----|----|----|----|----|
|                       | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
| 0                     | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 |
| 1                     | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| 2                     | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 3                     | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 |

**Figure 8.37. Sample shuffled deck array.**

(This item is displayed on page 463 in the print version)

### Sample shuffled deck array

|   | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 19 | 40 | 27 | 25 | 36 | 46 | 10 | 34 | 35 | 41 | 18 | 2  | 44 |
| 1 | 13 | 28 | 14 | 16 | 21 | 30 | 8  | 11 | 31 | 17 | 24 | 7  | 1  |
| 2 | 12 | 33 | 15 | 42 | 43 | 23 | 45 | 3  | 29 | 32 | 4  | 47 | 26 |
| 3 | 50 | 38 | 52 | 39 | 48 | 51 | 9  | 5  | 37 | 49 | 22 | 6  | 20 |

Note that, although the approach in this problem improves the shuffling algorithm, the dealing algorithm still requires searching the deck array for card 1, then card 2, then card 3 and so on. Worse yet, even after the dealing algorithm locates and deals the card, the algorithm continues searching through the remainder of the deck. Modify the program of Figs. 8.258.27 so that once a card is dealt, no further attempts are made to match that card number, and the program immediately proceeds with dealing the next card.

[Page 463]

### 8.17

(Simulation: The Tortoise and the Hare) In this exercise, you will re-create the classic race of the tortoise and the hare. You will use random number generation to develop a simulation of this memorable event.

Our contenders begin the race at "square 1" of 70 squares. Each square represents a possible position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally the contenders lose ground.

There is a clock that ticks once per second. With each tick of the clock, your program should adjust the position of the animals according to the rules in Fig. 8.38.

**Figure 8.38. Rules for moving the tortoise and the hare.**

(This item is displayed on page 464 in the print version)

| Animal | Move type | Percentage of the time | Actual move |
|--------|-----------|------------------------|-------------|
|        |           |                        |             |

|          |            |     |                        |
|----------|------------|-----|------------------------|
| Tortoise | Fast plod  | 50% | 3 squares to the right |
|          | Slip       | 20% | 6 squares to the left  |
|          | Slow plod  | 30% | 1 square to the right  |
| Hare     | Sleep      | 20% | No move at all         |
|          | Big hop    | 20% | 9 squares to the right |
|          | Big slip   | 10% | 12 squares to the left |
|          | Small hop  | 30% | 1 square to the right  |
|          | Small slip | 20% | 2 squares to the left  |

Use variables to keep track of the positions of the animals (i.e., position numbers are 170). Start each animal at position 1 (i.e., the "starting gate"). If an animal slips left before square 1, move the animal back to square 1.

Generate the percentages in the preceding table by producing a random integer  $i$  in the range  $1 \leq i \leq 10$ . For the tortoise, perform a "fast plod" when  $1 \leq i \leq 5$ , a "slip" when  $6 \leq i \leq 7$  or a "slow plod" when  $8 \leq i \leq 10$ . Use a similar technique to move the hare.

Begin the race by printing

BANG ! ! ! !

AND THEY'RE OFF ! ! ! ! !

For each tick of the clock (i.e., each repetition of a loop), print a 70-position line showing the letter  $T$  in the tortoise's position and the letter  $H$  in the hare's position. Occasionally, the contenders land on the same square. In this case, the tortoise bites the hare and your program should print OUCH! ! ! beginning at that position. All print positions other than the  $T$ , the  $H$  or the OUCH! ! ! (in case of a tie) should be blank.

After printing each line, test if either animal has reached or passed square 70. If so, print the winner and terminate the simulation. If the tortoise wins, print TORTOISE WINS ! ! ! YAY ! ! ! If the hare wins, print Hare wins. Yuch. If both animals win on the same clock tick, you may want to favor the tortoise (the "underdog"), or you may want to print It's a tie. If neither animal wins, perform the loop again to simulate the next tick of the clock. When you are ready to run your program, assemble a group of fans to watch the race. You'll be amazed how involved the audience gets!

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 465]

Now let us consider two simple SML programs. The first SML program (Fig. 8.40) reads two numbers from the keyboard and computes and prints their sum. The instruction +1007 reads the first number from the keyboard and places it into location 07 (which has been initialized to zero). Instruction +1008 reads the next number into location 08. The load instruction, +2007, places (copies) the first number into the accumulator, and the add instruction, +3008, adds the second number to the number in the accumulator. All SML arithmetic instructions leave their results in the accumulator. The store instruction, +2109, places (copies) the result back into memory location 09. Then the write instruction, +1109, takes the number and prints it (as a signed four-digit decimal number). The halt instruction, +4300, terminates execution.

[Page 466]

**Figure 8.40. SML Example 1.**

| Location | Number | Instruction  |
|----------|--------|--------------|
| 00       | +1007  | (Read A)     |
| 01       | +1008  | (Read B)     |
| 02       | +2007  | (Load A)     |
| 03       | +3008  | (Add B)      |
| 04       | +2109  | (Store C)    |
| 05       | +1109  | (Write C)    |
| 06       | +4300  | (Halt)       |
| 07       | +0000  | (Variable A) |
| 08       | +0000  | (Variable B) |
| 09       | +0000  | (Result C)   |

The SML program in Fig. 8.41 reads two numbers from the keyboard, then determines and prints the

larger value. Note the use of the instruction +4107 as a conditional transfer of control, much the same as C++'s if statement.

**Figure 8.41. SML Example 2.**

| Location | Number | Instruction             |
|----------|--------|-------------------------|
| 00       | +1009  | (Read A)                |
| 01       | +1010  | (Read B)                |
| 02       | +2009  | (Load A)                |
| 03       | +3110  | (Subtract B)            |
| 04       | +4107  | (Branch negative to 07) |
| 05       | +1109  | (Write A)               |
| 06       | +4300  | (Halt)                  |
| 07       | +1110  | (Write B)               |
| 08       | +4300  | (Halt)                  |
| 09       | +0000  | (Variable A)            |
| 10       | +0000  | (Variable B)            |

Now write SML programs to accomplish each of the following tasks:

**a.**

Use a sentinel-controlled loop to read positive numbers and compute and print their sum. Terminate input when a negative number is entered.

**b.**

Use a counter-controlled loop to read seven numbers, some positive and some negative, and compute and print their average.

**c.**

Read a series of numbers, and determine and print the largest number. The first number read

indicates how many numbers should be processed.

[Page 467]

## 8.19

(Computer Simulator) It may at first seem outrageous, but in this problem you are going to build your own computer. No, you will not be soldering components together. Rather, you will use the powerful technique of software-based simulation to create a software model of the Simpletron. You will not be disappointed. Your Simpletron simulator will turn the computer you are using into a Simpletron, and you actually will be able to run, test and debug the SML programs you wrote in [Exercise 8.18](#).

When you run your Simpletron simulator, it should begin by printing

```
*** Welcome to Simpletron! ***
*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** location number and a question mark (?). ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***
*** your program. ***
```

Your program should simulate the Simpletron's memory with a single-subscripted, 100-element array memory. Now assume that the simulator is running, and let us examine the dialog as we enter the program of Example 2 of [Exercise 8.18](#):

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999

*** Program loading completed ***
*** Program execution begins ***
```

Note that the numbers to the right of each ? in the preceding dialog represent the SML program

instructions input by the user.

The SML program has now been placed (or loaded) into array `memory`. Now the Simpletron executes your SML program. Execution begins with the instruction in location 00 and, like C++, continues sequentially, unless directed to some other part of the program by a transfer of control.

Use variable `accumulator` to represent the accumulator register. Use variable `counter` to keep track of the location in memory that contains the instruction being performed. Use variable `operationCode` to indicate the operation currently being performed (i.e., the left two digits of the instruction word). Use variable `operand` to indicate the memory location on which the current instruction operates. Thus, `operand` is the rightmost two digits of the instruction currently being performed. Do not execute instructions directly from memory. Rather, transfer the next instruction to be performed from memory to a variable called `instructionRegister`. Then "pick off" the left two digits and place them in `operationCode`, and "pick off" the right two digits and place them in `operand`. When Simpletron begins execution, the special registers are all initialized to zero.

Now let us "walk through" the execution of the first SML instruction, +1009 in memory location 00. This is called an instruction execution cycle.

The counter tells us the location of the next instruction to be performed. We fetch the contents of that location from `memory` by using the C++ statement

```
instructionRegister = memory[counter];
```

The operation code and operand are extracted from the instruction register by the statements

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

[Page 468]

Now, the Simpletron must determine that the operation code is actually a read (versus a write, a load, etc.). A `switch` differentiates among the 12 operations of SML.

In the `switch` statement, the behavior of various SML instructions is simulated as shown in Fig. 8.42 (we leave the others to the reader).

**Figure 8.42. Behavior of SML instructions.**

|         |                                                      |
|---------|------------------------------------------------------|
| read:   | <code>cin &gt;&gt; memory[ operand ];</code>         |
| load:   | <code>accumulator = memory[ operand ];</code>        |
| add:    | <code>accumulator += memory[ operand ];</code>       |
| branch: | We will discuss the branch instructions shortly.     |
| halt:   | This instruction prints the message                  |
|         | <code>*** Simpletron execution terminated ***</code> |

The halt instruction also causes the Simpletron to print the name and contents of each register, as well as the complete contents of memory. Such a printout is often called a computer dump (and, no, a computer dump is not a place where old computers go). To help you program your dump function, a sample dump format is shown in Fig. 8.43. Note that a dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated. To format numbers with their sign as shown in the dump, use stream manipulator `showpos`. To disable the display of the sign, use stream manipulator `noshowpos`. For numbers that have fewer than four digits, you can format numbers with leading zeros between the sign and the value by using the following statement before outputting the value:

```
cout << setfill('0') << internal;
```

**Figure 8.43. A sample dump.**

|                     |       |       |       |       |       |       |       |       |       |
|---------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| <b>REGISTERS:</b>   |       |       |       |       |       |       |       |       |       |
| accumulator         | +0000 |       |       |       |       |       |       |       |       |
| counter             | 00    |       |       |       |       |       |       |       |       |
| instructionRegister | +0000 |       |       |       |       |       |       |       |       |
| operationCode       | 00    |       |       |       |       |       |       |       |       |
| operand             | 00    |       |       |       |       |       |       |       |       |
| <b>MEMORY:</b>      |       |       |       |       |       |       |       |       |       |
| 0                   | 1     | 2     | 3     | 4     | 5     | 6     | 7     | 8     | 9     |
| 0                   | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 10                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 20                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 30                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 40                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 50                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 60                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 70                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |
| 80                  | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 | +0000 |

```
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Parameterized stream manipulator `setfill` (from header `<iomanip>`) specifies the fill character that will appear between the sign and the value when a number is displayed with a field width of five characters but does not have four digits. (One position in the field width is reserved for the sign.) Stream manipulator `internal` indicates that the fill characters should appear between the sign and the numeric value.

[Page 469]

Let us proceed with the execution of our program's first instruction+1009 in location 00. As we have indicated, the `switch` statement simulates this by performing the C++ statement

```
cin >> memory[operand];
```

A question mark (?) should be displayed on the screen before the `cin` statement executes to prompt the user for input. The Simpletron waits for the user to type a value and press the Enter key. The value is then read into location 09.

At this point, simulation of the first instruction is complete. All that remains is to prepare the Simpletron to execute the next instruction. The instruction just performed was not a transfer of control, so we need merely increment the instruction counter register as follows:

```
++counter;
```

This completes the simulated execution of the first instruction. The entire process (i.e., the instruction execution cycle) begins anew with the fetch of the next instruction to execute.

Now let us consider how to simulate the branching instructions (i.e., the transfers of control). All we need to do is adjust the value in the instruction counter appropriately. Therefore, the unconditional branch instruction (40) is simulated in the `switch` as

```
counter = operand;
```

The conditional "branch if accumulator is zero" instruction is simulated as

```
if (accumulator == 0)
 counter = operand;
```

At this point, you should implement your Simpletron simulator and run each of the SML programs you wrote in [Exercise 8.18](#). You may embellish SML with additional features and provide for these in your simulator.

Your simulator should check for various types of errors. During the program loading phase, for example, each number the user types into the Simpletron's memory must be in the range -9999 to +9999. Your simulator should use a `while` loop to test that each number entered is in this range and, if not, keep prompting the user to reenter the number until the user enters a correct number.

During the execution phase, your simulator should check for various serious errors, such as attempts to divide by zero, attempts to execute invalid operation codes, accumulator overflows (i.e., arithmetic operations resulting in values larger than +9999 or smaller than -9999) and the like. Such serious errors are called **fatal errors**. When a fatal error is detected, your simulator should print an error message such as

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

and should print a full computer dump in the format we have discussed previously. This will help the user locate the error in the program.

 PREV

page footer

NEXT 

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 470]

```
1 // Ex. 8.21: ex08_21.cpp
2 // What does this program do?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 void mystery1(char *, const char *); // prototype
9
10 int main()
11 {
12 char string1[80];
13 char string2[80];
14
15 cout << "Enter two strings: ";
16 cin >> string1 >> string2;
17 mystery1(string1, string2);
18 cout << string1 << endl;
19 return 0; // indicates successful termination
20 } // end main
21
22 // What does this function do?
23 void mystery1(char *s1, const char *s2)
24 {
25 while (*s1 != '\0')
26 ++s1;
27
28 for (; *s1 = *s2; s1++, s2++)
29 ; // empty statement
30 } // end function mystery1
```

## 8.22

What does this program do?

```
1 // Ex. 8.22: ex08_22.cpp
2 // What does this program do?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
```

```

6 using std::endl;
7
8 int mystery2(const char *); // prototype
9
10 int main()
11 {
12 char string1[80];
13
14 cout << "Enter a string: ";
15 cin >> string1;
16 cout << mystery2(string1) << endl;
17 return 0; // indicates successful termination
18 } // end main
19

```

[Page 471]

```

20 // What does this function do?
21 int mystery2(const char *s)
22 {
23 int x;
24
25 for (x = 0; *s != '\0'; s++)
26 ++x;
27
28 return x;
29 } // end function mystery2

```

## 8.23

Find the error in each of the following segments. If the error can be corrected, explain how.

a.

```

int *number;
cout << number << endl;

```

b.

```

double *realPtr;
long *integerPtr;
integerPtr = realPtr;

```

c.

```

int *x, y;

```

```
x = y;
```

**d.**

```
char s[] = "this is a character array";
for (; *s != '\0'; s++)
 cout << *s << ' ';
```

**e.**

```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7;
```

**f.**

```
double x = 19.34;
double xPtr = &x;
cout << xPtr << endl;
```

**g.**

```
char *s;
cout << s << endl;
```

## 8.24

(Quicksort) You have previously seen the sorting techniques of the bucket sort and selection sort. We now present the recursive sorting technique called Quicksort. The basic algorithm for a single-subscripted array of values is as follows:

**a.**

**Partitioning Step:** Take the first element of the unsorted array and determine its final location in the sorted array (i.e., all values to the left of the element in the array are less than the element, and all values to the right of the element in the array are greater than the element). We now have one element in its proper location and two unsorted subarrays.

**b.**

**Recursive Step:** Perform Step 1 on each unsorted subarray.

Each time Step 1 is performed on a subarray, another element is placed in its final location of the sorted array, and two unsorted subarrays are created. When a subarray consists of one element, that subarray

must be sorted; therefore, that element is in its final location.

The basic algorithm seems simple enough, but how do we determine the final position of the first element of each subarray? As an example, consider the following set of values (the element in bold is the partitioning element it will be placed in its final location in the sorted array):

37 2 6 4 89 8 10 12 68 45

a.

Starting from the rightmost element of the array, compare each element with 37 until an element less than 37 is found. Then swap 37 and that element. The first element less than 37 is 12, so 37 and 12 are swapped. The values now reside in the array as follows:

12    2    6    4    89    8    10    37    68    45

Element 12 is in *italics* to indicate that it was just swapped with 37.

[Page 472]

b.

Starting from the left of the array, but beginning with the element after 12, compare each element with 37 until an element greater than 37 is found. Then swap 37 and that element. The first element greater than 37 is 89, so 37 and 89 are swapped. The values now reside in the array as follows:

12    2    6    4    37    8    10    89    68    45

c.

Starting from the right, but beginning with the element before 89, compare each element with 37 until an element less than 37 is found. Then swap 37 and that element. The first element less than 37 is 10, so 37 and 10 are swapped. The values now reside in the array as follows:

12    2    6    4    10    8    37    89    68    45

d.

Starting from the left, but beginning with the element after 10, compare each element with 37 until an element greater than 37 is found. Then swap 37 and that element. There are no more elements greater than 37, so when we compare 37 with itself, we know that 37 has been placed in its final location of the sorted array.

Once the partition has been applied to the array, there are two unsorted subarrays. The subarray with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subarray with values greater than 37 contains 89, 68 and 45. The sort continues with both subarrays being partitioned in the same manner as the original array.

Based on the preceding discussion, write recursive function `quicksort` to sort a single-subscripted integer array. The function should receive as arguments an integer array, a starting subscript and an ending subscript. Function `partition` should be called by `quicksort` to perform the partitioning step.

## 8.25

(Maze Traversal) The grid of hashes (#) and dots (.) in Fig. 8.44 is a two-dimensional array representation of a maze. In the two-dimensional array, the hashes represent the walls of the maze and the dots represent squares in the possible paths through the maze. Moves can be made only to a location in the array that contains a dot.

**Figure 8.44. Two-dimensional array representation of a maze.**

```
#
. . . #
. . # . # . # # # . #
. # # .
. . . . # # # . # . .
. # . # . # .
. . # . # . # . # .
. # . # . # . # .
. # .
. # # # .
. # . . .
#
```

There is a simple algorithm for walking through a maze that guarantees finding the exit (assuming that there is an exit). If there is not an exit, you will arrive at the starting location again. Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand from the wall, eventually you will arrive at the exit of the maze. There may be a shorter path than the one you have taken, but you are guaranteed to get out of the maze if you follow the algorithm.

Write recursive function `mazeTraverse` to walk through the maze. The function should receive arguments that include a 12-by-12 character array representing the maze and the starting location of the maze. As `mazeTraverse` attempts to locate the exit from the maze, it should place the character `x` in each square in the path. The function should display the maze after each move, so the user can watch as the maze is solved.

## 8.26

(Generating Mazes Randomly) Write a function `mazeGenerator` that takes as an argument a two-dimensional 12-by-12 character array and randomly produces a maze. The function should also provide the starting and ending locations of the maze. Try your function `mazeTraverse` from [Exercise 8.25](#), using several randomly generated mazes.

## 8.27

(Mazes of Any Size) Generalize functions `mazeTraverse` and `mazeGenerator` of [Exercise 8.25](#) and [Exercise 8.26](#) to process mazes of any width and height.

## 8.28

(Modifications to the Simpletron Simulator) In [Exercise 8.19](#), you wrote a software simulation of a computer that executes programs written in Simpletron Machine Language (SML). In this exercise, we propose several modifications and enhancements to the Simpletron Simulator. In [Exercise 21.26](#) and [Exercise 21.27](#), we propose building a compiler that converts programs written in a high-level programming language (a variation of BASIC) to SML. Some of the following modifications and enhancements may be required to execute the programs produced by the compiler. [Note: Some modifications may conflict with others and therefore must be done separately.]

a.

Extend the Simpletron Simulator's memory to contain 1000 memory locations to enable the Simpletron to handle larger programs.

b.

Allow the simulator to perform modulus calculations. This requires an additional Simpletron Machine Language instruction.

c.

Allow the simulator to perform exponentiation calculations. This requires an additional Simpletron Machine Language instruction.

Modify the simulator to use hexadecimal values rather than integer values to represent Simpletron Machine Language instructions.

e.

Modify the simulator to allow output of a newline. This requires an additional Simpletron Machine Language instruction.

f.

Modify the simulator to process floating-point values in addition to integer values.

g.

Modify the simulator to handle string input. [Hint: Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII decimal equivalent of a character. Add a machine-language instruction that will input a string and store the string beginning at a specific Simpletron memory location. The first half of the word at that location will be a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII character expressed as two decimal digits. The machine-language instruction converts each character into its ASCII equivalent and assigns it to a halfword.]

h.

Modify the simulator to handle output of strings stored in the format of part (g). [Hint: Add a machine-language instruction that will print a string beginning at a certain Simpletron memory location. The first half of the word at that location is a count of the number of characters in the string (i.e., the length of the string). Each succeeding halfword contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and prints the string by translating each two-digit number into its equivalent character.]

i.

Modify the simulator to include instruction `SML_DEBUG` that prints a memory dump after each instruction executes. Give `SML_DEBUG` an operation code of 44. The word +4401 turns on debug mode, and +4400 turns off debug mode.

```
1 // Ex. 8.29: ex08_29.cpp
2 // What does this program do?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 bool mystery3(const char * , const char *) ; // prototype
9
10 int main()
11 {
12 char string1[80] , string2[80];
13
14 cout << "Enter two strings: ";
15 cin >> string1 >> string2;
16 cout << "The result is " << mystery3(string1, string2) << endl;
17 return 0; // indicates successful termination
18 } // end main
19
20 // What does this function do?
21 bool mystery3(const char *s1, const char *s2)
22 {
23 for (; *s1 != '\0' && *s2 != '\0'; s1++, s2++)
24
25 if (*s1 != *s2)
26 return false;
27
28 return true;
29 } // end function mystery3
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 475]

After completing the program, modify it to produce a short story consisting of several of these sentences. (How about the possibility of a random term-paper writer!)

### 8.33

(Limericks) A limerick is a humorous five-line verse in which the first and second lines rhyme with the fifth, and the third line rhymes with the fourth. Using techniques similar to those developed in [Exercise 8.32](#), write a C++ program that produces random limericks. Polishing this program to produce good limericks is a challenging problem, but the result will be worth the effort!

### 8.34

Write a program that encodes English language phrases into pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig Latin phrases. For simplicity, use the following algorithm: To form a pig-Latin phrase from an English-language phrase, tokenize the phrase into words with function `strtok`. To translate each English word into a pig-Latin word, place the first letter of the English word at the end of the English word and add the letters "ay." Thus, the word "jump" becomes "umpjay," the word "the" becomes "hetay" and the word "computer" becomes "omputercay." Blanks between words remain as blanks. Assume that the English phrase consists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Function `printLatinWord` should display each word. [Hint: Each time a token is found in a call to `strtok`, pass the token pointer to function `printLatinWord` and print the pig-Latin word.]

### 8.35

Write a program that inputs a telephone number as a string in the form (555) 555-5555. The program should use function `strtok` to extract the area code as a token, the first three digits of the phone number as a token, and the last four digits of the phone number as a token. The seven digits of the phone number should be concatenated into one string. Both the area code and the phone number should be printed.

### 8.36

Write a program that inputs a line of text, tokenizes the line with function `strtok` and outputs the tokens in reverse order.

### 8.37

Use the string-comparison functions discussed in [Section 8.13.2](#) and the techniques for sorting arrays developed in [Chapter 7](#) to write a program that alphabetizes a list of strings. Use the names of 10 or 15 towns in your area as data for your program.

### 8.38

Write two versions of each string copy and string-concatenation function in [Fig. 8.30](#). The first version should use array subscripting, and the second should use pointers and pointer arithmetic.

### 8.39

Write two versions of each string-comparison function in [Fig. 8.30](#). The first version should use array subscripting, and the second should use pointers and pointer arithmetic.

### 8.40

Write two versions of function `strlen` in [Fig. 8.30](#). The first version should use array subscripting, and the second should use pointers and pointer arithmetic.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 476]

## 8.41

(Text Analysis) The availability of computers with string-manipulation capabilities has resulted in some rather interesting approaches to analyzing the writings of great authors. Much attention has been focused on whether William Shakespeare ever lived. Some scholars believe there is substantial evidence indicating that Christopher Marlowe or other authors actually penned the masterpieces attributed to Shakespeare. Researchers have used computers to find similarities in the writings of these two authors. This exercise examines three methods for analyzing texts with a computer. Note that thousands of texts, including Shakespeare, are available online at [www.gutenberg.org](http://www.gutenberg.org).

a.

Write a program that reads several lines of text from the keyboard and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the phrase

To be, or not to be: that is the question:

contains one "a," two "b's," no "c's," etc.

b.

Write a program that reads several lines of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, etc., appearing in the text. For example, the phrase

Whether 'tis nobler in the mind to suffer

contains the following word lengths and occurrences:

| Word length | Occurrences        |
|-------------|--------------------|
| 1           | 0                  |
| 2           | 2                  |
| 3           | 1                  |
| 4           | 2 (including 'tis) |

|   |   |
|---|---|
| 5 | 0 |
| 6 | 2 |
| 7 | 1 |

**c.**

Write a program that reads several lines of text and prints a table indicating the number of occurrences of each different word in the text. The first version of your program should include the words in the table in the same order in which they appear in the text. For example, the lines

To be, or not to be: that is the question:  
Whether 'tis nobler in the mind to suffer

contain the words "to" three times, the word "be" two times, the word "or" once, etc. A more interesting (and useful) printout should then be attempted in which the words are sorted alphabetically.

**8.42**

(Word Processing) One important function in word-processing systems is type justification—the alignment of words to both the left and right margins of a page. This generates a professional-looking document that gives the appearance of being set in type rather than prepared on a typewriter. Type justification can be accomplished on computer systems by inserting blank characters between each of the words in a line so that the rightmost word aligns with the right margin.

[Page 477]

Write a program that reads several lines of text and prints this text in type-justified format. Assume that the text is to be printed on paper 8 1/2 inches wide and that one-inch margins are to be allowed on both the left and right sides. Assume that the computer prints 10 characters to the horizontal inch. Therefore, your program should print 6 1/2 inches of text, or 65 characters per line.

**8.43**

(Printing Dates in Various Formats) Dates are commonly printed in several different formats in business correspondence. Two of the more common formats are

07/21/1955

July 21, 1955

Write a program that reads a date in the first format and prints that date in the second format.

### 8.44

(Check Protection) Computers are frequently employed in check-writing systems such as payroll and accounts-payable applications. Many strange stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of \$1 million. Weird amounts are printed by computerized check-writing systems, because of human error or machine failure. Systems designers build controls into their systems to prevent such erroneous checks from being issued.

Another serious problem is the intentional alteration of a check amount by someone who intends to cash a check fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called check protection.

Checks designed for imprinting by computer contain a fixed number of spaces in which the computer may print an amount. Suppose that a paycheck contains eight blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all eight of those spaces will be filled, for example,

1 , 230 . 60 (check amount)

-----

12345678 (position numbers)

On the other hand, if the amount is less than \$1000, then several of the spaces would ordinarily be left blank. For example,

99 . 87

-----

12345678

contains three blank spaces. If a check is printed with blank spaces, it is easier for someone to alter the amount of the check. To prevent a check from being altered, many check-writing systems insert leading asterisks to protect the amount as follows:

\*\*\*99 . 87

-----

12345678

Write a program that inputs a dollar amount to be printed on a check and then prints the amount in check-protected format with leading asterisks if necessary. Assume that nine spaces are available for printing an amount.

### 8.45

(Writing the Word Equivalent of a Check Amount) Continuing the discussion of the previous example, we reiterate the importance of designing check-writing systems to prevent alteration of check amounts. One common security method requires that the check amount be both written in numbers and "spelled out" in words. Even if someone is able to alter the numerical amount of the check, it is extremely difficult to change the amount in words.

[Page 478]

Write a program that inputs a numeric check amount and writes the word equivalent of the amount. Your program should be able to handle check amounts as large as \$99.99. For example, the amount 112.43 should be written as

ONE HUNDRED TWELVE and 43/100

### 8.46

(Morse Code) Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit and a few special characters (such as period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound, and the dash represents a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems.

Separation between words is indicated by a space, or, quite simply, the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short period of time during which no sound is transmitted. The international version of the Morse code appears in [Fig. 8.45](#).

**Figure 8.45. Morse code alphabet.**

| Character | Code  | Character | Code |
|-----------|-------|-----------|------|
| A         | . -   | N         | - .  |
| B         | - ... | O         | ---  |

|        |             |   |             |
|--------|-------------|---|-------------|
| C      | - . - .     | P | . - - .     |
| D      | - . .       | Q | . - - . -   |
| E      | .           | R | . - .       |
| F      | . . - .     | S | . . .       |
| G      | . - - .     | T | -           |
| H      | . . . .     | U | . . -       |
| I      | . . .       | V | . . . -     |
| J      | . - - -     | W | . - -       |
| K      | - . -       | X | - . . -     |
| L      | . - . .     | Y | - . - -     |
| M      | --          | Z | - - . .     |
| Digits |             |   |             |
| 1      | . - - - -   | 6 | - . . . .   |
| 2      | . . - - -   | 7 | - - . . .   |
| 3      | . . . - - - | 8 | - - - . . . |
| 4      | . . . . -   | 9 | - - - - .   |
| 5      | . . . . .   | 0 | - - - - -   |

Write a program that reads an English-language phrase and encodes it into Morse code. Also write a program that reads a phrase in Morse code and converts it into the English-language equivalent. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word.

---

[Page 479]

### 8.47

(A Metric Conversion Program) Write a program that will assist the user with metric conversions. Your program should allow the user to specify the names of the units as strings (i.e., centimeters, liters, grams, etc., for the metric system and inches, quarts, pounds, etc., for the English system) and should respond to simple questions such as

"How many inches are in 2 meters?"  
 "How many liters are in 10 quarts?"

Your program should recognize invalid conversions. For example, the question

"How many feet are in 5 kilograms?"

is not meaningful, because "feet" are units of length, while "kilograms" are units of weight.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 479 (continued)]

## A Challenging String-Manipulation Project

- 8.48** (A Crossword Puzzle Generator) Most people have worked a crossword puzzle, but few have ever attempted to generate one. Generating a crossword puzzle is a difficult problem. It is suggested here as a string-manipulation project requiring substantial sophistication and effort. There are many issues that the programmer must resolve to get even the simplest crossword puzzle generator program working. For example, how does one represent the grid of a crossword puzzle inside the computer? Should one use a series of strings, or should two-dimensional arrays be used? The programmer needs a source of words (i.e., a computerized dictionary) that can be directly referenced by the program. In what form should these words be stored to facilitate the complex manipulations required by the program? The really ambitious reader will want to generate the "clues" portion of the puzzle, in which the brief hints for each "across" word and each "down" word are printed for the puzzle worker. Merely printing a version of the blank puzzle itself is not a simple problem.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 481]

## Outline

[9.1 Introduction](#)

[9.2 Time Class Case Study](#)

[9.3 Class Scope and Accessing Class Members](#)

[9.4 Separating Interface from Implementation](#)

[9.5 Access Functions and Utility Functions](#)

[9.6 Time Class Case Study: Constructors with Default Arguments](#)

[9.7 Destructors](#)

[9.8 When Constructors and Destructors Are Called](#)

[9.9 Time Class Case Study: A Subtle TrapReturning a Reference to a `private` Data Member](#)

[9.10 Default Memberwise Assignment](#)

[9.11 Software Reusability](#)

[9.12 \(Optional\) Software Engineering Case Study: Starting to Program the Classes of the ATM System](#)

[9.13 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

## Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 482]

In the second example of the Time class case study, we demonstrate how to pass arguments to constructors and show how default arguments can be used in a constructor to enable client code to initialize objects of a class using a variety of arguments. Next, we discuss a special member function called a destructor that is part of every class and is used to perform "termination housekeeping" on an object before the object is destroyed. We then demonstrate the order in which constructors and destructors are called, because your programs' correctness depends on using properly initialized objects that have not yet been destroyed.

Our last example of the Time class case study in this chapter shows a dangerous programming practice in which a member function returns a reference to `private` data. We discuss how this breaks the encapsulation of a class and allows client code to directly access an object's data. This last example shows that objects of the same class can be assigned to one another using default memberwise assignment, which copies the data members in the object on the right side of the assignment into the corresponding data members of the object on the left side of the assignment. The chapter concludes with a discussion of software reusability.

 PREV

page footer

NEXT 

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 483]

## Time Class Definition

The class definition ([Fig. 9.1](#)) contains prototypes (lines 1316) for member functions `Time`, `setTime`, `printUniversal` and `printStandard`. The class includes private integer members `hour`, `minute` and `second` (lines 1820). Class `Time`'s private data members can be accessed only by its four member functions. [Chapter 12](#) introduces a third access specifier, `protected`, as we study inheritance and the part it plays in object-oriented programming.

### Good Programming Practice 9.1



For clarity and readability, use each access specifier only once in a class definition. Place `public` members first, where they are easy to locate.

### Software Engineering Observation 9.1



Each element of a class should have `private` visibility unless it can be proven that the element needs `public` visibility. This is another example of the principle of least privilege.

In [Fig. 9.1](#), note that the class definition is enclosed in the following **preprocessor wrapper** (lines 57 and 23):

```
// prevent multiple inclusions of header file
#ifndef TIME_H
#define TIME_H

...
#endif
```

When we build larger programs, other definitions and declarations will also be placed in header files. The preceding preprocessor wrapper prevents the code between `#ifndef` (which means "if not defined") and `#endif` from being included if the name `TIME_H` has been defined. If the header has not been included

previously in a file, the name TIME\_H is defined by the `#define` directive and the header file statements are included. If the header has been included previously, TIME\_H is defined already and the header file is not included again. Attempts to include a header file multiple times (inadvertently) typically occur in large programs with many header files that may themselves include other header files. [Note: The commonly used convention for the symbolic constant name in the preprocessor directives is simply the header file name in upper case with the underscore character replacing the period.]

### Error-Prevention Tip 9.1



Use `#ifndef`, `#define` and `#endif` preprocessor directives to form a preprocessor wrapper that prevents header files from being included more than once in a program.

### Good Programming Practice 9.2



Use the name of the header file in upper case with the period replaced by an underscore in the `#ifndef` and `#define` preprocessor directives of a header file.

## Time Class Member Functions

In Fig. 9.2, the Time constructor (lines 1417) initializes the data members to 0 (i.e., the universal-time equivalent of 12 AM). This ensures that the object begins in a consistent state. Invalid values cannot be stored in the data members of a Time object, because the constructor is called when the Time object is created, and all subsequent attempts by a client to modify the data members are scrutinized by function setTime (discussed shortly). Finally, it is important to note that the programmer can define several overloaded constructors for a class.

---

[Page 484]

**Figure 9.2. Time class member-function definitions.**

```

1 // Fig. 9.2: Time.cpp
2 // Member-function definitions for class Time.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // include definition of class Time from Time.h
11
12 // Time constructor initializes each data member to zero.
13 // Ensures all Time objects start in a consistent state.
14 Time::Time()
15 {
16 hour = minute = second = 0;
17 } // end Time constructor
18
19 // set new Time value using universal time; ensure that
20 // the data remains consistent by setting invalid values to zero
21 void Time::setTime(int h, int m, int s)
22 {
23 hour = (h >= 0 && h < 24) ? h : 0; // validate hour
24 minute = (m >= 0 && m < 60) ? m : 0; // validate minute
25 second = (s >= 0 && s < 60) ? s : 0; // validate second
26 } // end function setTime
27
28 // print Time in universal-time format (HH:MM:SS)
29 void Time::printUniversal()
30 {
31 cout << setfill('0') << setw(2) << hour << ":"
32 << setw(2) << minute << ":" << setw(2) << second;
33 } // end function printUniversal
34
35 // print Time in standard-time format (HH:MM:SS AM or PM)
36 void Time::printStandard()
37 {
38 cout << ((hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
39 << setfill('0') << setw(2) << minute << ":" << setw(2)
40 << second << (hour < 12 ? " AM" : " PM");
41 } // end function printStandard

```

The data members of a class cannot be initialized where they are declared in the class body. It is strongly recommended that these data members be initialized by the class's constructor (as there is no default initialization for fundamental-type data members). Data members can also be assigned values by Time's

set functions. [Note: Chapter 10 demonstrates that only a class's static const data members of integral or enum types can be initialized in the class's body.]

---

[Page 485]

## Common Programming Error 9.1



Attempting to initialize a non-static data member of a class explicitly in the class definition is a syntax error.

Function `setTime` (lines 2126) is a public function that declares three `int` parameters and uses them to set the time. A conditional expression tests each argument to determine whether the value is in a specified range. For example, the `hour` value (line 23) must be greater than or equal to 0 and less than 24, because the universal-time format represents hours as integers from 0 to 23 (e.g., 1 PM is hour 13 and 11 PM is hour 23; midnight is hour 0 and noon is hour 12). Similarly, both `minute` and `second` values (lines 24 and 25) must be greater than or equal to 0 and less than 60. Any values outside these ranges are set to zero to ensure that a `Time` object always contains consistent data—that is, the object's data values are always kept in range, even if the values provided as arguments to function `setTime` were incorrect. In this example, zero is a consistent value for `hour`, `minute` and `second`.

A value passed to `setTime` is a correct value if it is in the allowed range for the member it is initializing. So, any number in the range 023 would be a correct value for the `hour`. A correct value is always a consistent value. However, a consistent value is not necessarily a correct value. If `setTime` sets `hour` to 0 because the argument received was out of range, then `hour` is correct only if the current time is coincidentally midnight.

Function `printUniversal` (lines 2933 of Fig. 9.2) takes no arguments and outputs the date in universal-time format, consisting of three colon-separated pairs of digits for the hour, minute and second, respectively. For example, if the time were 1:30:07 PM, function `printUniversal` would return 13 : 30 : 07. Note that line 31 uses parameterized stream manipulator `setfill` to specify the **fill character** that is displayed when an integer is output in a field wider than the number of digits in the value. By default, the fill characters appear to the left of the digits in the number. In this example, if the minute value is 2, it will be displayed as 02, because the fill character is set to zero ('0'). If the number being output fills the specified field, the fill character will not be displayed. Note that, once the fill character is specified with `setfill`, it applies for all subsequent values that are displayed in fields wider than the value being displayed (i.e., `setfill` is a "sticky" setting). This is in contrast to `setw`, which applies only to the next value displayed (`setw` is a "nonsticky" setting).

## Error-Prevention Tip 9.2



Each sticky setting (such as a fill character or floating-point precision) should be restored to its previous setting when it is no longer needed. Failure to do so may result in incorrectly formatted output later in a program. [Chapter 15](#), Stream Input/Output, discusses how to reset the fill character and precision.

Function `printStandard` (lines 3641) takes no arguments and outputs the date in standard-time format, consisting of the `hour`, `minute` and `second` values separated by colons and followed by an AM or PM indicator (e.g., `1:27:06 PM`). Like function `printUniversal`, function `printStandard` uses `setfill( '0' )` to format the `minute` and `second` as two digit values with leading zeros if necessary. Line 38 uses a conditional operator (`? :`) to determine the value of `hour` to be displayed if the `hour` is 0 or 12 (AM or PM), it appears as 12; otherwise, the `hour` appears as a value from 1 to 11. The conditional operator in line 40 determines whether AM or PM will be displayed.

[Page 486]

## Defining Member Functions Outside the Class Definition; Class Scope

Even though a member function declared in a class definition may be defined outside that class definition (and "tied" to the class via the binary scope resolution operator), that member function is still within that **class's scope**; i.e., its name is known only to other members of the class unless referred to via an object of the class, a reference to an object of the class, a pointer to an object of the class or the binary scope resolution operator. We will say more about class scope shortly.

If a member function is defined in the body of a class definition, the C++ compiler attempts to inline calls to the member function. Member functions defined outside a class definition can be inlined by explicitly using keyword `inline`. Remember that the compiler reserves the right not to inline any function.

### Performance Tip 9.1



Defining a member function inside the class definition inlines the member function (if the compiler chooses to do so). This can improve performance.

### Software Engineering Observation 9.2



Defining a small member function inside the class definition does not promote the best software engineering, because clients of the class will be able to see the implementation of the function, and the client code must be recompiled if the function definition changes.

## Software Engineering Observation 9.3



Only the simplest and most stable member functions (i.e., whose implementations are unlikely to change) should be defined in the class header.

## Member Functions vs. Global Functions

It is interesting that the `printUniversal` and `printStandard` member functions take no arguments. This is because these member functions implicitly know that they are to print the data members of the particular `Time` object for which they are invoked. This can make member function calls more concise than conventional function calls in procedural programming.

## Software Engineering Observation 9.4



Using an object-oriented programming approach can often simplify function calls by reducing the number of parameters to be passed. This benefit of object-oriented programming derives from the fact that encapsulating data members and member functions within an object gives the member functions the right to access the data members.

## Software Engineering Observation 9.5



Member functions are usually shorter than functions in non-object-oriented programs, because the data stored in data members have ideally been validated by a constructor or by member functions that store new data. Because the data is already in the object, the member-function calls often have no arguments or at least have fewer arguments than typical function calls in non-object-oriented languages. Thus, the calls are shorter, the function definitions are shorter and the function prototypes are shorter. This facilitates many aspects of program development.

## Error-Prevention Tip 9.3



The fact that member function calls generally take either no arguments or substantially fewer arguments than conventional function calls in non-object-oriented languages reduces the likelihood of passing the wrong arguments, the wrong types of arguments or the wrong number of arguments.

## Using Class Time

Once class `Time` has been defined, it can be used as a type in object, array, pointer and reference declarations as follows:

```
Time sunset; // object of type Time
Time arrayOfTimes[5], // array of 5 Time objects
Time &dinnerTime = sunset; // reference to a Time object
Time *timePtr = &dinnerTime, // pointer to a Time object
```

Figure 9.3 uses class `Time`. Line 12 instantiates a single object of class `Time` called `t`. When the object is instantiated, the `Time` constructor is called to initialize each `private` data member to 0. Then, lines 16 and 18 print the time in universal and standard formats to confirm that the members were initialized properly. Line 20 sets a new time by calling member function `setTime`, and lines 24 and 26 print the time again in both formats. Line 28 attempts to use `setTime` to set the data members to invalid valuesfunction `setTime` recognizes this and sets the invalid values to 0 to maintain the object in a consistent state. Finally, lines 33 and 35 print the time again in both formats.

**Figure 9.3. Program to test class Time.**

(This item is displayed on page 488 in the print version)

```
1 // Fig. 9.3: fig09_03.cpp
2 // Program to test class Time.
3 // NOTE: This file must be compiled with Time.cpp.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Time.h" // include definition of class Time from Time.h
9
10 int main()
11 {
12 Time t; // instantiate object t of class Time
13
14 // output Time object t's initial values
15 cout << "The initial universal time is ";
16 t.printUniversal(); // 00:00:00
17 cout << "\nThe initial standard time is ";
18 t.printStandard(); // 12:00:00 AM
19
20 t.setTime(13, 27, 6); // change time
21 }
```

```

22 // output Time object t's new values
23 cout << "\n\nUniversal time after setTime is ";
24 t.printUniversal(); // 13:27:06
25 cout << "\nStandard time after setTime is ";
26 t.printStandard(); // 1:27:06 PM
27
28 t.setTime(99, 99, 99); // attempt invalid settings
29
30 // output t's values after specifying invalid values
31 cout << "\n\nAfter attempting invalid settings:"
32 << "\nUniversal time: ";
33 t.printUniversal(); // 00:00:00
34 cout << "\nStandard time: ";
35 t.printStandard(); // 12:00:00 AM
36 cout << endl;
37 return 0;
38 } // end main

```

The initial universal time is 00:00:00  
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06  
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:  
Universal time: 00:00:00  
Standard time: 12:00:00 AM

## Looking Ahead to Composition and Inheritance

Often, classes do not have to be created "from scratch." Rather, they can include objects of other classes as members or they may be **derived** from other classes that provide attributes and behaviors the new classes can use. Such software reuse can greatly enhance programmer productivity and simplify code maintenance. Including class objects as members of other classes is called **composition** (or **aggregation**) and is discussed in [Chapter 10](#). Deriving new classes from existing classes is called **inheritance** and is discussed in [Chapter 12](#).

## Object Size

People new to object-oriented programming often suppose that objects must be quite large because they

contain data members and member functions. Logically, this is true—the programmer may think of objects as containing data and functions (and our discussion has certainly encouraged this view); physically, however, this is not true.

### Performance Tip 9.2



Objects contain only data, so objects are much smaller than if they also contained member functions. Applying operator `sizeof` to a class name or to an object of that class will report only the size of the class's data members. The compiler creates one copy (only) of the member functions separate from all objects of the class. All objects of the class share this one copy. Each object, of course, needs its own copy of the class's data, because the data can vary among the objects. The function code is nonmodifiable (also called **reentrant code** or **pure procedure**) and, hence, can be shared among all objects of one class.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

---

[Page 488]

[Page 489]

Member functions of a class can be overloaded, but only by other member functions of that class. To overload a member function, simply provide in the class definition a prototype for each version of the overloaded function, and provide a separate function definition for each version of the function.

Variables declared in a member function have block scope and are known only to that function. If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the block-scope variable in the block scope. Such a hidden variable can be accessed by preceding the variable name with the class name followed by the scope resolution operator (::). Hidden global variables can be accessed with the unary scope resolution operator (see [Chapter 6](#)).

The dot member selection operator (.) is preceded by an object's name or with a reference to an object to access the object's members. The arrow member selection operator (->) is preceded by a pointer to an object to access the object's members.

[Figure 9.4](#) uses a simple class called Count (lines 825) with private data member x of type int (line 24), public member function setx (lines 1215) and public member function print (lines 1821) to illustrate accessing the members of a class with the member selection operators. For simplicity, we have included this small class in the same file as the main function that uses it. Lines 2931 create three variables related to type Countcounter (a Count object), counterPtr (a pointer to a Count object) and counterRef (a reference to a Count object). Variable counterRef refers to counter, and variable counterPtr points to counter. In lines 3435 and 3839, note that the program can invoke member functions setx and print by using the dot (.) member selection operator preceded by either the name of the object (counter) or a reference to the object (counterRef, which is an alias for counter). Similarly, lines 4243 demonstrate that the program can invoke member functions setx and print by using a pointer (countPtr) and the arrow (->) member selection operator.

**Figure 9.4. Accessing an object's member functions through each type of object handle—the object's name, a reference to the object and a pointer to the object.**

(This item is displayed on page 490 in the print version)

```

1 // Fig. 9.4: fig09_04.cpp
2 // Demonstrating the class member access operators . and ->
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // class Count definition
8 class Count
9 {
10 public: // public data is dangerous
11 // sets the value of private data member x
12 void setX(int value)
13 {
14 x = value;
15 } // end function setX
16
17 // prints the value of private data member x
18 void print()
19 {
20 cout << x << endl;
21 } // end function print
22
23 private:
24 int x;
25 }; // end class Count
26
27 int main()
28 {
29 Count counter; // create counter object
30 Count *counterPtr = &counter; // create pointer to counter
31 Count &counterRef = counter; // create reference to counter
32
33 cout << "Set x to 1 and print using the object's name: ";
34 counter.setX(1); // set data member x to 1
35 counter.print(); // call member function print
36
37 cout << "Set x to 2 and print using a reference to an object: ";
38 counterRef.setX(2); // set data member x to 2
39 counterRef.print(); // call member function print
40
41 cout << "Set x to 3 and print using a pointer to an object: ";
42 counterPtr->setX(3); // set data member x to 3
43 counterPtr->print(); // call member function print
44 return 0;
45 } // end main

```

Set x to 1 and print using the object's name: 1  
Set x to 2 and print using a reference to an object: 2  
Set x to 3 and print using a pointer to an object: 3

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 491]

## Software Engineering Observation 9.7



Information important to the interface to a class should be included in the header file. Information that will be used only internally in the class and will not be needed by clients of the class should be included in the unpublished source file. This is yet another example of the principle of least privilege.

PREV

page footer

NEXT

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 493]

**Figure 9.6. SalesPerson class member-function definitions.**

(This item is displayed on pages 492 - 493 in the print version)

```
1 // Fig. 9.6: SalesPerson.cpp
2 // Member functions for class SalesPerson.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 #include "SalesPerson.h" // include SalesPerson class definition
13
14 // initialize elements of array sales to 0.0
15 SalesPerson::SalesPerson()
16 {
17 for (int i = 0; i < 12; i++)
18 sales[i] = 0.0;
19 } // end SalesPerson constructor
20
21 // get 12 sales figures from the user at the keyboard
22 void SalesPerson::getSalesFromUser()
23 {
24 double salesFigure;
25
26 for (int i = 1; i <= 12; i++)
27 {
28 cout << "Enter sales amount for month " << i << ": ";
29 cin >> salesFigure;
30 setSales(i, salesFigure);
31 } // end for
32 } // end function getSalesFromUser
33
34 // set one of the 12 monthly sales figures; function subtracts
35 // one from month value for proper subscript in sales array
36 void SalesPerson::setSales(int month, double amount)
```

```

37 {
38 // test for valid month and amount values
39 if (month >= 1 && month <= 12 && amount > 0)
40 sales[month - 1] = amount; // adjust for subscripts 0-11
41 else // invalid month or amount value
42 cout << "Invalid month or sales figure" << endl;
43 } // end function setSales
44
45 // print total annual sales (with the help of utility function)
46 void SalesPerson::printAnnualSales()
47 {
48 cout << setprecision(2) << fixed
49 << "\nThe total annual sales are: $"
50 << totalAnnualSales() << endl; // call utility function
51 } // end function printAnnualSales
52
53 // private utility function to total annual sales
54 double SalesPerson::totalAnnualSales()
55 {
56 double total = 0.0; // initialize total
57
58 for (int i = 0; i < 12; i++) // summarize sales results
59 total += sales[i]; // add month i sales to total
60
61 return total;
62 } // end function totalAnnualSales

```

In Fig. 9.7, notice that the application's `main` function includes only a simple sequence of member-function calls—there are no control statements. The logic of manipulating the `sales` array is completely encapsulated in class `SalesPerson`'s member functions.

**Figure 9.7. Utility function demonstration.**

(This item is displayed on pages 493 - 494 in the print version)

```

1 // Fig. 9.7: fig09_07.cpp
2 // Demonstrating a utility function.
3 // Compile this program with SalesPerson.cpp
4
5 // include SalesPerson class definition from SalesPerson.h
6 #include "SalesPerson.h"
7
8 int main()
9 {
10 SalesPerson s; // create SalesPerson object s
11
12 s.getSalesFromUser(); // note simple sequential code;
13 s.printAnnualSales(); // no control statements in main
14 return 0;
15 } // end main

```

```

Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

```

The total annual sales are: \$60120.59

## Software Engineering Observation 9.8



A phenomenon of object-oriented programming is that once a class is defined, creating and manipulating objects of that class often involve issuing only a simple sequence of memberfunction callsfew, if any, control statements are needed. By contrast, it is common to have control statements in the implementation of a class's member functions.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 495]

[Page 496]

**Figure 9.8. Time class containing a constructor with default arguments.**

(This item is displayed on page 494 in the print version)

```
1 // Fig. 9.8: Time.h
2 // Declaration of class Time.
3 // Member functions defined in Time.cpp.
4
5 // prevent multiple inclusions of header file
6 #ifndef TIME_H
7 #define TIME_H
8
9 // Time abstract data type definition
10 class Time
11 {
12 public:
13 Time(int = 0, int = 0, int = 0); // default constructor
14
15 // set functions
16 void setTime(int, int, int); // set hour, minute, second
17 void setHour(int); // set hour (after validation)
18 void setMinute(int); // set minute (after validation)
19 void setSecond(int); // set second (after validation)
20
21 // get functions
22 int getHour(); // return hour
23 int getMinute(); // return minute
24 int getSecond(); // return second
25
26 void printUniversal(); // output time in universal-time format
27 void printStandard(); // output time in standard-time format
28 private:
29 int hour; // 0 - 23 (24-hour clock format)
30 int minute; // 0 - 59
31 int second; // 0 - 59
32 }; // end class Time
33
34 #endif
```

**Figure 9.9. Time class member-function definitions including a constructor that takes arguments.**

(This item is displayed on pages 495 - 496 in the print version)

```

1 // Fig. 9.9: Time.cpp
2 // Member-function definitions for class Time.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // include definition of class Time from Time.h
11
12 // Time constructor initializes each data member to zero;
13 // ensures that Time objects start in a consistent state
14 Time::Time(int hr, int min, int sec)
15 {
16 setTime(hr, min, sec); // validate and set time
17 } // end Time constructor
18
19 // set new Time value using universal time; ensure that
20 // the data remains consistent by setting invalid values to zero
21 void Time::setTime(int h, int m, int s)
22 {
23 setHour(h); // set private field hour
24 setMinute(m); // set private field minute
25 setSecond(s); // set private field second
26 } // end function setTime
27
28 // set hour value
29 void Time::setHour(int h)
30 {
31 hour = (h >= 0 && h < 24) ? h : 0; // validate hour
32 } // end function setHour
33
34 // set minute value
35 void Time::setMinute(int m)
36 {
37 minute = (m >= 0 && m < 60) ? m : 0; // validate minute
38 } // end function setMinute
39
40 // set second value
41 void Time::setSecond(int s)

```

```

42 {
43 second = (s >= 0 && s < 60) ? s : 0; // validate second
44 } // end function setTime
45
46 // return hour value
47 int Time::getHour()
48 {
49 return hour;
50 } // end function getHour
51
52 // return minute value
53 int Time::getMinute()
54 {
55 return minute;
56 } // end function getMinute
57
58 // return second value
59 int Time::getSecond()
60 {
61 return second;
62 } // end function getSecond
63
64 // print Time in universal-time format (HH:MM:SS)
65 void Time::printUniversal()
66 {
67 cout << setfill('0') << setw(2) << getHour() << ":"
68 << setw(2) << getMinute() << ":" << setw(2) << getSecond();
69 } // end function printUniversal
70
71 // print Time in standard-time format (HH:MM:SS AM or PM)
72 void Time::printStandard()
73 {
74 cout << ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12)
75 << ":" << setfill('0') << setw(2) << getMinute()
76 << ":" << setw(2) << getSecond() << (hour < 12 ? " AM" : " PM");
77 } // end function printStandard

```

In Fig. 9.9, line 16 of the constructor calls member function setTime with the values passed to the constructor (or the default values). Function setTime calls setHour to ensure that the value supplied for hour is in the range 023, then calls setMinute and setSecond to ensure that the values for minute and second are each in the range 059. If a value is out of range, that value is set to zero (to ensure that each data member remains in a consistent state). In Chapter 16, Exception Handling, we throw exceptions to inform the user that a value is out of range, rather than simply assigning a default consistent value.

Note that the `Time` constructor could be written to include the same statements as member function `setTime`, or even the individual statements in the `setHour`, `setMinute` and `setSecond` functions. Calling `setHour`, `setMinute` and `setSecond` from the constructor may be slightly more efficient because the extra call to `setTime` would be eliminated. Similarly, copying the code from lines 31, 37 and 43 into constructor would eliminate the overhead of calling `setTime`, `setHour`, `setMinute` and `setSecond`. Coding the `Time` constructor or member function `setTime` as a copy of the code in lines 31, 37 and 43 would make maintenance of this class more difficult. If the implementations of `setHour`, `setMinute` and `setSecond` were to change, the implementation of any member function that duplicates lines 31, 37 and 43 would have to change accordingly. Having the `Time` constructor call `setTime` and having `setTime` call `setHour`, `setMinute` and `setSecond` enables us to limit the changes to code that validates the hour, minute or second to the corresponding set function. This reduces the likelihood of errors when altering the class's implementation. Also, the performance of the `Time` constructor and `setTime` can be enhanced by explicitly declaring them `inline` or by defining them in the class definition (which implicitly inlines the function definition).

#### Software Engineering Observation 9.9



If a member function of a class already provides all or part of the functionality required by a constructor (or other member function) of the class, call that member function from the constructor (or other member function). This simplifies the maintenance of the code and reduces the likelihood of an error if the implementation of the code is modified. As a general rule: Avoid repeating code.

#### Software Engineering Observation 9.10



Any change to the default argument values of a function requires the client code to be recompiled (to ensure that the program still functions correctly).

Function `main` in Fig. 9.10 initializes five `Time` objects one with all three arguments defaulted in the implicit constructor call (line 11), one with one argument specified (line 12), one with two arguments specified (line 13), one with three arguments specified (line 14) and one with three invalid arguments specified (line 15). Then the program displays each object in universal-time and standard-time formats.

#### Figure 9.10. Constructor with default arguments.

(This item is displayed on pages 497 - 498 in the print version)

```

1 // Fig. 9.10: fig09_10.cpp
2 // Demonstrating a default constructor for class Time.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Time.h" // include definition of class Time from Time.h
8
9 int main()
10 {
11 Time t1; // all arguments defaulted
12 Time t2(2); // hour specified; minute and second defaulted
13 Time t3(21, 34); // hour and minute specified; second defaulted
14 Time t4(12, 25, 42); // hour, minute and second specified
15 Time t5(27, 74, 99); // all bad values specified
16
17 cout << "Constructed with:\n\tt1: all arguments defaulted\n\t";
18 t1.printUniversal(); // 00:00:00
19 cout << "\n\t";
20 t1.printStandard(); // 12:00:00 AM
21
22 cout << "\n\tt2: hour specified; minute and second defaulted\n\t";
23 t2.printUniversal(); // 02:00:00
24 cout << "\n\t";
25 t2.printStandard(); // 2:00:00 AM
26
27 cout << "\n\tt3: hour and minute specified; second defaulted\n\t";
28 t3.printUniversal(); // 21:34:00
29 cout << "\n\t";
30 t3.printStandard(); // 9:34:00 PM
31
32 cout << "\n\tt4: hour, minute and second specified\n\t";
33 t4.printUniversal(); // 12:25:42
34 cout << "\n\t";
35 t4.printStandard(); // 12:25:42 PM
36
37 cout << "\n\tt5: all invalid values specified\n\t";
38 t5.printUniversal(); // 00:00:00
39 cout << "\n\t";
40 t5.printStandard(); // 12:00:00 AM
41 cout << endl;
42 return 0;
43 } // end main

```

Constructed with:

```
t1: all arguments defaulted
00:00:00
12:00:00 AM

t2: hour specified; minute and second defaulted
02:00:00
2:00:00 AM

t3: hour and minute specified; second defaulted
21:34:00
9:34:00 PM

t4: hour, minute and second specified
12:25:42
12:25:42 PM

t5: all invalid values specified
00:00:00
12:00:00 AM
```

[Page 499]

## Notes Regarding Class Time's Set and Get Functions and Constructor

Time's set and get functions are called throughout the body of the class. In particular, function `setTime` (lines 2126 of Fig. 9.9) calls functions `setHour`, `setMinute` and `setSecond`, and functions `printUniversal` and `printStandard` call functions `getHour`, `getMinute` and `getSecond` in line 6768 and lines 7476, respectively. In each case, these functions could have accessed the class's private data directly without calling the set and get functions. However, consider changing the representation of the time from three `int` values (requiring 12 bytes of memory) to a single `int` value representing the total number of seconds that have elapsed since midnight (requiring only four bytes of memory). If we made such a change, only the bodies of the functions that access the private data directly would need to change in particular, the individual set and get functions for the `hour`, `minute` and `second`. There would be no need to modify the bodies of functions `setTime`, `printUniversal` or `printStandard`, because they do not access the data directly. Designing the class in this manner reduces the likelihood of programming errors when altering the class's implementation.

Similarly, the `Time` constructor could be written to include a copy of the appropriate statements from function `setTime`. Doing so may be slightly more efficient, because the extra constructor call and call to

`setTime` are eliminated. However, duplicating statements in multiple functions or constructors makes changing the class's internal data representation more difficult. Having the `Time` constructor call function `setTime` directly requires any changes to the implementation of `setTime` to be made only once.

## Common Programming Error 9.2



A constructor can call other member functions of the class, such as set or get functions, but because the constructor is initializing the object, the data members may not yet be in a consistent state. Using data members before they have been properly initialized can cause logic errors.

[◀ PREV](#)[NEXT ▶](#)**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 500]

## Software Engineering Observation 9.11



As we will see in the remainder of the book, constructors and destructors have much greater prominence in C++ and object-oriented programming than is possible to convey after only our brief introduction here.

[PREV](#)  
**page footer**

[\*\*NEXT\*\*](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 501]

**Figure 9.11.** CreateAndDestroy class definition.

```

1 // Fig. 9.11: CreateAndDestroy.h
2 // Definition of class CreateAndDestroy.
3 // Member functions defined in CreateAndDestroy.cpp.
4 #include <string>
5 using std::string;
6
7 #ifndef CREATE_H
8 #define CREATE_H
9
10 class CreateAndDestroy
11 {
12 public:
13 CreateAndDestroy(int, string); // constructor
14 ~CreateAndDestroy(); // destructor
15 private:
16 int objectID; // ID number for object
17 string message; // message describing object
18 }; // end class CreateAndDestroy
19
20 #endif

```

**Figure 9.12.** CreateAndDestroy class member-function definitions.

```

1 // Fig. 9.12: CreateAndDestroy.cpp
2 // Member-function definitions for class CreateAndDestroy.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "CreateAndDestroy.h" // include CreateAndDestroy class definition
8
9 // constructor
10 CreateAndDestroy::CreateAndDestroy(int ID, string messageString)
11 {
12 objectID = ID; // set object's ID number
13 message = messageString; // set object's descriptive message

```

```

14 cout << "Object " << objectID << " constructor runs "
15 << message << endl;
16 } // end CreateAndDestroy constructor
17
18
19 // destructor
20 CreateAndDestroy::~CreateAndDestroy()
21 {
22 // output newline for certain objects; helps readability
23 cout << (objectID == 1 || objectID == 6 ? "\n" : "");
24
25 cout << "Object " << objectID << " destructor runs "
26 << message << endl;
27 } // end ~CreateAndDestroy destructor

```

---

[Page 502]

**Figure 9.13** defines object `first` (line 12) in global scope. Its constructor is actually called before any statements in `main` execute and its destructor is called at program termination after the destructors for all other objects have run.

Function `main` (lines 1426) declares three objects. Objects `second` (line 17) and `fourth` (line 23) are local automatic objects, and object `third` (line 18) is a `static` local object. The constructor for each of these objects is called when execution reaches the point where that object is declared. The destructors for objects `fourth` and then `second` are called (i.e., the reverse of the order in which their constructors were called) when execution reaches the end of `main`. Because object `third` is `static`, it exists until program termination. The destructor for object `third` is called before the destructor for global object `first`, but after all other objects are destroyed.

Function `create` (lines 2936) declares three objects `fifth` (line 32) and `seventh` (line 34) as local automatic objects, and `sixth` (line 33) as a `static` local object. The destructors for objects `seventh` and then `fifth` are called (i.e., the reverse of the order in which their constructors were called) when `create` terminates. Because `sixth` is `static`, it exists until program termination. The destructor for `sixth` is called before the destructors for `third` and `first`, but after all other objects are destroyed.

---

[Page 503]

**Figure 9.13. Order in which constructors and destructors are called.**

(This item is displayed on pages 502 - 503 in the print version)

```
1 // Fig. 9.13: fig09_13.cpp
2 // Demonstrating the order in which constructors and
3 // destructors are called.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "CreateAndDestroy.h" // include CreateAndDestroy class definition
9
10 void create(void); // prototype
11
12 CreateAndDestroy first(1, "(global before main)"); // global object
13
14 int main()
15 {
16 cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
17 CreateAndDestroy second(2, "(local automatic in main)");
18 static CreateAndDestroy third(3, "(local static in main)");
19
20 create(); // call function to create objects
21
22 cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
23 CreateAndDestroy fourth(4, "(local automatic in main)");
24 cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
25 return 0;
26 } // end main
27
28 // function to create objects
29 void create(void)
30 {
31 cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
32 CreateAndDestroy fifth(5, "(local automatic in create)");
33 static CreateAndDestroy sixth(6, "(local static in create)");
34 CreateAndDestroy seventh(7, "(local automatic in create)");
35 cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
36 } // end function create
```

Object 1 constructor runs (global before main)

MAIN FUNCTION: EXECUTION BEGINS

Object 2 constructor runs (local automatic in main)

Object 3 constructor runs (local static in main)

CREATE FUNCTION: EXECUTION BEGINS

Object 5 constructor runs (local automatic in create)

Object 6 constructor runs (local static in create)

Object 7 constructor runs (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS

Object 7 destructor runs (local automatic in create)

Object 5 destructor runs (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES

Object 4 constructor runs (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS

Object 4 destructor runs (local automatic in main)

Object 2 destructor runs (local automatic in main)

Object 6 destructor runs (local static in create)

Object 3 destructor runs (local static in main)

Object 1 destructor runs (global before main)

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 504]

**Figure 9.14.** Returning a reference to a private data member.

```
1 // Fig. 9.14: Time.h
2 // Declaration of class Time.
3 // Member functions defined in Time.cpp
4
5 // prevent multiple inclusions of header file
6 #ifndef TIME_H
7 #define TIME_H
8
9 class Time
10 {
11 public:
12 Time(int = 0, int = 0, int = 0);
13 void setTime(int, int, int);
14 int getHour();
15 int &badSetHour(int); // DANGEROUS reference return
16 private:
17 int hour;
18 int minute;
19 int second;
20 }; // end class Time
21
22 #endif
```

**Figure 9.15.** Returning a reference to a private data member.

(This item is displayed on pages 504 - 505 in the print version)

```

1 // Fig. 9.15: Time.cpp
2 // Member-function definitions for Time class.
3 #include "Time.h" // include definition of class Time
4
5 // constructor function to initialize private data;
6 // calls member function setTime to set variables;
7 // default values are 0 (see class definition)
8 Time::Time(int hr, int min, int sec)
9 {
10 setTime(hr, min, sec);
11 } // end Time constructor
12
13 // set values of hour, minute and second
14 void Time::setTime(int h, int m, int s)
15 {
16 hour = (h >= 0 && h < 24) ? h : 0; // validate hour
17 minute = (m >= 0 && m < 60) ? m : 0; // validate minute
18 second = (s >= 0 && s < 60) ? s : 0; // validate second
19 } // end function setTime
20
21 // return hour value
22 int Time::getHour()
23 {
24 return hour;
25 } // end function getHour
26
27 // POOR PROGRAMMING PRACTICE:
28 // Returning a reference to a private data member.
29 int &Time::badSetHour(int hh)
30 {
31 hour = (hh >= 0 && hh < 24) ? hh : 0;
32 return hour; // DANGEROUS reference return
33 } // end function badSetHour

```

[Page 505]

Figure 9.16 declares Time object `t` (line 12) and reference `hourRef` (line 15), which is initialized with the reference returned by the call `t.badSetHour(20)`. Line 17 displays the value of the alias `hourRef`. This shows how `hourRef` breaks the encapsulation of the class. statements in `main` should not have access to the private data of the class. Next, line 18 uses the alias to set the value of `hour` to 30 (an invalid value) and line 19 displays the value returned by function `getHour` to show that assigning a value to `hourRef` actually modifies the private data in the `Time` object `t`. Finally, line 23 uses the `badSetHour` function call itself as an lvalue and assigns 74 (another invalid value) to the reference

returned by the function. Line 28 again displays the value returned by function `getHour` to show that assigning a value to the result of the function call in line 23 modifies the private data in the `Time` object `t`.

[Page 506]

**Figure 9.16. Returning a reference to a private data member.**

(This item is displayed on pages 505 - 506 in the print version)

```
1 // Fig. 9.16: fig09_16.cpp
2 // Demonstrating a public member function that
3 // returns a reference to a private data member.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Time.h" // include definition of class Time
9
10 int main()
11 {
12 Time t; // create Time object
13
14 // initialize hourRef with the reference returned by badSetHour
15 int &hourRef = t.badSetHour(20); // 20 is a valid hour
16
17 cout << "Valid hour before modification: " << hourRef;
18 hourRef = 30; // use hourRef to set invalid value in Time object t
19 cout << "\nInvalid hour after modification: " << t.getHour();
20
21 // Dangerous: Function call that returns
22 // a reference can be used as an lvalue!
23 t.badSetHour(12) = 74; // assign another invalid value to hour
24
25 cout << "\n*****\n"
26 << "POOR PROGRAMMING PRACTICE!!!!!!\n"
27 << "t.badSetHour(12) as an lvalue, invalid hour: "
28 << t.getHour()
29 << "\n*****" << endl;
30 return 0;
31 } // end main
```

```
Valid hour before modification: 20
Invalid hour after modification: 30
```

```

POOR PROGRAMMING PRACTICE!!!!!!
t.badSetHour(12) as an lvalue, invalid hour: 74

```

#### Error-Prevention Tip 9.4



Returning a reference or a pointer to a private data member breaks the encapsulation of the class and makes the client code dependent on the representation of the class's data. So, returning pointers or references to private data is a dangerous practice that should be avoided.

[◀ PREV](#)

page footer

[NEXT ▶](#)

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 508]

**Figure 9.17. Date class header file.**

(This item is displayed on page 506 in the print version)

```
1 // Fig. 9.17: Date.h
2 // Declaration of class Date.
3 // Member functions are defined in Date.cpp
4
5 // prevent multiple inclusions of header file
6 #ifndef DATE_H
7 #define DATE_H
8
9 // class Date definition
10 class Date
11 {
12 public:
13 Date(int = 1, int = 1, int = 2000); // default constructor
14 void print();
15 private:
16 int month;
17 int day;
18 int year;
19 }; // end class Date
20
21 #endif
```

**Figure 9.18. Date class member-function definitions.**

(This item is displayed on page 507 in the print version)

```

1 // Fig. 9.18: Date.cpp
2 // Member-function definitions for class Date.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Date.h" // include definition of class Date from Date.h
8
9 // Date constructor (should do range checking)
10 Date::Date(int m, int d, int y)
11 {
12 month = m;
13 day = d;
14 year = y;
15 } // end constructor Date
16
17 // print Date in the format mm/dd/yyyy
18 void Date::print()
19 {
20 cout << month << '/' << day << '/' << year;
21 } // end function print

```

**Figure 9.19. Default memberwise assignment.**

(This item is displayed on pages 507 - 508 in the print version)

```

1 // Fig. 9.19: fig09_19.cpp
2 // Demonstrating that class objects can be assigned
3 // to each other using default memberwise assignment.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Date.h" // include definition of class Date from Date.h
9
10 int main()
11 {
12 Date date1(7, 4, 2004);
13 Date date2; // date2 defaults to 1/1/2000
14
15 cout << "date1 = ";
16 date1.print();
17 cout << "\ndate2 = ";
18 date2.print();

```

```

19
20 date2 = date1; // default memberwise assignment
21
22 cout << "\n\nAfter default memberwise assignment, date2 = ";
23 date2.print();
24 cout << endl;
25 return 0;
26 } // end main

```

date1 = 7/4/2004  
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004

Objects may be passed as function arguments and may be returned from functions. Such passing and returning is performed using pass-by-value by default a copy of the object is passed or returned. In such cases, C++ creates a new object and uses a **copy constructor** to copy the original object's values into the new object. For each class, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object. Like memberwise assignment, copy constructors can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory. Chapter 11 discusses how programmers can define a customized copy constructor that properly copies objects containing pointers to dynamically allocated memory.

### Performance Tip 9.3



*Passing an object by value* is good from a security standpoint, because the called function has no access to the original object in the caller, but pass-by-value can degrade performance when making a copy of a large object. An object can be passed by reference by passing either a pointer or a reference to the object. Pass-by-reference offers good performance but is weaker from a security standpoint, because the called function is given access to the original object. Pass-by-const-reference is a safe, good-performing alternative (this can be implemented with a const reference parameter or with a pointer-to-const-data parameter).

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 509]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

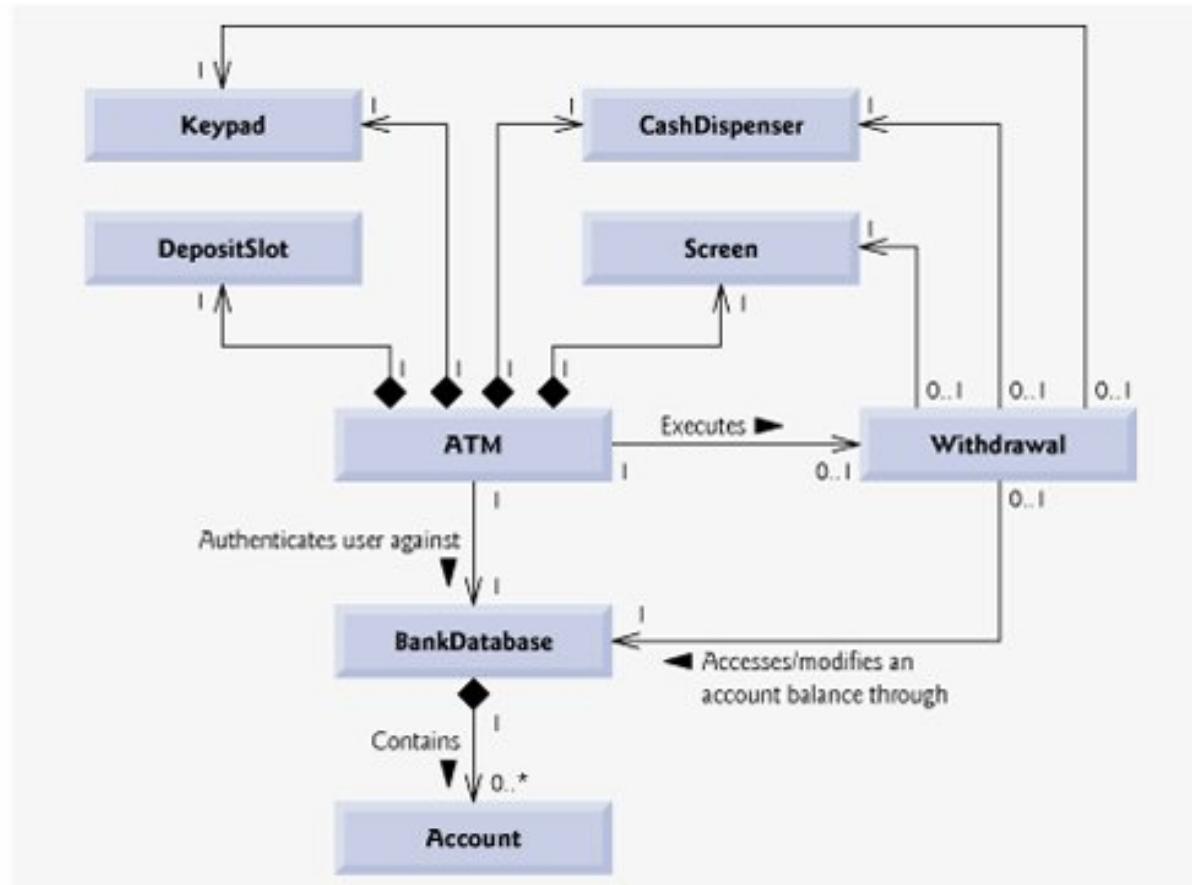
Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 510]

**Figure 9.21. Class diagram with navigability arrows.**

(This item is displayed on page 511 in the print version)

[\[View full size image\]](#)



Like the class diagram of Fig. 3.23 the class diagram of Fig. 9.21 omits classes BalanceInquiry and Deposit to keep the diagram simple. The navigability of the associations in which these classes participate closely parallels the navigability of class withdrawal's associations. Recall from Section 3.11 that BalanceInquiry has an association with class Screen. We can navigate from class BalanceInquiry to class Screen along this association, but we cannot navigate from class Screen to class BalanceInquiry. Thus, if we were to model class BalanceInquiry in Fig. 9.21, we would place a navigability arrow at class Screen's end of this association. Also recall that class Deposit associates with classes Screen, Keypad and DepositSlot. We can navigate from class Deposit to each of these classes, but not vice versa. We therefore would place navigability arrows at the Screen, Keypad and DepositSlot ends of these associations. [Note: We model these additional classes and associations in our final class diagram in Section 13.10, after we have simplified the structure of our system by incorporating the object-oriented concept of inheritance.]

## Implementing the ATM System from Its UML Design

We are now ready to begin implementing the ATM system. We first convert the classes in the diagrams of Fig. 9.20 and Fig. 9.21 into C++ header files. This code will represent the "skeleton" of the system. In Chapter 13, we modify the header files to incorporate the object-oriented concept of inheritance. In Appendix G, ATM Case Study Code, we present the complete working C++ code for our model.

As an example, we begin to develop the header file for class `Withdrawal` from our design of class `Withdrawal` in Fig. 9.20. We use this figure to determine the attributes and operations of the class. We use the UML model in Fig. 9.21 to determine the associations among classes. We follow the following five guidelines for each class:

1.

Use the name located in the first compartment of a class in a class diagram to define the class in a header file (Fig. 9.22). Use `#ifndef`, `#define` and `#endif` preprocessor directives to prevent the header file from being included more than once in a program.

**Figure 9.22. Definition of class `Withdrawal` enclosed in preprocessor wrappers.**

(This item is displayed on page 512 in the print version)

```

1 // Fig. 9.22: Withdrawal.h
2 // Definition of class Withdrawal that represents a withdrawal transaction
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 class Withdrawal
7 {
8 }; // end class Withdrawal
9
10#endif // WITHDRAWAL_H

```

2.

Use the attributes located in the class's second compartment to declare the data members. For example, the private attributes `accountNumber` and `amount` of class `Withdrawal` yield the code in Fig. 9.23.

**Figure 9.23. Adding attributes to the `Withdrawal` class header file.**

(This item is displayed on page 513 in the print version)

```

1 // Fig. 9.23: Withdrawal.h
2 // Definition of class Withdrawal that represents a withdrawal transaction
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 class Withdrawal
7 {
8 private:
9 // attributes
10 int accountNumber; // account to withdraw funds from
11 double amount; // amount to withdraw
12 }; // end class Withdrawal
13
14 #endif // WITHDRAWAL_H

```

**3.**

Use the associations described in the class diagram to declare references (or pointers, where appropriate) to other objects. For example, according to Fig. 9.21, Withdrawal can access one object of class Screen, one object of class Keypad, one object of class CashDispenser and one object of class BankDatabase. Class Withdrawal must maintain handles on these objects to send messages to them, so lines 1922 of Fig. 9.24 declare four references as private data members. In the implementation of class Withdrawal in Appendix G, a constructor initializes these data members with references to actual objects. Note that lines 69 #include the header files containing the definitions of classes Screen, Keypad, CashDispenser and BankDatabase so that we can declare references to objects of these classes in lines 1922.

---

[Page 512]

**Figure 9.24. Declaring references to objects associated with class Withdrawal.**

(This item is displayed on page 513 in the print version)

```

1 // Fig. 9.24: Withdrawal.h
2 // Definition of class Withdrawal that represents a withdrawal transaction
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 #include "Screen.h" // include definition of class Screen
7 #include "Keypad.h" // include definition of class Keypad
8 #include "CashDispenser.h" // include definition of class CashDispenser
9 #include "BankDatabase.h" // include definition of class BankDatabase
10
11 class Withdrawal
12 {
13 private:
14 // attributes

```

```

15 int accountNumber; // account to withdraw funds from
16 double amount; // amount to withdraw
17
18 // references to associated objects
19 Screen &screen; // reference to ATM's screen
20 Keypad &keypad; // reference to ATM's keypad
21 CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22 BankDatabase &bankDatabase; // reference to the account info database
23 } ; // end class Withdrawal
24
25 #endif // WITHDRAWAL_H

```

## 4.

It turns out that including the header files for classes `Screen`, `Keypad`, `CashDispenser` and `BankDatabase` in Fig. 9.24 does more than is necessary. Class `Withdrawal` contains references to objects of these classesit does not contain actual objectsand the amount of information required by the compiler to create a reference differs from that which is required to create an object. Recall that creating an object requires that you provide the compiler with a definition of the class that introduces the name of the class as a new user-defined type and indicates the data members that determine how much memory is required to store the object. Declaring a reference (or pointer) to an object, however, requires only that the compiler knows that the object's class existsit does not need to know the size of the object. Any reference (or pointer), regardless of the class of the object to which it refers, contains only the memory address of the actual object. The amount of memory required to store an address is a physical characteristic of the computer's hardware. The compiler thus knows the size of any reference (or pointer). As a result, including a class's full header file when declaring only a reference to an object of that class is unnecessarywe need to introduce the name of the class, but we do not need to provide the data layout of the object, because the compiler already knows the size of all references. C++ provides a statement called a **forward declaration** that signifies that a header file contains references or pointers to a class, but that the class definition lies outside the header file. We can replace the `#includes` in the `Withdrawal` class definition of Fig. 9.24 with forward declarations of classes `Screen`, `Keypad`, `CashDispenser` and `BankDatabase` (lines 69 in Fig. 9.25). Rather than `#include` the entire header file for each of these classes, we place only a forward declaration of each class in the header file for class `Withdrawal`. Note that if class `Withdrawal` contained actual objects instead of references (i.e., if the ampersands in lines 1922 were omitted), then we would indeed need to `#include` the full header files.

**Figure 9.25. Using forward declarations in place of #include directives.**

(This item is displayed on page 514 in the print version)

```

1 // Fig. 9.25: Withdrawal.h
2 // Definition of class Withdrawal that represents a withdrawal transaction
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 class Screen; // forward declaration of class Screen
7 class Keypad; // forward declaration of class Keypad
8 class CashDispenser; // forward declaration of class CashDispenser
9 class BankDatabase; // forward declaration of class BankDatabase
10
11 class Withdrawal
12 {
13 private:
14 // attributes
15 int accountNumber; // account to withdraw funds from
16 double amount; // amount to withdraw
17
18 // references to associated objects
19 Screen &screen; // reference to ATM's screen
20 Keypad &keypad; // reference to ATM's keypad
21 CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22 BankDatabase &bankDatabase; // reference to the account info database
23 }; // end class Withdrawal
24
25 #endif // WITHDRAWAL_H

```

---

[Page 513]

Note that using a forward declaration (where possible) instead of including a full header file helps avoid a preprocessor problem called a [circular include](#). This problem occurs when the header file for a class A `#includes` the header file for a class B and vice versa. Some preprocessors are not able to resolve such `#include` directives, causing a compilation error. If class A, for example, uses only a reference to an object of class B, then the `#include` in class A's header file can be replaced by a forward declaration of class B to prevent the circular include.

---

[Page 514]

5.

Use the operations located in the third compartment of [Fig. 9.20](#) to write the function prototypes of the class's member functions. If we have not yet specified a return type for an operation, we declare the member function with return type `void`. Refer to the class diagrams of [Figs. 6.226.25](#) to declare any necessary parameters. For example, adding the `public` operation `execute` in class `Withdrawal`, which has an empty parameter list, yields the prototype in line 15 of [Fig. 9.26](#). [Note: We code the definitions of member functions in `.cpp` files when we implement the complete ATM system in [Appendix G](#).]

**Figure 9.26. Adding operations to the withdrawal class header file.**

(This item is displayed on pages 514 - 515 in the print version)

```

1 // Fig. 9.26: Withdrawal.h
2 // Definition of class Withdrawal that represents a withdrawal transaction
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 class Screen; // forward declaration of class Screen
7 class Keypad; // forward declaration of class Keypad
8 class CashDispenser; // forward declaration of class CashDispenser
9 class BankDatabase; // forward declaration of class BankDatabase
10
11 class Withdrawal
12 {
13 public:
14 // operations
15 void execute(); // perform the transaction
16 private:
17 // attributes
18 int accountNumber; // account to withdraw funds from
19 double amount; // amount to withdraw
20
21 // references to associated objects
22 Screen &screen; // reference to ATM's screen
23 Keypad &keypad; // reference to ATM's keypad
24 CashDispenser &cashDispenser; // reference to ATM's cash dispenser
25 BankDatabase &bankDatabase; // reference to the account info database
26 }; // end class Withdrawal
27
28 #endif // WITHDRAWAL_H

```

[Page 515]

## Software Engineering Observation 9.12



Several UML modeling tools can convert UML-based designs into C++ code, considerably speeding the implementation process. For more information on these "automatic" code generators, refer to the Internet and Web resources listed at the end of [Section 2.8](#).

This concludes our discussion of the basics of generating class header files from UML diagrams. In the final "Software Engineering Case Study" section ([Section 3.11](#)), we demonstrate how to modify the header files to incorporate the object-oriented concept of inheritance.

## Software Engineering Case Study Self-Review Exercises

- 9.1** State whether the following statement is true or false, and if false, explain why: If an attribute of a class is marked with a minus sign (-) in a class diagram, the attribute is not directly accessible outside of the class.
- 9.2** In Fig. 9.21, the association between the ATM and the Screen indicates that:
- we can navigate from the Screen to the ATM
  - we can navigate from the ATM to the Screen
  - Both a and b; the association is bidirectional
  - None of the above
- 9.3** Write C++ code to begin implementing the design for class Account.

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

- 9.1** True. The minus sign (-) indicates private visibility. We've mentioned "friendship" as an exception to private visibility. Friendship is discussed in Chapter 10.
- 9.2** b.

**9.3** The design for class Account yields the header file in Fig. 9.27.

**Figure 9.27. Account class header file based on Fig. 9.20 and Fig. 9.21.**

```
1 // Fig. 9.27: Account.h
2 // Account class definition. Represents a bank account.
3 #ifndef ACCOUNT_H
4 #define ACCOUNT_H
5
6 class Account
7 {
8 public:
9 bool validatePIN(int); // is user-specified PIN correct?
10 double getAvailableBalance(); // returns available balance
11 double getTotalBalance(); // returns total balance
12 void credit(double); // adds an amount to the Account
13 void debit(double); // subtracts an amount from the Account
14 private:
15 int accountNumber; // account number
16 int pin; // PIN for authentication
17 double availableBalance; // funds available for withdrawal
18 double totalBalance; // funds available + funds waiting to clear
19 }; // end class Account
20
21 #endif // ACCOUNT_H
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 517]

[Chapter 10](#) presents additional class features. We will demonstrate how `const` can be used to indicate that a member function does not modify an object of a class. You will learn how to build classes with composition that is, classes that contain objects of other classes as members. We'll show how a class can allow so-called "friend" functions to access the class's non-public members. We'll also show how a class's non-static member functions can use a special pointer named `this` to access an object's members. Next, you'll learn how to use C++'s `new` and `delete` operators, which enable programmers to obtain and release memory as necessary during a program's execution.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 518]

- The dot member selection operator (.) is preceded by an object's name or by a reference to an object to access the object's `public` members.
- The arrow member selection operator (->) is preceded by a pointer to an object to access that object's `public` members.
- Header files do contain some portions of the implementation and hints about others. Inline member functions, for example, need to be in a header file, so that when the compiler compiles a client, the client can include the `inline` function definition in place.
- A class's `private` members that are listed in the class definition in the header file are visible to clients, even though the clients may not access the `private` members.
- A utility function (also called a helper function) is a `private` member function that supports the operation of the class's `public` member functions. Utility functions are not intended to be used by clients of a class (but can be used by `friends` of a class).
- Like other functions, constructors can specify default arguments.
- A class's destructor is called implicitly when an object of the class is destroyed.
- The name of the destructor for a class is the tilde (~) character followed by the class name.
- A destructor does not actually release an object's storage; it performs termination housekeeping before the system reclaims an object's memory, so the memory may be reused to hold new objects.
- A destructor receives no parameters and returns no value. A class may have only one destructor.
- If the programmer does not explicitly provide a destructor, the compiler creates an "empty" destructor, so every class has exactly one destructor.
- The order in which constructors and destructors are called depends on the order in which execution enters and leaves the scopes where the objects are instantiated.
- Generally, destructor calls are made in the reverse order of the corresponding constructor calls, but the storage classes of objects can alter the order in which destructors are called.
- A reference to an object is an alias for the name of the object and, hence, may be used on the left side of an assignment statement. In this context, the reference makes a perfectly acceptable lvalue that can receive a value. One way to use this capability (unfortunately!) is to have a `public` member function of a class return a reference to a `private` data member of that class. If the function returns a `const` reference, then the reference cannot be used as a modifiable lvalue.
- The assignment operator (=) can be used to assign an object to another object of the same type. By default, such assignment is performed by memberwise assignment; each member of the object on the right of the assignment operator is assigned individually to the same member in the object on the left of the assignment operator.
- Objects may be passed as function arguments and may be returned from functions. Such passing and returning is performed using pass-by-value; by default, a copy of the object is passed or returned. In such cases, C++ creates a new object and uses a copy constructor to copy the original object's values into the new object. We explain these in detail in [Chapter 11](#), Operator Overloading; String and Array Objects.
- For each class, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object.
- Many substantial class libraries exist, and others are being developed worldwide.
- Software reusability speeds the development of powerful, high-quality software. Rapid applications development (RAD) through the mechanisms of reusable componentry has become an important field.

 PREV

NEXT 

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 519]

## Terminology

abort function

access function

aggregation

arrow member selection operator (->)

assigning class objects

class libraries

class scope

composition

copy constructor

default arguments with constructors

default memberwise assignment

#define preprocessor directive

derive one class from another

destructor

#endif preprocessor directive

exit function

file scope

fill character

handle on an object

helper function

#ifndef preprocessor directive

implicit handle on an object

inheritance

initializer

memberwise assignment

name handle on an object

object handle

object leaves scope

order in which constructors and destructors are called

overloaded constructor

overloaded member function

pass an object by value

pointer handle on an object

predicate function

preprocessor wrapper

pure procedure

rapid application development (RAD)

reentrant code

reference handle on an object

reusable componentry

`setfill` parameterized stream manipulator

software asset

termination housekeeping

tilde character (~) in a destructor name

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 519 (continued)]

## Self-Review Exercises

**9.1** Fill in the blanks in each of the following:

a.

Class members are accessed via the \_\_\_\_\_ operator in conjunction with the name of an object (or reference to an object) of the class or via the \_\_\_\_\_ operator in conjunction with a pointer to an object of the class.

b.

Class members specified as \_\_\_\_\_ are accessible only to member functions of the class and `friends` of the class.

c.

Class members specified as \_\_\_\_\_ are accessible anywhere an object of the class is in scope.

d.

\_\_\_\_\_ can be used to assign an object of a class to another object of the same class.

**9.2** Find the error(s) in each of the following and explain how to correct it (them):

**a.**

Assume the following prototype is declared in class Time:

```
void ~Time(int);
```

**b.**

The following is a partial definition of class Time:

```
class Time
{
public:
 // function prototypes

private:
 int hour = 0;
 int minute = 0;
 int second = 0;
}; // end class Time
```

**c.**

Assume the following prototype is declared in class Employee:

```
int Employee(const char *, const char *);
```

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 520]

## Answers to Self-Review Exercises

**9.1** a) dot (.), arrow (->). b) `private`. c) `public`. d) Default memberwise assignment (performed by the assignment operator).

**9.2**

a.

Error: Destructors are not allowed to return values (or even specify a return type) or take arguments.

Correction: Remove the return type `void` and the parameter `int` from the declaration.

b.

Error: Members cannot be explicitly initialized in the class definition.

Correction: Remove the explicit initialization from the class definition and initialize the data members in a constructor.

c.

Error: Constructors are not allowed to return values.

Correction: Remove the return type `int` from the declaration.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 521]

•

Multiplying two Rational numbers. The result should be stored in reduced form.

•

Dividing two Rational numbers. The result should be stored in reduced form.

•

Printing Rational numbers in the form  $a/b$ , where  $a$  is the numerator and  $b$  is the denominator.

•

Printing Rational numbers in floating-point format.

## 9.7

(Enhancing Class Time) Modify the Time class of [Figs. 9.89.9](#) to include a tick member function that increments the time stored in a Time object by one second. The Time object should always remain in a consistent state. Write a program that tests the tick member function in a loop that prints the time in standard format during each iteration of the loop to illustrate that the tick member function works correctly. Be sure to test the following cases:

a.

Incrementing into the next minute.

b.

Incrementing into the next hour.

c.

Incrementing into the next day (i.e., 11:59:59 PM to 12:00:00 AM).

## 9.8

(Enhancing Class Date) Modify the Date class of [Figs. 9.179.18](#) to perform error checking on the initializer values for data members month, day and year. Also, provide a member function `nextDay` to increment the day by one. The Date object should always remain in a consistent state. Write a program that tests function `nextDay` in a loop that prints the date during each iteration to illustrate that `nextDay` works correctly. Be sure to test the following cases:

a.

Incrementing into the next month.

b.

Incrementing into the next year.

## 9.9

(Combining Class Time and Class Date) Combine the modified Time class of [Exercise 9.7](#) and the modified Date class of [Exercise 9.8](#) into one class called `DateAndTime`. (In [Chapter 12](#), we will discuss inheritance, which will enable us to accomplish this task quickly without modifying the existing class definitions.) Modify the `tick` function to call the `nextday` function if the time increments into the next day. Modify functions `printStandard` and `printUniversal` to output the date and time. Write a program to test the new class `DateAndTime`. Specifically, test incrementing the time into the next day.

## 9.10

(Returning Error Indicators from Class Time's set Functions) Modify the set functions in the Time class of [Figs. 9.89.9](#) to return appropriate error values if an attempt is made to set a data member of an object of class Time to an invalid value. Write a program that tests your new version of class Time. Display error messages when set functions return error values.

## 9.11

(Rectangle Class) Create a class `Rectangle` with attributes `length` and `width`, each of which defaults to 1. Provide member functions that calculate the `perimeter` and the `area` of the rectangle. Also, provide set and get functions for the `length` and `width` attributes. The set functions should verify that `length` and `width` are each floating-point numbers larger than 0.0 and less than 20.0.

## 9.12

(Enhancing Class Rectangle) Create a more sophisticated `Rectangle` class than the one you created in [Exercise 9.11](#). This class stores only the Cartesian coordinates of the four corners of the rectangle. The constructor calls a set function that accepts four sets of coordinates and verifies that each of these is in the first quadrant with no single x- or y-coordinate larger than 20.0. The set function also verifies that the supplied coordinates do, in fact, specify a rectangle. Provide member functions that calculate the `length`,

width, perimeter and area. The length is the larger of the two dimensions. Include a predicate function `square` that determines whether the rectangle is a square.

### 9.13

(Enhancing Class Rectangle) Modify class `Rectangle` from [Exercise 9.12](#) to include a `draw` function that displays the rectangle inside a 25-by-25 box enclosing the portion of the first quadrant in which the rectangle resides. Include a `setFillCharacter` function to specify the character out of which the body of the rectangle will be drawn. Include a `setPerimeterCharacter` function to specify the character that will be used to draw the border of the rectangle. If you feel ambitious, you might include functions to scale the size of the rectangle, rotate it, and move it around within the designated portion of the first quadrant.

---

[Page 522]

### 9.14

(`HugeInteger` Class) Create a class `HugeInteger` that uses a 40-element array of digits to store integers as large as 40 digits each. Provide member functions `input`, `output`, `add` and `subtract`. For comparing `HugeInteger` objects, provide functions `isEqualTo`, `isNotEqualTo`, `isGreater Than`, `isLessThan`, `isGreaterThanOrEqualTo` and `isLessThanOrEqualTo`; each of these is a "predicate" function that simply returns `TRUE` if the relationship holds between the two `HugeIntegers` and returns `false` if the relationship does not hold. Also, provide a predicate function `isZero`. If you feel ambitious, provide member functions `multiply`, `divide` and `modulus`.

### 9.15

(`TicTacToe` Class) Create a class `TicTacToe` that will enable you to write a complete program to play the game of tic-tac-toe. The class contains as private data a 3-by-3 two-dimensional array of integers. The constructor should initialize the empty board to all zeros. Allow two human players. Wherever the first player moves, place a 1 in the specified square. Place a 2 wherever the second player moves. Each move must be to an empty square. After each move, determine whether the game has been won or is a draw. If you feel ambitious, modify your program so that the computer makes the moves for one of the players. Also, allow the player to specify whether he or she wants to go first or second. If you feel exceptionally ambitious, develop a program that will play three-dimensional tic-tac-toe on a 4-by-4-by-4 board.  
[Caution: This is an extremely challenging project that could take many weeks of effort!]

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 524]

## Outline

[10.1 Introduction](#)

[10.2 const \(Constant\) Objects and const Member Functions](#)

[10.3 Composition: Objects as Members of Classes](#)

[10.4 friend Functions and friend Classes](#)

[10.5 Using the this Pointer](#)

[10.6 Dynamic Memory Management with Operators new and delete](#)

[10.7 static Class Members](#)

[10.8 Data Abstraction and Information Hiding](#)

[10.8.1 Example: Array Abstract Data Type](#)

[10.8.2 Example: String Abstract Data Type](#)

[10.8.3 Example: Queue Abstract Data Type](#)

[10.9 Container Classes and Iterators](#)

[10.10 Proxy Classes](#)

[10.11 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

## Answers to Self-Review Exercises

### Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 524 (continued)]

## 10.1. Introduction

In this chapter, we continue our study of classes and data abstraction with several more advanced topics. We use `const` objects and `const` member functions to prevent modifications of objects and enforce the principle of least privilege. We discuss compositiona form of reuse in which a class can have objects of other classes as members. Next, we introduce friendship, which enables a class designer to specify non-member functions that can access class's `non-public` membersa technique that is often used in operator overloading ([Chapter 11](#)) for performance reasons. We discuss a special pointer (called `this`), which is an implicit argument to each of a class's `non-static` member functions that allows those member functions to access the correct object's data members and other `non-static` member functions. We then discuss dynamic memory management and show how to create and destroy objects dynamically with the `new` and `delete` operators. Next, we motivate the need for `static` class members and show how to use `static` data members and member functions in your own classes. Finally, we show how to create a proxy class to hide the implementation details of a class (including its `private` data members) from clients of the class.

Recall that [Chapter 3](#) introduced C++ Standard Library class `string` to represent strings as full-fledged class objects. In this chapter, however, we use the pointer-based strings we introduced in [Chapter 8](#) to help the reader master pointers and prepare for the professional world in which the reader will see a great deal of C legacy code implemented over the last two decades. Thus, the reader will become familiar with the two most prevalent methods of creating and manipulating strings in C++.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 525]

Some objects need to be modifiable and some do not. The programmer may use keyword `const` to specify that an object is not modifiable and that any attempt to modify the object should result in a compilation error. The statement

```
const Time noon(12, 0, 0);
```

declares a `const` object `noon` of class `Time` and initializes it to 12 noon.

#### Software Engineering Observation 10.1



Declaring an object as `const` helps enforce the principle of least privilege. Attempts to modify the object are caught at compile time rather than causing execution-time errors. Using `const` properly is crucial to proper class design, program design and coding.

#### Performance Tip 10.1



Declaring variables and objects `const` can improve performance today's sophisticated optimizing compilers can perform certain optimizations on constants that cannot be performed on variables.

C++ compilers disallow member function calls for `const` objects unless the member functions themselves are also declared `const`. This is true even for get member functions that do not modify the object. In addition, the compiler does not allow member functions declared `const` to modify the object.

A function is specified as `const` both in its prototype ([Fig. 10.1](#); lines 1924) and in its definition ([Fig. 10.2](#); lines 47, 53, 59 and 65) by inserting the keyword `const` after the function's parameter list and, in the case of the function definition, before the left brace that begins the function body.

#### **Figure 10.1. Time class definition with const member functions.**

(This item is displayed on page 526 in the print version)

```

1 // Fig. 10.1: Time.h
2 // Definition of class Time.
3 // Member functions defined in Time.cpp.
4 #ifndef TIME_H
5 #define TIME_H
6
7 class Time
8 {
9 public:
10 Time(int = 0, int = 0, int = 0); // default constructor
11
12 // set functions
13 void setTime(int, int, int); // set time
14 void setHour(int); // set hour
15 void setMinute(int); // set minute
16 void setSecond(int); // set second
17
18 // get functions (normally declared const)
19 int getHour() const; // return hour
20 int getMinute() const; // return minute
21 int getSecond() const; // return second
22
23 // print functions (normally declared const)
24 void printUniversal() const; // print universal time
25 void printStandard(); // print standard time (should be const)
26 private:
27 int hour; // 0 - 23 (24-hour clock format)
28 int minute; // 0 - 59
29 int second; // 0 - 59
30 }; // end class Time
31
32 #endif

```

**Figure 10.2. Time class member-function definitions, including const member functions.**

(This item is displayed on pages 527 - 528 in the print version)

```

1 // Fig. 10.2: Time.cpp
2 // Member-function definitions for class Time.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // include definition of class Time
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables;
14 // default values are 0 (see class definition)
15 Time::Time(int hour, int minute, int second)
16 {
17 setTime(hour, minute, second);
18 } // end Time constructor
19
20 // set hour, minute and second values
21 void Time::setTime(int hour, int minute, int second)
22 {
23 setHour(hour);
24 setMinute(minute);
25 setSecond(second);
26 } // end function setTime
27
28 // set hour value
29 void Time::setHour(int h)
30 {
31 hour = (h >= 0 && h < 24) ? h : 0; // validate hour
32 } // end function setHour
33
34 // set minute value
35 void Time::setMinute(int m)
36 {
37 minute = (m >= 0 && m < 60) ? m : 0; // validate minute
38 } // end function setMinute
39
40 // set second value
41 void Time::setSecond(int s)
42 {
43 second = (s >= 0 && s < 60) ? s : 0; // validate second
44 } // end function setSecond
45
46 // return hour value
47 int Time::getHour() const // get functions should be const
48 {
49 return hour;
50 } // end function getHour

```

```

51 // return minute value
52 int Time::getMinute() const
53 {
54 return minute;
55 } // end function getMinute
56
57
58 // return second value
59 int Time::getSecond() const
60 {
61 return second;
62 } // end function getSecond
63
64 // print Time in universal-time format (HH:MM:SS)
65 void Time::printUniversal() const
66 {
67 cout << setfill('0') << setw(2) << hour << ":"
68 << setw(2) << minute << ":" << setw(2) << second;
69 } // end function printUniversal
70
71 // print Time in standard-time format (HH:MM:SS AM or PM)
72 void Time::printStandard() // note lack of const declaration
73 {
74 cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
75 << ":" << setfill('0') << setw(2) << minute
76 << ":" << setw(2) << second << (hour < 12 ? " AM" : " PM");
77 } // end function printStandard

```

## Common Programming Error 10.1



Defining as `const` a member function that modifies a data member of an object is a compilation error.

## Common Programming Error 10.2



Defining as `const` a member function that calls a non-`const` member function of the class on the same instance of the class is a compilation error.

## Common Programming Error 10.3



Invoking a non-const member function on a `const` object is a compilation error.

## Software Engineering Observation 10.2



A `const` member function can be overloaded with a non-`const` version. The compiler chooses which overloaded member function to use based on the object on which the function is invoked. If the object is `const`, the compiler uses the `const` version. If the object is not `const`, the compiler uses the non-`const` version.

An interesting problem arises for constructors and destructors, each of which typically modifies objects. The `const` declaration is not allowed for constructors and destructors. A constructor must be allowed to modify an object so that the object can be initialized properly. A destructor must be able to perform its termination housekeeping chores before an object's memory is reclaimed by the system.

[Page 526]

## Common Programming Error 10.4



Attempting to declare a constructor or destructor `const` is a compilation error.

## Defining and Using `const` Member Functions

The program of Figs. 10.110.3 modifies class `Time` of Figs. 9.99.10 by making its get functions and `printUniversal` function `const`. In the header file `Time.h` (Fig. 10.1), lines 1921 and 24 now include keyword `const` after each function's parameter list. The corresponding definition of each function in Fig. 10.2 (lines 47, 53, 59 and 65, respectively) also specifies keyword `const` after each function's parameter list.

Figure 10.3 instantiates two `Time` objects non-`const` object `wakeUp` (line 7) and `const` object `noon` (line 8). The program attempts to invoke non-`const` member functions `setHour` (line 13) and `printStandard` (line 20) on the `const` object `noon`. In each case, the compiler generates an error message. The program also illustrates the three other member-function-call combinations on objects a non-`const` member function on a non-`const` object (line 11), a `const` member function on a non-`const` object (line 15) and a `const` member function on a `const` object (lines 1718). The error messages generated for non-`const` member functions called on a `const` object are shown in the output window. Notice that, although some current compilers issue only warning messages for lines 13 and 20 (thus allowing this program to be executed), we consider these warnings to be errors the ANSI/ISO C++ standard disallows the invocation of a non-`const` member function

on a `const` object.

[Page 527]

### Figure 10.3. `const` objects and `const` member functions.

(This item is displayed on pages 528 - 529 in the print version)

```

1 // Fig. 10.3: fig10_03.cpp
2 // Attempting to access a const object with non-const member functions.
3 #include "Time.h" // include Time class definition
4
5 int main()
6 {
7 Time wakeUp(6, 45, 0); // non-constant object
8 const Time noon(12, 0, 0); // constant object
9
10 // OBJECT MEMBER FUNCTION
11 wakeUp.setHour(18); // non-const non-const
12
13 noon.setHour(12); // const non-const
14
15 wakeUp.getHour(); // non-const const
16
17 noon.getMinute(); // const const
18 noon.printUniversal(); // const const
19
20 noon.printStandard(); // const non-const
21 return 0;
22 } // end main

```

Borland C++ command-line compiler error messages:

```

Warning W8037 fig10_03.cpp 13: Non-const function Time::setHour(int)
 called for const object in function main()
Warning W8037 fig10_03.cpp 20: Non-const function Time::printStandard()
 called for const object in function main()

```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphttp5_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
'Time &
 Conversion loses qualifiers
C:\cpphttp5_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to
'Time &
 Conversion loses qualifiers
```

GNU C++ compiler error messages:

```
fig10_03.cpp:13: error: passing `const Time' as `this' argument of
`void Time::setHour(int)' discards qualifiers
fig10_03.cpp:20: error: passing `const Time' as `this' argument of
`void Time::printStandard()' discards qualifiers
```

[Page 529]

Notice that even though a constructor must be a non-const member function ([Fig. 10.2](#), lines 1518), it can still be used to initialize a `const` object ([Fig. 10.3](#), line 8). The definition of the `Time` constructor ([Fig. 10.2](#), lines 1518) shows that the `Time` constructor calls another non-const member function `setTime` (lines 2126) to perform the initialization of a `Time` object. Invoking a non-const member function from the constructor call as part of the initialization of a `const` object is allowed. The "const ness" of a `const` object is enforced from the time the constructor completes initialization of the object until that object's destructor is called.

Also notice that line 20 in [Fig. 10.3](#) generates a compilation error even though member function `printStandard` of class `Time` does not modify the object on which it is invoked. The fact that a member function does not modify an object is not sufficient to indicate that the function is constant function the function must explicitly be declared `const`.

## Initializing a `const` Data Member with a Member Initializer

The program of [Figs. 10.410.6](#) introduces using **member initializer syntax**. All data members can be initialized using member initializer syntax, but `const` data members and data members that are references must be initialized using member initializers. Later in this chapter, we will see that member objects must be initialized this way as well. In [Chapter 12](#) when we study inheritance, we will see that base-class portions of derived classes also must be initialized this way.

The constructor definition ([Fig. 10.5](#), lines 1116) uses a **member initializer list** to initialize class Increment's data members non-const integer `count` and const integer `increment` (declared in lines 1920 of [Fig. 10.4](#)). Member initializers appear between a constructor's parameter list and the left brace that begins the constructor's body. The member initializer list ([Fig. 10.5](#), lines 1213) is separated from the parameter list with a colon (:). Each member initializer consists of the data member name followed by parentheses containing the member's initial value. In this example, `count` is initialized with the value of constructor parameter `c` and `increment` is initialized with the value of constructor parameter `i`. Note that multiple member initializers are separated by commas. Also, note that the member initializer list executes before the body of the constructor executes.

**Figure 10.4. Increment class definition containing non-const data member count and const data member increment.**

(This item is displayed on page 530 in the print version)

```

1 // Fig. 10.4: Increment.h
2 // Definition of class Increment.
3 #ifndef INCREMENT_H
4 #define INCREMENT_H
5
6 class Increment
7 {
8 public:
9 Increment(int c = 0, int i = 1); // default constructor
10
11 // function addIncrement definition
12 void addIncrement()
13 {
14 count += increment;
15 } // end function addIncrement
16
17 void print() const; // prints count and increment
18 private:
19 int count;
20 const int increment; // const data member
21 }; // end class Increment
22
23 #endif

```

**Figure 10.5. Member initializer used to initialize a constant of a built-in data type.**

(This item is displayed on page 530 in the print version)

```

1 // Fig. 10.5: Increment.cpp
2 // Member-function definitions for class Increment demonstrate using a
3 // member initializer to initialize a constant of a built-in data type.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Increment.h" // include definition of class Increment
9
10 // constructor
11 Increment::Increment(int c, int i)
12 : count(c), // initializer for non-const member
13 increment(i) // required initializer for const member
14 {
15 // empty body
16 } // end constructor Increment
17
18 // print count and increment values
19 void Increment::print() const
20 {
21 cout << "count = " << count << ", increment = " << increment << endl;
22 } // end function print

```

**Figure 10.6. Invoking an Increment object's print and addIncrement member functions.**

```

1 // Fig. 10.6: fig10_06.cpp
2 // Program to test class Increment.
3 #include <iostream>
4 using std::cout;
5
6 #include "Increment.h" // include definition of class Increment
7
8 int main()
9 {
10 Increment value(10, 5);
11
12 cout << "Before incrementing: ";
13 value.print();
14
15 for (int j = 1; j <= 3; j++)
16 {
17 value.addIncrement();
18 cout << "After increment " << j << ":" ;
19 value.print();

```

```

20 } // end for
21
22 return 0;
23 } // end main

```

Before incrementing: count = 10, increment = 5  
After increment 1: count = 15, increment = 5  
After increment 2: count = 20, increment = 5  
After increment 3: count = 25, increment = 5

### Software Engineering Observation 10.3



A `const` object cannot be modified by assignment, so it must be initialized. When a data member of a class is declared `const`, a member initializer must be used to provide the constructor with the initial value of the data member for an object of the class. The same is true for references.

---

[Page 532]

### Erroneously Attempting to Initialize a `const` Data Member with an Assignment

The program of Figs. 10.710.9 illustrates the compilation errors caused by attempting to initialize `const` data member `increment` with an assignment statement (Fig. 10.8, line 14) in the `Increment` constructor's body rather than with a member initializer. Note that line 13 of Fig. 10.8 does not generate a compilation error, because `count` is not declared `const`. Also note that the compilation errors produced by Microsoft Visual C++ .NET refer to `int` data member `increment` as a "const object." The ANSI/ISO C++ standard defines an "object" as any "region of storage." Like instances of classes, fundamental-type variables also occupy space in memory, so they are often referred to as "objects."

**Figure 10.7. Increment class definition containing non-`const` data member `count` and `const` data member `increment`.**

(This item is displayed on pages 532 - 533 in the print version)

```

1 // Fig. 10.7: Increment.h
2 // Definition of class Increment.
3 #ifndef INCREMENT_H
4 #define INCREMENT_H
5
6 class Increment
7 {
8 public:
9 Increment(int c = 0, int i = 1); // default constructor
10
11 // function addIncrement definition
12 void addIncrement()
13 {
14 count += increment;
15 } // end function addIncrement
16
17 void print() const; // prints count and increment
18 private:
19 int count;
20 const int increment; // const data member
21 }; // end class Increment
22
23 #endif

```

**Figure 10.8. Erroneous attempt to initialize a constant of a built-in data type by assignment.**

(This item is displayed on page 533 in the print version)

```

1 // Fig. 10.8: Increment.cpp
2 // Attempting to initialize a constant of
3 // a built-in data type with an assignment.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Increment.h" // include definition of class Increment
9
10 // constructor; constant member 'increment' is not initialized
11 Increment::Increment(int c, int i)
12 {
13 count = c; // allowed because count is not constant
14 increment = i; // ERROR: Cannot modify a const object
15 } // end constructor Increment
16
17 // print count and increment values
18 void Increment::print() const

```

```

19 {
20 cout << "count = " << count << ", increment = " << increment << endl;
21 } // end function print

```

## Common Programming Error 10.5



Not providing a member initializer for a `const` data member is a compilation error.

## Software Engineering Observation 10.4



Constant data members (`const` objects and `const` variables) and data members declared as references must be initialized with member initializer syntax; assignments for these types of data in the constructor body are not allowed.

Note that function `print` (Fig. 10.8, lines 1821) is declared `const`. It might seem strange to label this function `const`, because a program probably will never have a `const Increment` object. However, it is possible that a program will have a `const` reference to an `Increment` object or a pointer to `const` that points to an `Increment` object. Typically, this occurs when objects of class `Increment` are passed to functions or returned from functions. In these cases, only the `const` member functions of class `Increment` can be called through the reference or pointer. Thus, it is reasonable to declare function `print` as `const`; doing so prevents errors in these situations where an `Increment` object is treated as a `const` object.

## Error-Prevention Tip 10.1



Declare as `const` all of a class's member functions that do not modify the object in which they operate. Occasionally this may seem inappropriate, because you will have no intention of creating `const` objects of that class or accessing objects of that class through `const` references or pointers to `const`. Declaring such member functions `const` does offer a benefit, though. If the member function is inadvertently written to modify the object, the compiler will issue an error message.

---

[Page 533]

**Figure 10.9. Program to test class Increment generates compilation errors.**

(This item is displayed on pages 533 - 534 in the print version)

```

1 // Fig. 10.9: fig10_09.cpp
2 // Program to test class Increment.
3 #include <iostream>
4 using std::cout;
5
6 #include "Increment.h" // include definition of class Increment
7
8 int main()
9 {
10 Increment value(10, 5);
11
12 cout << "Before incrementing: ";
13 value.print();
14
15 for (int j = 1; j <= 3; j++)
16 {
17 value.addIncrement();
18 cout << "After increment " << j << ":" ;
19 value.print();
20 } // end for
21
22 return 0;
23 } // end main

```

Borland C++ command-line compiler error message:

```
Error E2024 Increment.cpp 14: Cannot modify a const object in function
Increment::Increment(int,int)
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphttp5_examples\ch10\Fig10_07_09\Increment.cpp(12) : error C2758:
'Increment::increment' : must be initialized in constructor
base/member initializer list
C:\cpphttp5_examples\ch10\Fig10_07_09\Increment.h(20) :
 see declaration of 'Increment::increment'
C:\cpphttp5_examples\ch10\Fig10_07_09\Increment.cpp(14) : error C2166:
l-value specifies const object
```

## GNU C++ compiler error messages:

```
Increment.cpp:12: error: uninitialized member 'Increment::increment' with
 'const' type 'const int'
Increment.cpp:14: error: assignment of read-only data-member
 `Increment::increment'
```



page footer



The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 535]

The program of Figs. 10.10–10.14 uses class Date (Figs. 10.10–10.11) and class Employee (Figs. 10.12–10.13) to demonstrate objects as members of other objects. The definition of class Employee (Fig. 10.12) contains private data members firstName, lastName, birthDate and hireDate. Members birthDate and hireDate are const objects of class Date, which contains private data members month, day and year. The Employee constructor's header (Fig. 10.13, lines 1821) specifies that the constructor receives four parameters (first, last, dateOfBirth and dateOfHire). The first two parameters are used in the constructor's body to initialize the character arrays firstName and lastName. The last two parameters are passed via member initializers to the constructor for class Date. The colon (:) in the header separates the member initializers from the parameter list. The member initializers specify the Employee constructor parameters being passed to the constructors of the member Date objects. Parameter dateOfBirth is passed to object birthDate's constructor (Fig. 10.13, line 20), and parameter dateOfHire is passed to object hireDate's constructor (Fig. 10.13, line 21). Again, member initializers are separated by commas. As you study class Date (Fig. 10.10), notice that the class does not provide a constructor that receives a parameter of type Date. So, how is the member initializer list in class Employee's constructor able to initialize the birthDate and hireDate objects by passing Date object's to their Date constructors? As we mentioned in Chapter 9, the compiler provides each class with a default copy constructor that copies each member of the constructor's argument object into the corresponding member of the object being initialized. Chapter 11 discusses how programmers can define customized copy constructors.

**Figure 10.10. Date class definition.**

```

1 // Fig. 10.10: Date.h
2 // Date class definition; Member functions defined in Date.cpp
3 #ifndef DATE_H
4 #define DATE_H
5
6 class Date
7 {
8 public:
9 Date(int = 1, int = 1, int = 1900); // default constructor
10 void print() const; // print date in month/day/year format
11 ~Date(); // provided to confirm destruction order
12 private:
13 int month; // 1-12 (January-December)
14 int day; // 1-31 based on month
15 int year; // any year
16
17 // utility function to check if day is proper for month and year

```

```

18 int checkDay(int) const;
19 } // end class Date
20
21 #endif

```

**Figure 10.11. Date class member-function definitions.**

(This item is displayed on pages 536 - 537 in the print version)

```

1 // Fig. 10.11: Date.cpp
2 // Member-function definitions for class Date.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Date.h" // include Date class definition
8
9 // constructor confirms proper value for month; calls
10 // utility function checkDay to confirm proper value for day
11 Date::Date(int mn, int dy, int yr)
12 {
13 if (mn > 0 && mn <= 12) // validate the month
14 month = mn;
15 else
16 {
17 month = 1; // invalid month set to 1
18 cout << "Invalid month (" << mn << ") set to 1.\n";
19 } // end else
20
21 year = yr; // could validate yr
22 day = checkDay(dy); // validate the day
23
24 // output Date object to show when its constructor is called
25 cout << "Date object constructor for date ";
26 print();
27 cout << endl;
28 } // end Date constructor
29
30 // print Date object in form month/day/year
31 void Date::print() const
32 {
33 cout << month << '/' << day << '/' << year;
34 } // end function print
35
36 // output Date object to show when its destructor is called

```

```

37 Date::~Date()
38 {
39 cout << "Date object destructor for date ";
40 print();
41 cout << endl;
42 } // end ~Date destructor
43
44 // utility function to confirm proper day value based on
45 // month and year; handles leap years, too
46 int Date::checkDay(int testDay) const
47 {
48 static const int daysPerMonth[13] =
49 { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
50
51 // determine whether testDay is valid for specified month
52 if (testDay > 0 && testDay <= daysPerMonth[month])
53 return testDay;
54
55 // February 29 check for leap year
56 if (month == 2 && testDay == 29 && (year % 400 == 0 ||
57 (year % 4 == 0 && year % 100 != 0)))
58 return testDay;
59
60 cout << "Invalid day (" << testDay << ") set to 1.\n";
61 return 1; // leave object in consistent state if bad value
62 } // end function checkDay

```

**Figure 10.12. Employee class definition showing composition.**

(This item is displayed on page 537 in the print version)

```

1 // Fig. 10.12: Employee.h
2 // Employee class definition.
3 // Member functions defined in Employee.cpp.
4 #ifndef EMPLOYEE_H
5 #define EMPLOYEE_H
6
7 #include "Date.h" // include Date class definition
8
9 class Employee
10 {
11 public:
12 Employee(const char * const, const char * const,
13 const Date &, const Date &);

```

```

14 void print() const;
15 ~Employee(); // provided to confirm destruction order
16 private:
17 char firstName[25];
18 char lastName[25];
19 const Date birthDate; // composition: member object
20 const Date hireDate; // composition: member object
21 }; // end class Employee
22
23 #endif

```

**Figure 10.13. Employee class member-function definitions, including constructor with a member initializer list.**

(This item is displayed on pages 538 - 539 in the print version)

```

1 // Fig. 10.13: Employee.cpp
2 // Member-function definitions for class Employee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // strlen and strncpy prototypes
8 using std::strlen;
9 using std::strncpy;
10
11 #include "Employee.h" // Employee class definition
12 #include "Date.h" // Date class definition
13
14 // constructor uses member initializer list to pass initializer
15 // values to constructors of member objects birthDate and hireDate
16 // [Note: This invokes the so-called "default copy constructor" which the
17 // C++ compiler provides implicitly.]
18 Employee::Employee(const char * const first, const char * const last,
19 const Date &dateOfBirth, const Date &dateOfHire)
20 : birthDate(dateOfBirth), // initialize birthDate
21 hireDate(dateOfHire) // initialize hireDate
22 {
23 // copy first into firstName and be sure that it fits
24 int length = strlen(first);
25 length = (length < 25 ? length : 24);
26 strncpy(firstName, first, length);
27 firstName[length] = '\0';
28
29 // copy last into lastName and be sure that it fits

```

```

30 length = strlen(last);
31 length = (length < 25 ? length : 24);
32 strncpy(lastName, last, length);
33 lastName[length] = '\0';
34
35 // output Employee object to show when constructor is called
36 cout << "Employee object constructor: "
37 << firstName << ' ' << lastName << endl;
38 } // end Employee constructor
39
40 // print Employee object
41 void Employee::print() const
42 {
43 cout << lastName << ", " << firstName << " Hired: ";
44 hireDate.print();
45 cout << " Birthday: ";
46 birthDate.print();
47 cout << endl;
48 } // end function print
49
50 // output Employee object to show when its destructor is called
51 Employee::~Employee()
52 {
53 cout << "Employee object destructor: "
54 << lastName << ", " << firstName << endl;
55 } // end ~Employee destructor

```

---

[Page 536]

**Figure 10.14** creates two `Date` objects (lines 1112) and passes them as arguments to the constructor of the `Employee` object created in line 13. Line 16 outputs the `Employee` object's data. When each `Date` object is created in lines 1112, the `Date` constructor defined at lines 1128 of Fig. 10.11 displays a line of output to show that the constructor was called (see the first two lines of the sample output). [Note: Line 13 of Fig. 10.14 causes two additional `Date` constructor calls that do not appear in the program's output. When each of the `Employee`'s `Date` member object's is initialized in the `Employee` constructor's member initializer list, the default copy constructor for class `Date` is called. This constructor is defined implicitly by the compiler and does not contain any output statements to demonstrate when it is called. We discuss copy constructors and default copy constructors in detail in Chapter 11.]

---

[Page 539]

### Figure 10.14. Member-object initializers.

```

1 // Fig. 10.14: fig10_14.cpp
2 // Demonstrating composition--an object with member objects.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Employee.h" // Employee class definition
8
9 int main()
10 {
11 Date birth(7, 24, 1949);
12 Date hire(3, 12, 1988);
13 Employee manager("Bob", "Blue", birth, hire);
14
15 cout << endl;
16 manager.print();
17
18 cout << "\nTest Date constructor with invalid values:\n";
19 Date lastDayOff(14, 35, 1994); // invalid month and day
20 cout << endl;
21 return 0;
22 } // end main

```

```

Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob Hired: 3/12/1988 Birthday: 7/24/1949

Test Date constructor with invalid values:
Invalid month (14) set to 1.
Invalid day (35) set to 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949

```

Class `Date` and class `Employee` each include a destructor (lines 3742 of Fig. 10.11 and lines 5155 of Fig. 10.13, respectively) that prints a message when an object of its class is destroyed. This enables us to confirm in the program output that objects are constructed from the inside out and destroyed in the reverse order from the outside in (i.e., the `Date` member objects are destroyed after the `Employee` object that contains them). Notice the last four lines in the output of Fig. 10.14. The last two lines are the outputs of the `Date` destructor running on `Date` objects `hire` (line 12) and `birth` (line 11), respectively. These outputs confirm that the three objects created in `main` are destructed in the reverse of the order in which they were constructed. (The `Employee` destructor output is five lines from the bottom.) The fourth and third lines from the bottom of the output window show the destructors running for the `Employee`'s member objects `hireDate` (Fig. 10.12, line 20) and `birthDate` (Fig. 10.12, line 19). These outputs confirm that the `Employee` object is destructed from the outside in.i.e., the `Employee` destructor runs first (output shown five lines from the bottom of the output window), then the member objects are destructed in the reverse order from which they were constructed. Again, the outputs in Fig. 10.14 did not show the constructors running for these objects, because these were the default copy constructors provided by the C++ compiler.

[Page 540]

A member object does not need to be initialized explicitly through a member initializer. If a member initializer is not provided, the member object's default constructor will be called implicitly. Values, if any, established by the default constructor can be overridden by set functions. However, for complex initialization, this approach may require significant additional work and time.

### Common Programming Error 10.6



A compilation error occurs if a member object is not initialized with a member initializer and the member object's class does not provide a default constructor (i.e., the member object's class defines one or more constructors, but none is a default constructor).

### Performance Tip 10.2



Initialize member objects explicitly through member initializers. This eliminates the overhead of "doubly initializing" member objects once when the member object's default constructor is called and again when set functions are called in the constructor body (or later) to initialize the member object.

### Software Engineering Observation 10.6



If a class member is an object of another class, making that member object public does not violate the encapsulation and hiding of that member object's private members. However, it does violate the encapsulation and hiding of the containing class's implementation, so member objects of class types should still be private, like all other data members.

In line 26 of Fig. 10.11, notice the call to Date member function `print`. Many member functions of classes in C++ require no arguments. This is because each member function contains an implicit handle (in the form of a pointer) to the object on which it operates. We discuss the implicit pointer, which is represented by keyword `this`, in Section 10.5.

Class `Employee` uses two 25-character arrays (Fig. 10.12, lines 1718) to represent the first name and last name of the `Employee`. These arrays may waste space for names shorter than 24 characters. (Remember, one character in each array is for the terminating null character, '\0', of the string.) Also, names longer than 24 characters must be truncated to fit in these fixed-size character arrays. Section 10.7 presents another version of class `Employee` that dynamically creates the exact amount of space required to hold the first and the last name.

[Page 541]

Note that the simplest way to represent an `Employee`'s first and last name using the exact amount of space required is to use two `string` objects (C++ Standard Library class `string` was introduced in Chapter 3). If we did this, the `Employee` constructor would appear as follows

```
Employee::Employee(const string &first, const string &last,
 const Date &dateOfBirth, const Date &dateOfHire)
 : firstName(first), // initialize firstName
 lastName(last), // initialize lastName
 birthDate(dateOfBirth), // initialize birthDate
 hireDate(dateOfHire) // initialize hireDate
{
 // output Employee object to show when constructor is called
 cout << "Employee object constructor: "
 << firstName << ' ' << lastName << endl;
} // end Employee constructor
```

Notice that data members `firstName` and `lastName` (now `string` objects) are initialized through member initializers. The `Employee` classes presented in Chapters 1213 use `string` objects in this fashion. In this chapter, we use pointer-based strings to provide the reader with additional exposure to pointer manipulation.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 542]

## Good Programming Practice 10.1



Place all friendship declarations first inside the class definition's body and do not precede them with any access specifier.

Friendship is granted, not taken. i.e., for class B to be a friend of class A, class A must explicitly declare that class B is its friend. Also, the friendship relation is neither symmetric nor transitive; i.e., if class A is a friend of class B, and class B is a friend of class C, you cannot infer that class B is a friend of class A (again, friendship is not symmetric), that class C is a friend of class B (also because friendship is not symmetric), or that class A is a friend of class C (friendship is not transitive).

## Software Engineering Observation 10.9



Some people in the OOP community feel that "friendship" corrupts information hiding and weakens the value of the object-oriented design approach. In this text, we identify several examples of the responsible use of friendship.

## Modifying a Class's private Data With a Friend Function

[Figure 10.15](#) is a mechanical example in which we define `friend` function `setX` to set the `private` data member `x` of class `Count`. Note that the `friend` declaration (line 10) appears first (by convention) in the class definition, even before `public` member functions are declared. Again, this `friend` declaration can appear anywhere in the class.

**Figure 10.15. Friends can access private members of a class.**

(This item is displayed on pages 542 - 543 in the print version)

```

1 // Fig. 10.15: fig10_15.cpp
2 // Friends can access private members of a class.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // Count class definition
8 class Count
9 {
10 friend void setX(Count &, int); // friend declaration
11 public:
12 // constructor
13 Count()
14 : x(0) // initialize x to 0
15 {
16 // empty body
17 } // end constructor Count
18
19 // output x
20 void print() const
21 {
22 cout << x << endl;
23 } // end function print
24 private:
25 int x; // data member
26 }; // end class Count
27
28 // function setX can modify private data of Count
29 // because setX is declared as a friend of Count (line 10)
30 void setX(Count &c, int val)
31 {
32 c.x = val; // allowed because setX is a friend of Count
33 } // end function setX
34
35 int main()
36 {
37 Count counter; // create Count object
38
39 cout << "counter.x after instantiation: ";
40 counter.print();
41
42 setX(counter, 8); // set x using a friend function
43 cout << "counter.x after call to setX friend function: ";
44 counter.print();
45 return 0;
46 } // end main

```

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

[Page 543]

Function `setX` (lines 3033) is a C-style, stand-alone function it is not a member function of class `Count`. For this reason, when `setX` is invoked for object `counter`, line 42 passes `counter` as an argument to `setX` rather than using a handle (such as the name of the object) to call the function, as in

```
counter.setX(8);
```

As we mentioned, Fig. 10.15 is a mechanical example of using the `friend` construct. It would normally be appropriate to define function `setX` as a member function of class `Count`. It would also normally be appropriate to separate the program of Fig. 10.15 into three files:

**1.**

A header file (e.g., `Count.h`) containing the `Count` class definition, which in turn contains the prototype of `friend` function `setX`

**2.**

An implementation file (e.g., `Count.cpp`) containing the definitions of class `Count`'s member functions and the definition of `friend` function `setX`

**3.**

A test program (e.g., `fig10_15.cpp`) with `main`

## Erroneously Attempting to Modify a private Member with a Non-friend Function

The program of Fig. 10.16 demonstrates the error messages produced by the compiler when non-`friend` function `cannotSetX` (lines 2932) is called to modify `private` data member `x`.

**Figure 10.16. Non-friend/nonmember functions cannot access private members.**

(This item is displayed on pages 544 - 545 in the print version)

```

1 // Fig. 10.16: fig10_16.cpp
2 // Non-friend/non-member functions cannot access private data of a class.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // Count class definition (note that there is no friendship declaration)
8 class Count
9 {
10 public:
11 // constructor
12 Count()
13 : x(0) // initialize x to 0
14 {
15 // empty body
16 } // end constructor Count
17
18 // output x
19 void print() const
20 {
21 cout << x << endl;
22 } // end function print
23 private:
24 int x; // data member
25 }; // end class Count
26
27 // function cannotSetX tries to modify private data of Count,
28 // but cannot because the function is not a friend of Count
29 void cannotSetX(Count &c, int val)
30 {
31 c.x = val; // ERROR: cannot access private member in Count
32 } // end function cannotSetX
33
34 int main()
35 {
36 Count counter; // create Count object
37
38 cannotSetX(counter, 3); // cannotSetX is not a friend
39 return 0;
40 } // end main

```

Borland C++ command-line compiler error message:

```
Error E2247 Fig10_16/fig10_16.cpp 31: 'Count::x' is not accessible in
function cannotSetX(Count &,int)
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphttp5_examples\ch10\Fig10_16\fig10_16.cpp(31) : error C2248: 'Count::x'
: cannot access private member declared in class 'Count'
C:\cpphttp5_examples\ch10\Fig10_16\fig10_16.cpp(24) : see declaration
of 'Count::x'
C:\cpphttp5_examples\ch10\Fig10_16\fig10_16.cpp(9) : see declaration
of 'Count'
```

GNU C++ compiler error messages:

```
fig10_16.cpp:24: error: `int Count::x' is private
fig10_16.cpp:31: error: within this context
```

It is possible to specify overloaded functions as friends of a class. Each overloaded function intended to be a friend must be explicitly declared in the class definition as a friend of the class.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 546]

Note the parentheses around `*this` (line 36) when used with the dot member selection operator (`.`). The parentheses are required because the dot operator has higher precedence than the `*` operator. Without the parentheses, the expression `*this.x` would be evaluated as if it were parenthesized as `* (this.x)`, which is a compilation error, because the dot operator cannot be used with a pointer.

One interesting use of the `this` pointer is to prevent an object from being assigned to itself. As we will see in [Chapter 11](#), self-assignment can cause serious errors when the object contains pointers to dynamically allocated storage.

[Page 547]

#### Common Programming Error 10.7



Attempting to use the member selection operator (`.`) with a pointer to an object is a compilation error. The dot member selection operator may be used only with an lvalue such as an object's name, a reference to an object or a dereferenced pointer to an object.

### Using the `this` Pointer to Enable Cascaded Function Calls

Another use of the `this` pointer is to enable **cascaded member-function calls** in which multiple functions are invoked in the same statement (as in line 14 of [Fig. 10.20](#)). The program of [Figs. 10.18](#) and [10.20](#) modifies class `Time`'s set functions `setTime`, `setHour`, `setMinute` and `setSecond` such that each returns a reference to a `Time` object to enable cascaded member-function calls. Notice in [Fig. 10.19](#) that the last statement in the body of each of these member functions returns `*this` (lines 26, 33, 40 and 47) into a return type of `Time &`.

**Figure 10.18. Time class definition modified to enable cascaded member-function calls.**

(This item is displayed on pages 547 - 548 in the print version)

```

1 // Fig. 10.18: Time.h
2 // Cascading member function calls.
3
4 // Time class definition.
5 // Member functions defined in Time.cpp.
6 #ifndef TIME_H
7 #define TIME_H
8
9 class Time
10 {
11 public:
12 Time(int = 0, int = 0, int = 0); // default constructor
13
14 // set functions (the Time & return types enable cascading)
15 Time &setTime(int, int, int); // set hour, minute, second
16 Time &setHour(int); // set hour
17 Time &setMinute(int); // set minute
18 Time &setSecond(int); // set second
19
20 // get functions (normally declared const)
21 int getHour() const; // return hour
22 int getMinute() const; // return minute
23 int getSecond() const; // return second
24
25 // print functions (normally declared const)
26 void printUniversal() const; // print universal time
27 void printStandard() const; // print standard time
28 private:
29 int hour; // 0 - 23 (24-hour clock format)
30 int minute; // 0 - 59
31 int second; // 0 - 59
32 }; // end class Time
33
34 #endif

```

**Figure 10.19. Time class member-function definitions modified to enable cascaded member-function calls.**

(This item is displayed on pages 548 - 549 in the print version)

```

1 // Fig. 10.19: Time.cpp
2 // Member-function definitions for Time class.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // Time class definition
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables;
14 // default values are 0 (see class definition)
15 Time::Time(int hr, int min, int sec)
16 {
17 setTime(hr, min, sec);
18 } // end Time constructor
19
20 // set values of hour, minute, and second
21 Time &Time::setTime(int h, int m, int s) // note Time & return
22 {
23 setHour(h);
24 setMinute(m);
25 setSecond(s);
26 return *this; // enables cascading
27 } // end function setTime
28
29 // set hour value
30 Time &Time::setHour(int h) // note Time & return
31 {
32 hour = (h >= 0 && h < 24) ? h : 0; // validate hour
33 return *this; // enables cascading
34 } // end function setHour
35
36 // set minute value
37 Time &Time::setMinute(int m) // note Time & return
38 {
39 minute = (m >= 0 && m < 60) ? m : 0; // validate minute
40 return *this; // enables cascading
41 } // end function setMinute
42
43 // set second value
44 Time &Time::setSecond(int s) // note Time & return
45 {
46 second = (s >= 0 && s < 60) ? s : 0; // validate second
47 return *this; // enables cascading
48 } // end function setSecond

```

```
49
50 // get hour value
51 int Time::getHour() const
52 {
53 return hour;
54 } // end function getHour
55
56 // get minute value
57 int Time::getMinute() const
58 {
59 return minute;
60 } // end function getMinute
61
62 // get second value
63 int Time::getSecond() const
64 {
65 return second;
66 } // end function getSecond
67
68 // print Time in universal-time format (HH:MM:SS)
69 void Time::printUniversal() const
70 {
71 cout << setfill('0') << setw(2) << hour << ":"
72 << setw(2) << minute << ":" << setw(2) << second;
73 } // end function printUniversal
74
75 // print Time in standard-time format (HH:MM:SS AM or PM)
76 void Time::printStandard() const
77 {
78 cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
79 << ":" << setfill('0') << setw(2) << minute
80 << ":" << setw(2) << second << (hour < 12 ? " AM" : " PM");
81 } // end function printStandard
```

**Figure 10.20. Cascading member-function calls.**

(This item is displayed on page 550 in the print version)

```

1 // Fig. 10.20: fig10_20.cpp
2 // Cascading member function calls with the this pointer.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Time.h" // Time class definition
8
9 int main()
10 {
11 Time t; // create Time object
12
13 // cascaded function calls
14 t.setHour(18).setMinute(30).setSecond(22);
15
16 // output time in universal and standard formats
17 cout << "Universal time: ";
18 t.printUniversal();
19
20 cout << "\nStandard time: ";
21 t.printStandard();
22
23 cout << "\n\nNew standard time: ";
24
25 // cascaded function calls
26 t.setTime(20, 20, 20).printStandard();
27 cout << endl;
28 return 0;
29 } // end main

```

```

Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM

```

The program of Fig. 10.20 creates `Time` object `t` (line 11), then uses it in cascaded member-function calls (lines 14 and 26). Why does the technique of returning `*this` as a reference work? The dot operator (`.`) associates from left to right, so line 14 first evaluates `t.setHour( 18 )` then returns a reference to object `t` as the value of this function call. The remaining expression is then interpreted as

```
t.setMinute(30).setSecond(22);
```

The `t.setMinute( 30 )` call executes and returns a reference to the object `t`. The remaining expression is interpreted as

```
t.setSecond(22);
```

Line 26 also uses cascading. The calls must appear in the order shown in line 26, because `printStandard` as defined in the class does not return a reference to `t`. Placing the call to `printStandard` before the call to `setTime` in line 26 results in a compilation error. Chapter 11 presents several practical examples of using cascaded function calls. One such example uses multiple `<<` operators with `cout` to output multiple values in a single statement.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 551]

Again, we present the modified `Employee` class as described here in the example of [Section 10.7](#). First, we present the details of using the `new` and `delete` operators to dynamically allocate memory to store objects, fundamental types and arrays.

Consider the following declaration and statement:

```
Time *timePtr;
timePtr = new Time;
```

The `new` operator allocates storage of the proper size for an object of type `Time`, calls the default constructor to initialize the object and returns a pointer of the type specified to the right of the `new` operator (i.e., a `Time *`). Note that `new` can be used to dynamically allocate any fundamental type (such as `int` or `double`) or class type. If `new` is unable to find sufficient space in memory for the object, it indicates that an error occurred by "throwing an exception." [Chapter 16](#), Exception Handling, discusses how to deal with `new` failures in the context of the ANSI/ISO C++ standard. In particular, we will show how to "catch" the exception thrown by `new` and deal with it. When a program does not "catch" an exception, the program terminates immediately. [Note: The `new` operator returns a 0 pointer in versions of C++ prior to the ANSI/ISO standard. We use the standard version of operator `new` throughout this book.]

To destroy a dynamically allocated object and free the space for the object, use the `delete` operator as follows:

```
delete timePtr;
```

This statement first calls the destructor for the object to which `timePtr` points, then deallocates the memory associated with the object. After the preceding statement, the memory can be reused by the system to allocate other objects.

#### Common Programming Error 10.8



Not releasing dynamically allocated memory when it is no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "**memory leak**."

C++ allows you to provide an [initializer](#) for a newly created fundamental-type variable, as in

```
double *ptr = new double(3.14159);
```

which initializes a newly created `double` to 3.14159 and assigns the resulting pointer to `ptr`. The same syntax can be used to specify a comma-separated list of arguments to the constructor of an object. For example,

[Page 552]

```
Time *timePtr = new Time(12, 45, 0);
```

initializes a newly created `Time` object to 12:45 PM and assigns the resulting pointer to `timePtr`.

As discussed earlier, the `new` operator can be used to allocate arrays dynamically. For example, a 10-element integer array can be allocated and assigned to `gradesArray` as follows:

```
int *gradesArray = new int[10];
```

which declares pointer `gradesArray` and assigns it a pointer to the first element of a dynamically allocated 10-element array of integers. Recall that the size of an array created at compile time must be specified using a constant integral expression. However, the size of a dynamically allocated array can be specified using any integral expression that can be evaluated at execution time. Also note that, when allocating an array of objects dynamically, the programmer cannot pass arguments to each object's constructor. Instead, each object in the array is initialized by its default constructor. To delete the dynamically allocated array to which `gradesArray` points, use the statement

```
delete [] gradesArray;
```

The preceding statement deallocates the array to which `gradesArray` points. If the pointer in the preceding statement points to an array of objects, the statement first calls the destructor for every object in the array, then deallocates the memory. If the preceding statement did not include the square brackets (`[]`) and `gradesArray` pointed to an array of objects, only the first object in the array would receive a destructor call.

### Common Programming Error 10.9



Using `delete` instead of `delete [ ]` for arrays of objects can lead to runtime logic errors. To ensure that every object in the array receives a destructor call, always delete memory allocated as an array with *operator delete [ ]*. Similarly, always delete memory allocated as an individual element with operator `delete`.

[◀ PREV](#)[NEXT ▶](#)

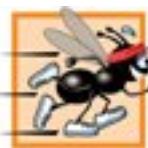
page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 553]

### Performance Tip 10.3



Use `static` data members to save storage when a single copy of the data for all objects of a class will suffice.

Although they may seem like global variables, a class's `static` data members have class scope. Also, `static` members can be declared `public`, `private` or `protected`. A fundamental-type `static` data member is initialized by default to 0. If you want a different initial value, a `static` data member can be initialized once (and only once). A `const` `static` data member of `int` or `enum` type can be initialized in its declaration in the class definition. However, all other `static` data members must be defined at file scope (i.e., outside the body of the class definition) and can be initialized only in those definitions. Note that `static` data members of class types (i.e., `static` member objects) that have default constructors need not be initialized because their default constructors will be called.

A class's `private` and `protected` `static` members are normally accessed through `public` member functions of the class or through `friends` of the class. (In [Chapter 12](#), we will see that a class's `private` and `protected` `static` members can also be accessed through `protected` member functions of the class.) A class's `static` members exist even when no objects of that class exist. To access a `public` `static` class member when no objects of the class exist, simply prefix the class name and the binary scope resolution operator (`::`) to the name of the data member. For example, if our preceding variable `martianCount` is `public`, it can be accessed with the expression `Martian::martianCount` when there are no `Martian` objects. (Of course, using `public` data is discouraged.)

A class's `public` `static` class members can also be accessed through any object of that class using the object's name, the dot operator and the name of the member (e.g., `myMartian.martianCount`). To access a `private` or `protected` `static` class member when no objects of the class exist, provide a `public` `static` member function and call the function by prefixing its name with the class name and binary scope resolution operator. (As we will see in [Chapter 12](#), a `protected` `static` member function can serve this purpose, too.) A `static` member function is a service of the class, not of a specific object of the class.

### Software Engineering Observation 10.10



A class's `static` data members and `static` member functions exist and can be used even if no objects of that class have been instantiated.

[Page 554]

The program of [Figs. 10.21](#)[10.23](#) demonstrates a `private static` data member called `count` ([Fig. 10.21](#), line 21) and a `public static` member function called `getCount` ([Fig. 10.21](#), line 15). In [Fig. 10.22](#), line 14 defines and initializes the data member `count` to zero at file scope and lines 1821 define `static` member function `getCount`. Notice that neither line 14 nor line 18 includes keyword `static`, yet both lines refer to static class members. When `static` is applied to an item at file scope, that item becomes known only in that file. The `static` members of the class need to be available from any client code that accesses the file, so we cannot declare them `static` in the `.cpp` file we declare them `static` only in the `.h` file. Data member `count` maintains a count of the number of objects of class `Employee` that have been instantiated. When objects of class `Employee` exist, member `count` can be referenced through any member function of an `Employee` object in [Fig. 10.22](#), `count` is referenced by both line 33 in the constructor and line 48 in the destructor. Also, note that since `count` is an `int`, it could have been initialized in the header file at line 21 of [Fig. 10.21](#).

**Figure 10.21. Employee class definition with a static data member to track the number of Employee objects in memory.**

```

1 // Fig. 10.21: Employee.h
2 // Employee class definition.
3 #ifndef EMPLOYEE_H
4 #define EMPLOYEE_H
5
6 class Employee
7 {
8 public:
9 Employee(const char * const, const char * const); // constructor
10 ~Employee(); // destructor
11 const char *getFirstName() const; // return first name
12 const char *getLastName() const; // return last name
13
14 // static member function
15 static int getCount(); // return number of objects instantiated
16 private:
17 char *firstName;
18 char *lastName;
19
20 // static data

```

```

21 static int count; // number of objects instantiated
22 } // end class Employee
23
24 #endif

```

**Figure 10.22. Employee class member-function definitions.**

(This item is displayed on pages 555 - 556 in the print version)

```

1 // Fig. 10.22: Employee.cpp
2 // Member-function definitions for class Employee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // strlen and strcpy prototypes
8 using std::strlen;
9 using std::strcpy;
10
11 #include "Employee.h" // Employee class definition
12
13 // define and initialize static data member at file scope
14 int Employee::count = 0;
15
16 // define static member function that returns number of
17 // Employee objects instantiated (declared static in Employee.h)
18 int Employee::getCount()
19 {
20 return count;
21 } // end static function getCount
22
23 // constructor dynamically allocates space for first and last name and
24 // uses strcpy to copy first and last names into the object
25 Employee::Employee(const char * const first, const char * const last)
26 {
27 firstName = new char[strlen(first) + 1];
28 strcpy(firstName, first);
29
30 lastName = new char[strlen(last) + 1];
31 strcpy(lastName, last);
32
33 count++; // increment static count of employees
34
35 cout << "Employee constructor for " << firstName

```

```

36 << ' ' << lastName << " called." << endl;
37 } // end Employee constructor
38
39 // destructor deallocates dynamically allocated memory
40 Employee::~Employee()
{
41 cout << "~Employee() called for " << firstName
42 << ' ' << lastName << endl;
43
44 delete [] firstName; // release memory
45 delete [] lastName; // release memory
46
47 count--; // decrement static count of employees
48 } // end ~Employee destructor
49
50
51 // return first name of employee
52 const char *Employee::getFirstName() const
53 {
54 // const before return type prevents client from modifying
55 // private data; client should copy returned string before
56 // destructor deletes storage to prevent undefined pointer
57 return firstName;
58 } // end function getFirstName
59
60 // return last name of employee
61 const char *Employee::getLastName() const
62 {
63 // const before return type prevents client from modifying
64 // private data; client should copy returned string before
65 // destructor deletes storage to prevent undefined pointer
66 return lastName;
67 } // end function getLastName

```

## Common Programming Error 10.10



It is a compilation error to include keyword `static` in the definition of a `static` data members at file scope.

In Fig. 10.22, note the use of the `new` operator (lines 27 and 30) in the `Employee` constructor to dynamically allocate the correct amount of memory for members `firstName` and `lastName`. If the `new` operator is unable to fulfill the request for memory for one or both of these character arrays, the program will terminate immediately. In Chapter 16, we will provide a better mechanism for dealing with cases in

which new is unable to allocate memory.

[Page 555]

Also note in Fig. 10.22 that the implementations of functions getFirstName (lines 5258) and getLastname (lines 6167) return pointers to const character data. In this implementation, if the client wishes to retain a copy of the first name or last name, the client is responsible for copying the dynamically allocated memory in the Employee object after obtaining the pointer to const character data from the object. It is also possible to implement getFirstName and getLastname, so the client is required to pass a character array and the size of the array to each function. Then the functions could copy the first or last name into the character array provided by the client. Once again, note that we could have used class string here to return a copy of a string object to the caller rather than returning a pointer to the private data.

[Page 556]

Figure 10.23 uses static member function getCount to determine the number of Employee objects currently instantiated. Note that when no objects are instantiated in the program, the Employee:: getCount() function call is issued (lines 14 and 38). However, when objects are instantiated, function getCount can be called through either of the objects, as shown in the statement at lines 2223, which uses pointer e1Ptr to invoke function getCount. Note that using e2Ptr->getCount() or Employee::getCount() in line 23 would produce the same result, because getCount always accesses the same static member count.

**Figure 10.23. static data member tracking the number of objects of a class.**

(This item is displayed on pages 556 - 557 in the print version)

```

1 // Fig. 10.23: fig10_23.cpp
2 // Driver to test class Employee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Employee.h" // Employee class definition
8
9 int main()
10 {
11 // use class name and binary scope resolution operator to
12 // access static number function getCount
13 cout << "Number of employees before instantiation of any objects is "
14 << Employee::getCount() << endl; // use class name
15 }
```

```

16 // use new to dynamically create two new Employees
17 // operator new also calls the object's constructor
18 Employee *e1Ptr = new Employee("Susan", "Baker");
19 Employee *e2Ptr = new Employee("Robert", "Jones");
20
21 // call getCount on first Employee object
22 cout << "Number of employees after objects are instantiated is "
23 << e1Ptr->getCount();
24
25 cout << "\n\nEmployee 1: "
26 << e1Ptr->getFirstName() << " " << e1Ptr->getLastName()
27 << "\nEmployee 2: "
28 << e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n\n";
29
30 delete e1Ptr; // deallocate memory
31 e1Ptr = 0; // disconnect pointer from free-store space
32 delete e2Ptr; // deallocate memory
33 e2Ptr = 0; // disconnect pointer from free-store space
34
35 // no objects exist, so call static member function getCount again
36 // using the class name and the binary scope resolution operator
37 cout << "Number of employees after objects are deleted is "
38 << Employee::getCount() << endl;
39 return 0;
40 } // end main

```

Number of employees before instantiation of any objects is 0  
 Employee constructor for Susan Baker called.  
 Employee constructor for Robert Jones called.  
 Number of employees after objects are instantiated is 2

Employee 1: Susan Baker  
 Employee 2: Robert Jones

$\sim$ Employee() called for Susan Baker  
 $\sim$ Employee() called for Robert Jones  
 Number of employees after objects are deleted is 0

## Software Engineering Observation 10.11



Some organizations specify in their software engineering standards that all calls to static member functions be made using the class name and not an object handle.

A member function should be declared `static` if it does not access non-static data members or non-static member functions of the class. Unlike non-static member functions, a static member function does not have a `this` pointer, because static data members and static member functions exist independently of any objects of a class. The `this` pointer must refer to a specific object of the class, and when a static member function is called, there might not be any objects of its class in memory.

[Page 558]

## Common Programming Error 10.11



Using the `this` pointer in a `static` member function is a compilation error.

## Common Programming Error 10.12



Declaring a `static` member function `const` is a compilation error. The `const` qualifier indicates that a function cannot modify the contents of the object in which it operates, but static member functions exist and operate independently of any objects of the class.

Lines 1819 of Fig. 10.23 use operator `new` to dynamically allocate two `Employee` objects. Remember that the program will terminate immediately if it is unable to allocate one or both of these objects. When each `Employee` object is allocated, its constructor is called. When `delete` is used at lines 30 and 32 to deallocate the two `Employee` objects, each object's destructor is called.

## Error-Prevention Tip 10.2



After deleting dynamically allocated memory, set the pointer that referred to that memory to 0. This disconnects the pointer from the previously allocated space on the free store. This space in memory could still contain information, despite having been deleted. By setting the pointer to 0, the program loses any access to that free-store space, which, in fact, could have already been reallocated for a different purpose. If you didn't set the pointer to 0, your code could inadvertently access this new information, causing extremely subtle, nonrepeatable logic errors.

[◀ PREV](#)[page footer](#)[NEXT ▶](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 559]

What is an abstract data type? Consider the built-in type `int`, which most people would associate with an integer in mathematics. Rather, an `int` is an abstract representation of an integer. Unlike mathematical integers, computer `ints` are fixed in size. For example, type `int` on today's popular 32-bit machines is typically limited to the range 2,147,483,648 to +2,147,483,647. If the result of a calculation falls outside this range, an "overflow" error occurs and the computer responds in some machine-dependent manner. It might, for example, "quietly" produce an incorrect result, such as a value too large to fit in an `int` variable (commonly called **arithmetic overflow**). Mathematical integers do not have this problem. Therefore, the notion of a computer `int` is only an approximation of the notion of a real-world integer. The same is true with `double`.

Even `char` is an approximation; `char` values are normally eight-bit patterns of ones and zeros; these patterns look nothing like the characters they represent, such as a capital `z`, a lowercase `z`, a dollar sign (`$`), a digit (5), and so on. Values of type `char` on most computers are quite limited compared with the range of real-world characters. The sevenbit ASCII character set ([Appendix B](#)) provides for 128 different character values. This is inadequate for representing languages such as Japanese and Chinese that require thousands of characters. As Internet and World Wide Web usage becomes pervasive, the newer Unicode character set is growing rapidly in popularity, owing to its ability to represent the characters of most languages. For more information on Unicode, visit [www.unicode.org](http://www.unicode.org).

The point is that even the built-in data types provided with programming languages like C++ are really only approximations or imperfect models of real-world concepts and behaviors. We have taken `int` for granted until this point, but now you have a new perspective to consider. Types like `int`, `double`, `char` and others are all examples of abstract data types. They are essentially ways of representing real-world notions to some satisfactory level of precision within a computer system.

An abstract data type actually captures two notions: A **data representation** and the **operations** that can be performed on those data. For example, in C++, an `int` contains an integer value (data) and provides addition, subtraction, multiplication, division and modulus operations (among others)division by zero is undefined. These allowed operations perform in a manner sensitive to machine parameters, such as the fixed word size of the underlying computer system. Another example is the notion of negative integers, whose operations and data representation are clear, but the operation of taking the square root of a negative integer is undefined. In C++, the programmer uses classes to implement abstract data types and their services. For example, to implement a stack ADT, we create our own stack classes in [Chapter 14](#), Templates and [Chapter 21](#), Data Structures, and we study the standard library `stack` class in [Chapter 23](#), Standard Template Library (STL).

### 10.8.1. Example: Array Abstract Data Type

We discussed arrays in [Chapter 7](#). As described there, an array is not much more than a pointer and some

space in memory. This primitive capability is acceptable for performing array operations if the programmer is cautious and undemanding. There are many operations that would be nice to perform with arrays, but that are not built into C++. With C++ classes, the programmer can develop an array ADT that is preferable to "raw" arrays. The array class can provide many helpful new capabilities such as

[Page 560]

- subscript range checking
- an arbitrary range of subscripts instead of having to start with 0
- array assignment
- array comparison
- array input/output
- arrays that know their sizes
- arrays that expand dynamically to accommodate more elements
- arrays that can print themselves in neat tabular format.

We create our own array class with many of these capabilities in [Chapter 11](#), Operator Overloading; String and Array Objects. Recall that C++ Standard Library class template `vector` (introduced in [Chapter 7](#)) provides many of these capabilities as well. [Chapter 23](#) explains class template `vector` in detail. C++ has a small set of built-in types. Classes extend the base programming language with new types.

#### Software Engineering Observation 10.12



The programmer is able to create new types through the class mechanism. These new types can be designed to be used as conveniently as the built-in types. Thus, C++ is an extensible language. Although the language is easy to extend with these new types, the base language itself cannot be changed.

New classes created in C++ environments can be proprietary to an individual, to small groups or to companies. Classes can also be placed in standard class libraries intended for wide distribution. ANSI (the American National Standards Institute) and ISO (the International Organization for Standardization) developed a standard version of C++ that includes a standard class library. The reader who learns C++ and object-oriented programming will be ready to take advantage of the new kinds of rapid, component-oriented software development made possible with increasingly abundant and rich libraries.

#### **10.8.2. Example: String Abstract Data Type**

C++ is an intentionally sparse language that provides programmers with only the raw capabilities needed to build a broad range of systems (consider it a tool for making tools). The language is designed to minimize performance burdens. C++ is appropriate for both applications programming and systems programming—the latter places extraordinary performance demands on programs. Certainly, it would have been possible to include a string data type among C++'s built-in data types. Instead, the language was designed to include mechanisms for creating and implementing string abstract data types through classes.

We introduced the C++ Standard Library class `string` in [Chapter 3](#), and in [Chapter 11](#) we will develop our own String ADT. We discuss class `string` in detail in [Chapter 18](#).

### 10.8.3. Example: Queue Abstract Data Type

Each of us stands in line from time to time. A waiting line is also called a **queue**. We wait in line at the supermarket checkout counter, we wait in line to get gasoline, we wait in line to board a bus, we wait in line to pay a highway toll, and students know all too well about waiting in line during registration to get the courses they want. Computer systems use many waiting lines internally, so we need to write programs that simulate what queues are and do.

[Page 561]

A queue is a good example of an abstract data type. Queues offer well-understood behavior to their clients. Clients put things in a queue one at a time invoking the queue's **enqueue** operation and the clients get those things back one at a time on demand invoking the queue's **dequeue** operation. Conceptually, a queue can become infinitely long. A real queue, of course, is finite. Items are returned from a queue in **first-in, first-out (FIFO)** order the first item inserted in the queue is the first item removed from the queue.

The queue hides an internal data representation that somehow keeps track of the items currently waiting in line, and it offers a set of operations to its clients, namely, enqueue and dequeue. The clients are not concerned about the implementation of the queue. Clients merely want the queue to operate "as advertised." When a client enqueues a new item, the queue should accept that item and place it internally in some kind of first-in, first-out data structure. When the client wants the next item from the front of the queue, the queue should remove the item from its internal representation and deliver it to the outside world (i.e., to the client of the queue) in FIFO order (i.e., the item that has been in the queue the longest should be the next one returned by the next dequeue operation).

The queue ADT guarantees the integrity of its internal data structure. Clients may not manipulate this data structure directly. Only the queue member functions have access to its internal data. Clients may cause only allowable operations to be performed on the data representation; operations not provided in the ADT's public interface are rejected in some appropriate manner. This could mean issuing an error message, throwing an exception (see [Chapter 16](#)), terminating execution or simply ignoring the operation request.

We create our own queue class in [Chapter 21](#), Data Structures, and we study the Standard Library `queue` class in [Chapter 23](#), Standard Template Library (STL).

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 561 (continued)]

## 10.9. Container Classes and Iterators

Among the most popular types of classes are **container classes** (also called **collection classes**), i.e., classes designed to hold collections of objects. Container classes commonly provide services such as insertion, deletion, searching, sorting, and testing an item to determine whether it is a member of the collection. Arrays, stacks, queues, trees and linked lists are examples of container classes; we studied arrays in [Chapter 7](#) and will study each of these other data structures in [Chapter 21](#), Data Structures, and [Chapter 23](#), Standard Template Library (STL).

It is common to associate **iterator objects** or more simply **iterators** with container classes. An iterator is an object that "walks through" a collection, returning the next item (or performing some action on the next item). Once an iterator for a class has been written, obtaining the next element from the class can be expressed simply. Just as a book being shared by several people could have several bookmarks in it at once, a container class can have several iterators operating on it at once. Each iterator maintains its own "position" information. We will discuss containers and iterators in detail in [Chapter 23](#).

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 563]

The member-function implementation file for proxy class `Interface` ([Fig. 10.26](#)) is the only file that includes the header file `Implementation.h` (line 5) containing class `Implementation`. The file `Interface.cpp` ([Fig. 10.26](#)) is provided to the client as a precompiled object code file along with the header file `Interface.h` that includes the function prototypes of the services provided by the proxy class. Because file `Interface.cpp` is made available to the client only as object code, the client is not able to see the interactions between the proxy class and the proprietary class (lines 9, 17, 23 and 29). Notice that the proxy class imposes an extra "layer" of function calls as the "price to pay" for hiding the private data of class `Implementation`. Given the speed of today's computers and the fact that many compilers can inline simple function calls automatically, the effect of these extra function calls on performance is often negligible.

[Page 564]

[Figure 10.27](#) tests class `Interface`. Notice that only the header file for `Interface` is included in the client code (line 7) there is no mention of the existence of a separate class called `Implementation`. Thus, the client never sees the private data of class `Implementation`, nor can the client code become dependent on the `Implementation` code.

Software Engineering Observation 10.13



A proxy class insulates client code from implementation changes.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 565]

## 10.11. Wrap-Up

In this chapter, we introduced several advanced topics related to classes and data abstraction. You learned how to specify `const` objects and `const` member functions to prevent modifications to objects, thus enforcing the principle of least privilege. You also learned that, through composition, a class can have objects of other classes as members. We introduced the topic of friendship and presented examples that demonstrate how to use `friend` functions.

You learned that the `this` pointer is passed as an implicit argument to each of a class's non-static member functions, allowing the functions to access the correct object's data members and other non-static member functions. You also saw explicit use of the `this` pointer to access the class's members and to enable cascaded member-function calls.

The chapter introduced the concept of dynamic memory management. You learned that C++ programmers can create and destroy objects dynamically with the `new` and `delete` operators. We motivated the need for static data members and demonstrated how to declare and use static data members and member functions in your own classes.

You learned about data abstraction and information hidingtwo of the fundamental concepts of object-oriented programming. We discussed abstract data typesways of representing real-world or conceptual notions to some satisfactory level of precision within a computer system. You then learned about three example abstract data typesarrays, strings and queues. We introduced the concept of a container class that holds a collection of objects, as well as the notion of an iterator class that walks through the elements of a container class. Finally, you learned how to create a proxy class to hide the implementation details (including the `private` data members) of a class from clients of the class.

In [Chapter 11](#), we continue our study of classes and objects by showing how to enable C++'s operators to work with objectsa process called operator overloading. For example, you will see how to "overload" the `<<` operator so it can be used to output a complete array without explicitly using a repetition statement.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 567]

which allocates an array of 100 integers and assigns the starting location of the array to `ptr`. The preceding array of integers is deleted with the statement

```
delete [] ptr;
```

- A static data member represents "class-wide" information (i.e., a property of the class shared by all instances, not a property of a specific object of the class).
- static data members have class scope and can be declared public, private or protected.
- A class's static members exist even when no objects of that class exist.
- To access a public static class member when no objects of the class exist, simply prefix the class name and the binary scope resolution operator (`::`) to the name of the data member.
- A class's public static class members can be accessed through any object of that class.
- A member function should be declared static if it does not access non-static data members or non-static member functions of the class. Unlike non-static member functions, a static member function does not have a `this` pointer, because static data members and static member functions exist independently of any objects of a class.
- Abstract data types are ways of representing real-world and conceptual notions to some satisfactory level of precision within a computer system.
- An abstract data type captures two notions: a data representation and the operations that can be performed on those data.
- C++ is an intentionally sparse language that provides programmers with only the raw capabilities needed to build a broad range of systems. C++ is designed to minimize performance burdens.
- Items are returned from a queue in first-in, first-out (FIFO) orderthe first item inserted in the queue is the first item removed from the queue.
- Container classes (also called collection classes) are designed to hold collections of objects. Container classes commonly provide services such as insertion, deletion, searching, sorting, and testing an item to determine whether it is a member of the collection.
- It is common to associate iterators with container classes. An iterator is an object that "walks through" a collection, returning the next item (or performing some action on the next item).
- Providing clients of your class with a proxy class that knows only the public interface to your class enables the clients to use your class's services without giving the clients access to your class's implementation details, such as its private data.
- When a class definition uses only a pointer or reference to an object of another class, the class header file for that other class (which would ordinarily reveal the private data of that class) is not required to be included with `#include`. You can simply declare that other class as a data type with a forward class declaration before the type is used in the file.
- The implementation file containing the member functions for a proxy class is the only file that includes the header file for the class whose private data we would like to hide.
- The implementation file containing the member functions for the proxy class is provided to the client as a precompiled object code file along with the header file that includes the function prototypes of the

services provided by the proxy class.

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 568]

container class

data abstraction

data representation

deallocate memory

`delete` operator

`delete[]` operator

dequeue (queue operation)

dynamic memory management

dynamic objects

enqueue (queue operation)

first-in, first-out (FIFO)

forward class declaration

free store

`friend` class

`friend` function

has-a relationship

heap

host object

information hiding

iterator

last-in, first-out (LIFO)

member initializer

member initializer list

member object

member object constructor

memory leak

`new [ ]` operator

`new` operator

operations in an ADT

proxy class

queue abstract data type

`static` data member

`static` member function

`this` pointer

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 569]

```
static int getCount() { cout << "Data is " << data << endl; return count; } // end function getCount
private: int data; static int count; }; // end class Example
```

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 569 (continued)]

## Answers to Self-Review Exercises

**10.1** a) member initializers. b) `friend`. c) `new`, pointer. d) initialized. e) `static`. f) `this`. g) `const`. h) default constructor. i) non-`static`. j) before. k) `delete`.

**10.2** Error: The class definition for `Example` has two errors. The first occurs in function `getIncrementedData`. The function is declared `const`, but it modifies the object.

Correction: To correct the first error, remove the `const` keyword from the definition of `getIncrementedData`.

Error: The second error occurs in function `getCount`. This function is declared `static`, so it is not allowed to access any non-`static` member of the class.

Correction: To correct the second error, remove the output line from the `getCount` definition.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 570]

## 10.9

Create class `IntegerSet` for which each object can hold integers in the range 0 through 100. A set is represented internally as an array of ones and zeros. Array element `a[ i ]` is 1 if integer `i` is in the set. Array element `a[ j ]` is 0 if integer `j` is not in the set. The default constructor initializes a set to the so-called "empty set," i.e., a set whose array representation contains all zeros.

Provide member functions for the common set operations. For example, provide a `unionOfSets` member function that creates a third set that is the set-theoretic union of two existing sets (i.e., an element of the third set's array is set to 1 if that element is 1 in either or both of the existing sets, and an element of the third set's array is set to 0 if that element is 0 in each of the existing sets).

Provide an `intersectionOfSets` member function which creates a third set which is the set-theoretic intersection of two existing sets (i.e., an element of the third set's array is set to 0 if that element is 0 in either or both of the existing sets, and an element of the third set's array is set to 1 if that element is 1 in each of the existing sets).

Provide an `insertElement` member function that inserts a new integer `k` into a set (by setting `a[ k ]` to 1). Provide a `deleteElement` member function that deletes integer `m` (by setting `a[ m ]` to 0).

Provide a `printSet` member function that prints a set as a list of numbers separated by spaces. Print only those elements that are present in the set (i.e., their position in the array has a value of 1). Print --- for an empty set.

Provide an `isEqualTo` member function that determines whether two sets are equal.

Provide an additional constructor that receives an array of integers and the size of that array and uses the array to initialize a set object.

Now write a driver program to test your `IntegerSet` class. Instantiate several `IntegerSet` objects. Test that all your member functions work properly.

## 10.10

It would be perfectly reasonable for the `Time` class of [Figs. 10.18 10.19](#) to represent the time internally as the number of seconds since midnight rather than the three integer values `hour`, `minute` and `second`. Clients could use the same public methods and get the same results. Modify the `Time` class of [Fig. 10.18](#) to implement the time as the number of seconds since midnight and show that there is no visible change in functionality to the clients of the class. [Note: This exercise nicely demonstrates the virtues of

implementation hiding.]

 PREV

NEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 572]

## Outline

[11.1 Introduction](#)

[11.2 Fundamentals of Operator Overloading](#)

[11.3 Restrictions on Operator Overloading](#)

[11.4 Operator Functions as Class Members vs. Global Functions](#)

[11.5 Overloading Stream Insertion and Stream Extraction Operators](#)

[11.6 Overloading Unary Operators](#)

[11.7 Overloading Binary Operators](#)

[11.8 Case Study: `Array` Class](#)

[11.9 Converting between Types](#)

[11.10 Case Study: `String` Class](#)

[11.11 Overloading `++` and `--`](#)

[11.12 Case Study: `A Date` Class](#)

[11.13 Standard Library Class `string`](#)

[11.14 explicit Constructors](#)

[11.15 Wrap-Up](#)

[Summary](#)

[Terminology](#)

## Self-Review Exercises

### Answers to Self-Review Exercises

### Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 573]

We discuss when to, and when not to, use operator overloading. We implement user-defined classes `PhoneNumber`, `Array`, `String` and `Date` to demonstrate how to overload operators, including the stream insertion, stream extraction, assignment, equality, relational, subscript, logical negation, parentheses and increment operators. The chapter ends with an example of C++'s Standard Library class `string`, which provides many overloaded operators that are similar to our `String` class that we present earlier in the chapter. In the exercises, we ask you to implement several classes with overloaded operators. The exercises also use classes `Complex` (for complex numbers) and `HugeInt` (for integers larger than a computer can represent with type `long`) to demonstrate overloaded arithmetic operators + and - and ask you to enhance those classes by overloading other arithmetic operators.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 574]

Overloading is especially appropriate for mathematical classes. These often require that a substantial set of operators be overloaded to ensure consistency with the way these mathematical classes are handled in the real world. For example, it would be unusual to overload only addition for a complex number class, because other arithmetic operators are also commonly used with complex numbers.

Operator overloading provides the same concise and familiar expressions for user-defined types that C++ provides with its rich collection of operators for fundamental types. Operator overloading is not automatic; you must write operator-overloading functions to perform the desired operations. Sometimes these functions are best made member functions; sometimes they are best as `friend` functions; occasionally they can be made global, non-`friend` functions. We discuss these issues throughout the chapter.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 575]

The associativity of an operator (i.e., whether the operator is applied right-to-left or left-to-right) cannot be changed by overloading.

It is not possible to change the "arity" of an operator (i.e., the number of operands an operator takes): Overloaded unary operators remain unary operators; overloaded binary operators remain binary operators. C++'s only ternary operator (?:) cannot be overloaded. Operators &, \*, + and - all have both unary and binary versions; these unary and binary versions can each be overloaded.

### Common Programming Error 11.2



Attempting to change the "arity" of an operator via operator overloading is a compilation error.

## Creating New Operators

It is not possible to create new operators; only existing operators can be overloaded. Unfortunately, this prevents the programmer from using popular notations like the \*\* operator used in some other programming languages for exponentiation. [Note: You could overload the ^ operator to perform exponentiation as it does in some other languages.]

### Common Programming Error 11.3



Attempting to create new operators via operator overloading is a syntax error.

## Operators for Fundamental Types

The meaning of how an operator works on objects of fundamental types cannot be changed by operator overloading. The programmer cannot, for example, change the meaning of how + adds two integers. Operator overloading works only with objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type.

## Software Engineering Observation 11.2



At least one argument of an *operator function* must be an object or reference of a user-defined type. This prevents programmers from changing how operators work on fundamental types.

## Common Programming Error 11.4



Attempting to modify how an operator works with objects of fundamental types is a compilation error.

## Related Operators

Overloading an assignment operator and an addition operator to allow statements like

```
object2 = object2 + object1;
```

does not imply that the `+=` operator is also overloaded to allow statements such as

```
object2 += object1;
```

Such behavior can be achieved only by explicitly overloading operator `+=` for that class.

## Common Programming Error 11.5



Assuming that overloading an operator such as `+` overloads related operators such as `+=` or that overloading `==` overloads a related operator like `!=` can lead to errors. Operators can be overloaded only explicitly; there is no implicit overloading.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 577]

## Commutative Operators

Another reason why one might choose a global function to overload an operator is to enable the operator to be commutative. For example, suppose we have an object, `number`, of type `long int`, and an object `bigInteger1`, of class `HugeInteger` (a class in which integers may be arbitrarily large rather than being limited by the machine word size of the underlying hardware; class `HugeInteger` is developed in the chapter exercises). The addition operator (+) produces a temporary `HugeInteger` object as the sum of a `HugeInteger` and a `long int` (as in the expression `bigInteger1 + number`), or as the sum of a `long int` and a `HugeInteger` (as in the expression `number + bigInteger1`). Thus, we require the addition operator to be commutative (exactly as it is with two fundamental-type operands). The problem is that the class object must appear on the left of the addition operator if that operator is to be overloaded as a member function. So, we overload the operator as a global function to allow the `HugeInteger` to appear on the right of the addition. The `operator+` function, which deals with the `HugeInteger` on the left, can still be a member function.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 579]

The stream extraction operator function `operator>>` (Fig. 11.4, lines 2231) takes `istream` reference `input` and `PhoneNumber` reference `num` as arguments and returns an `istream` reference. Operator function `operator>>` inputs phone numbers of the form

(800) 555-1212

**Figure 11.4. Overloaded stream insertion and stream extraction operators for class `PhoneNumber`.**

(This item is displayed on page 578 in the print version)

```

1 // Fig. 11.4: PhoneNumber.cpp
2 // Overloaded stream insertion and stream extraction operators
3 // for class PhoneNumber.
4 #include <iomanip>
5 using std::setw;
6
7 #include "PhoneNumber.h"
8
9 // overloaded stream insertion operator; cannot be
10 // a member function if we would like to invoke it with
11 // cout << somePhoneNumber;
12 ostream &operator<<(ostream &output, const PhoneNumber &number)
13 {
14 output << "(" << number.areaCode << ") "
15 << number.exchange << "-" << number.line;
16 return output; // enables cout << a << b << c;
17 } // end function operator<<
18
19 // overloaded stream extraction operator; cannot be
20 // a member function if we would like to invoke it with
21 // cin >> somePhoneNumber;
22 istream &operator>>(istream &input, PhoneNumber &number)
23 {
24 input.ignore(); // skip (
25 input >> setw(3) >> number.areaCode; // input area code
26 input.ignore(2); // skip) and space
27 input >> setw(3) >> number.exchange; // input exchange
28 input.ignore(); // skip dash (-)

```

```

29 input >> setw(4) >> number.line; // input line
30 return input; // enables cin >> a >> b >> c;
31 } // end function operator>>

```

into objects of class `PhoneNumber`. When the compiler sees the expression

```
cin >> phone
```

in line 19 of Fig. 11.5, the compiler generates the global function call

```
operator>>(cin, phone);
```

**Figure 11.5. Overloaded stream insertion and stream extraction operators.**

(This item is displayed on pages 578 - 579 in the print version)

```

1 // Fig. 11.5: fig11_05.cpp
2 // Demonstrating class PhoneNumber's overloaded stream insertion
3 // and stream extraction operators.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "PhoneNumber.h"
10
11 int main()
12 {
13 PhoneNumber phone; // create object phone
14
15 cout << "Enter phone number in the form (123) 456-7890:" << endl;
16
17 // cin >> phone invokes operator>> by implicitly issuing
18 // the global function call operator>>(cin, phone)
19 cin >> phone;
20
21 cout << "The phone number entered was: ";
22
23 // cout << phone invokes operator<< by implicitly issuing
24 // the global function call operator<<(cout, phone)
25 cout << phone << endl;
26 return 0;

```

```
27 } // end main
```

```
Enter phone number in the form (123) 456-7890:

(800) 555-1212

The phone number entered was: (800) 555-1212
```

When this call executes, reference parameter `input` (Fig. 11.4, line 22) becomes an alias for `cin` and reference parameter `number` becomes an alias for `phone`. The operator function reads as strings the three parts of the telephone number into the `areaCode` (line 25), `exchange` (line 27) and `line` (line 29) members of the `PhoneNumber` object referenced by parameter `number`. Stream manipulator `setw` limits the number of characters read into each character array. When used with `cin` and strings, `setw` restricts the number of characters read to the number of characters specified by its argument (i.e., `setw(3)` allows three characters to be read). The parentheses, space and dash characters are skipped by calling `istream` member function `ignore` (Fig. 11.4, lines 24, 26 and 28), which discards the specified number of characters in the input stream (one character by default). Function `operator>>` returns `istream` reference `input` (i.e., `cin`). This enables input operations on `PhoneNumber` objects to be cascaded with input operations on other `PhoneNumber` objects or on objects of other data types. For example, a program can input two `PhoneNumber` objects in one statement as follows:

---

[Page 580]

```
cin >> phone1 >> phone2;
```

First, the expression `cin >> phone1` executes by making the global function call

```
operator>>(cin, phone1);
```

This call then returns a reference to `cin` as the value of `cin >> phone1`, so the remaining portion of the expression is interpreted simply as `cin >> phone2`. This executes by making the global function call

```
operator>>(cin, phone2);
```

The stream insertion operator function (Fig. 11.4, lines 1217) takes an `ostream` reference (`output`) and a `const PhoneNumber` reference (`number`) as arguments and returns an `ostream` reference. Function `operator<<` displays objects of type `PhoneNumber`. When the compiler sees the expression

```
cout << phone
```

in line 25 of Fig. 11.5, the compiler generates the global function call

```
operator<<(cout, phone);
```

Function `operator<<` displays the parts of the telephone number as strings, because they are stored as `string` objects.

### Error-Prevention Tip 11.1



Returning a reference from an overloaded `<<` or `>>` operator function is typically successful because `cout`, `cin` and most stream objects are global, or at least long-lived. Returning a reference to an automatic variable or other temporary object is dangerous creating "dangling references" to nonexistent objects.

Note that the functions `operator>>` and `operator<<` are declared in `PhoneNumber` as global, friend functions (Fig. 11.3, lines 1516). They are global functions because the object of class `PhoneNumber` appears in each case as the right operand of the operator. Remember, overloaded operator functions for binary operators can be member functions only when the left operand is an object of the class in which the function is a member. Overloaded input and output operators are declared as friends if they need to access non-public class members directly for performance reasons or because the class may not offer appropriate get functions. Also note that the `PhoneNumber` reference in function `operator<<'s` parameter list (Fig. 11.4, line 12) is `const`, because the `PhoneNumber` will simply be output, and the `PhoneNumber` reference in function `operator>>'s` parameter list (line 22) is non-`const`, because the `PhoneNumber` object must be modified to store the input telephone number in the object.

### Software Engineering Observation 11.3



New input/output capabilities for user-defined types are added to C++ without modifying C++'s standard input/output library classes. This is another example of the extensibility of the C++ programming language.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 581]

## 11.6. Overloading Unary Operators

A unary operator for a class can be overloaded as a `non-static` member function with no arguments or as a global function with one argument; that argument must be either an object of the class or a reference to an object of the class. Member functions that implement overloaded operators must be `non-static` so that they can access the `non-static` data in each object of the class. Remember that `static` member functions can access only `static` data members of the class.

Later in this chapter, we will overload unary operator `!` to test whether an object of the `String` class we create ([Section 11.10](#)) is empty and return a `bool` result. Consider the expression `!s`, in which `s` is an object of class `String`. When a unary operator such as `!` is overloaded as a member function with no arguments and the compiler sees the expression `!s`, the compiler generates the call `s.operator!()`. The operand `s` is the class object for which the `String` class member function `operator!` is being invoked. The function is declared in the class definition as follows:

```
class String
{
public:
 bool operator!() const;
 ...
}; // end class String
```

A unary operator such as `!` may be overloaded as a global function with one argument in two different ways either with an argument that is an object (this requires a copy of the object, so the side effects of the function are not applied to the original object), or with an argument that is a reference to an object (no copy of the original object is made, so all side effects of this function are applied to the original object). If `s` is a `String` class object (or a reference to a `String` class object), then `!s` is treated as if the call `operator!( s )` had been written, invoking the global `operator!` function that is declared as follows:

```
bool operator!(const String &);
```

 PREV

NEXT 



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 582]

```
bool operator<(const String &, const String &);
```

 PREV

NEXT 

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 583]

**Figure 11.7. Array class member- and friend-function definitions.**

(This item is displayed on pages 583 - 586 in the print version)

```
1 // Fig 11.7: Array.cpp
2 // Member-function definitions for class Array
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // exit function prototype
13 using std::exit;
14
15 #include "Array.h" // Array class definition
16
17 // default constructor for class Array (default size 10)
18 Array::Array(int arraySize)
19 {
20 size = (arraySize > 0 ? arraySize : 10); // validate arraySize
21 ptr = new int[size]; // create space for pointer-based array
22
23 for (int i = 0; i < size; i++)
24 ptr[i] = 0; // set pointer-based array element
25 } // end Array default constructor
26
27 // copy constructor for class Array;
28 // must receive a reference to prevent infinite recursion
29 Array::Array(const Array &arrayToCopy)
30 : size(arrayToCopy.size)
31 {
32 ptr = new int[size]; // create space for pointer-based array
33
34 for (int i = 0; i < size; i++)
35 ptr[i] = arrayToCopy.ptr[i]; // copy into object
36 } // end Array copy constructor
```

```

37
38 // destructor for class Array
39 Array::~Array()
40 {
41 delete [] ptr; // release pointer-based array space
42 } // end destructor
43
44 // return number of elements of Array
45 int Array::getSize() const
46 {
47 return size; // number of elements in Array
48 } // end function getSize
49
50 // overloaded assignment operator;
51 // const return avoids: (a1 = a2) = a3
52 const Array &Array::operator=(const Array &right)
53 {
54 if (&right != this) // avoid self-assignment
55 {
56 // for Arrays of different sizes, deallocate original
57 // left-side array, then allocate new left-side array
58 if (size != right.size)
59 {
60 delete [] ptr; // release space
61 size = right.size; // resize this object
62 ptr = new int[size]; // create space for array copy
63 } // end inner if
64
65 for (int i = 0; i < size; i++)
66 ptr[i] = right.ptr[i]; // copy array into object
67 } // end outer if
68
69 return *this; // enables x = y = z, for example
70 } // end function operator=
71
72 // determine if two Arrays are equal and
73 // return true, otherwise return false
74 bool Array::operator==(const Array &right) const
75 {
76 if (size != right.size)
77 return false; // arrays of different number of elements
78
79 for (int i = 0; i < size; i++)
80 if (ptr[i] != right.ptr[i])
81 return false; // Array contents are not equal
82
83 return true; // Arrays are equal
84 } // end function operator==

```

```

85
86 // overloaded subscript operator for non-const Arrays;
87 // reference return creates a modifiable lvalue
88 int &Array::operator[](int subscript)
89 {
90 // check for subscript out-of-range error
91 if (subscript < 0 || subscript >= size)
92 {
93 cerr << "\nError: Subscript " << subscript
94 << " out of range" << endl;
95 exit(1); // terminate program; subscript out of range
96 } // end if
97
98 return ptr[subscript]; // reference return
99 } // end function operator[]
100
101 // overloaded subscript operator for const Arrays
102 // const reference return creates an rvalue
103 int Array::operator[](int subscript) const
104 {
105 // check for subscript out-of-range error
106 if (subscript < 0 || subscript >= size)
107 {
108 cerr << "\nError: Subscript " << subscript
109 << " out of range" << endl;
110 exit(1); // terminate program; subscript out of range
111 } // end if
112
113 return ptr[subscript]; // returns copy of this element
114 } // end function operator[]
115
116 // overloaded input operator for class Array;
117 // inputs values for entire Array
118 istream &operator>>(istream &input, Array &a)
119 {
120 for (int i = 0; i < a.size; i++)
121 input >> a.ptr[i];
122
123 return input; // enables cin >> x >> y;
124 } // end function
125
126 // overloaded output operator for class Array
127 ostream &operator<<(ostream &output, const Array &a)
128 {
129 int i;
130
131 // output private ptr-based array
132 for (i = 0; i < a.size; i++)

```

```

133 {
134 output << setw(12) << a.ptr[i];
135
136 if ((i + 1) % 4 == 0) // 4 numbers per row of output
137 output << endl;
138 } // end for
139
140 if (i % 4 != 0) // end last line of output
141 output << endl;
142
143 return output; // enables cout << x << y;
144 } // end function operator<<

```

[Page 586]

**Figure 11.8. Array class test program.**

(This item is displayed on pages 586 - 588 in the print version)

```

1 // Fig. 11.8: fig11_08.cpp
2 // Array class test program.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include "Array.h"
9
10 int main()
11 {
12 Array integers1(7); // seven-element Array
13 Array integers2; // 10-element Array by default
14
15 // print integers1 size and contents
16 cout << "Size of Array integers1 is "
17 << integers1.getSize()
18 << "\nArray after initialization:\n" << integers1;
19
20 // print integers2 size and contents
21 cout << "\nSize of Array integers2 is "
22 << integers2.getSize()
23 << "\nArray after initialization:\n" << integers2;
24

```

```

25 // input and print integers1 and integers2
26 cout << "\nEnter 17 integers:" << endl;
27 cin >> integers1 >> integers2;
28
29 cout << "\nAfter input, the Arrays contain:\n"
30 << "integers1:\n" << integers1
31 << "integers2:\n" << integers2;
32
33 // use overloaded inequality (!=) operator
34 cout << "\nEvaluating: integers1 != integers2" << endl;
35
36 if (integers1 != integers2)
37 cout << "integers1 and integers2 are not equal" << endl;
38
39 // create Array integers3 using integers1 as an
40 // initializer; print size and contents
41 Array integers3(integers1); // invokes copy constructor
42
43 cout << "\nSize of Array integers3 is "
44 << integers3.getSize()
45 << "\nArray after initialization:\n" << integers3;
46
47 // use overloaded assignment (=) operator
48 cout << "\nAssigning integers2 to integers1:" << endl;
49 integers1 = integers2; // note target Array is smaller
50
51 cout << "integers1:\n" << integers1
52 << "integers2:\n" << integers2;
53
54 // use overloaded equality (==) operator
55 cout << "\nEvaluating: integers1 == integers2" << endl;
56
57 if (integers1 == integers2)
58 cout << "integers1 and integers2 are equal" << endl;
59
60 // use overloaded subscript operator to create rvalue
61 cout << "\nintegers1[5] is " << integers1[5];
62
63 // use overloaded subscript operator to create lvalue
64 cout << "\n\nAssigning 1000 to integers1[5]" << endl;
65 integers1[5] = 1000;
66 cout << "integers1:\n" << integers1;
67
68 // attempt to use out-of-range subscript
69 cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
70 integers1[15] = 1000; // ERROR: out of range
71 return 0;
72 } // end main

```

```
Size of Array integers1 is 7
```

```
Array after initialization:
```

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 |   |

```
Size of Array integers2 is 10
```

```
Array after initialization:
```

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 0 | 0 |   |   |

```
Enter 17 integers:
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

```
After input, the Arrays contain:
```

```
integers1:
```

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 |   |

```
integers2:
```

|    |    |    |    |
|----|----|----|----|
| 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 |    |    |

```
Evaluating: integers1 != integers2
```

```
integers1 and integers2 are not equal
```

```
Size of Array integers3 is 7
```

```
Array after initialization:
```

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 |   |

```
Assigning integers2 to integers1:
```

```
integers1:
```

|    |    |    |    |
|----|----|----|----|
| 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 |    |    |

```
integers2:
```

|    |    |    |    |
|----|----|----|----|
| 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 |    |    |

```
Evaluating: integers1 == integers2
```

```
integers1 and integers2 are equal
```

```
integers1[5] is 13
```

```
Assigning 1000 to integers1[5]
```

```
integers1:
 8 9 10 11
 12 1000 14 15
 16 17
```

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range

[Page 589]

## Creating Arrays, Outputting Their Size and Displaying Their Contents

The program begins by instantiating two objects of class `Arrayintegers1` (Fig. 11.8, line 12) with seven elements, and `integers2` (Fig. 11.8, line 13) with the default `Array` size10 elements (specified by the `Array` default constructor's prototype in Fig. 11.6, line 15). Lines 1618 use member function `getSize` to determine the size of `integers1` and output `integers1`, using the `Array` overloaded stream insertion operator. The sample output confirms that the `Array` elements were set correctly to zeros by the constructor. Next, lines 2123 output the size of `Array integers2` and output `integers2`, using the `Array` overloaded stream insertion operator.

## Using the Overloaded Stream Insertion Operator to Fill an Array

Line 26 prompts the user to input 17 integers. Line 27 uses the `Array` overloaded stream extraction operator to read these values into both arrays. The first seven values are stored in `integers1` and the remaining 10 values are stored in `integers2`. Lines 2931 output the two arrays with the overloaded `Array` stream insertion operator to confirm that the input was performed correctly.

## Using the Overloaded Inequality Operator

Line 36 tests the overloaded inequality operator by evaluating the condition

```
integers1 != integers2
```

The program output shows that the Arrays indeed are not equal.

## Initializing a New Array with a Copy of an Existing Array's Contents

Line 41 instantiates a third `Array` called `integers3` and initializes it with a copy of `Array integers1`. This invokes the `Array` **copy constructor** to copy the elements of `integers1` into `integers3`. We discuss the details of the copy constructor shortly. Note that the copy constructor can also be invoked by writing line 41 as follows:

```
Array integers3 = integers1;
```

The equal sign in the preceding statement is not the assignment operator. When an equal sign appears in the declaration of an object, it invokes a constructor for that object. This form can be used to pass only a single argument to a constructor.

Lines 4345 output the size of `integers3` and output `integers3`, using the `Array` overloaded stream insertion operator to confirm that the `Array` elements were set correctly by the copy constructor.

## Using the Overloaded Assignment Operator

Next, line 49 tests the overloaded assignment operator (`=`) by assigning `integers2` to `integers1`. Lines 5152 print both `Array` objects to confirm that the assignment was successful. Note that `integers1` originally held 7 integers and was resized to hold a copy of the 10 elements in `integers2`. As we will see, the overloaded assignment operator performs this resizing operation in a manner that is transparent to the client code.

## Using the Overloaded Equality Operator

Next, line 57 uses the overloaded equality operator (`==`) to confirm that objects `integers1` and `integers2` are indeed identical after the assignment.

[Page 590]

## Using the Overloaded Subscript Operator

Line 61 uses the overloaded subscript operator to refer to `integers1[ 5 ]` an in-range element of `integers1`. This subscripted name is used as an rvalue to print the value stored in `integers1[ 5 ]`. Line 65 uses `integers1[ 5 ]` as a modifiable lvalue on the left side of an assignment statement to assign a new value, 1000, to element 5 of `integers1`. We will see that `operator[]` returns a reference to use as the modifiable lvalue after the operator confirms that 5 is a valid subscript for `integers1`.

Line 70 attempts to assign the value 1000 to `integers1[ 15 ]` an out-of-range element. In this example, `operator[]` determines that the subscript is out of range, prints a message and terminates the program. Note that we highlighted line 70 of the program in red to emphasize that it is an error to

access an element that is out of range. This is a runtime logic error, not a compilation error.

Interestingly, the array subscript operator [ ] is not restricted for use only with arrays; it also can be used, for example, to select elements from other kinds of container classes, such as linked lists, strings and dictionaries. Also, when `operator[ ]` functions are defined, subscripts no longer have to be integerscharacters, strings, floats or even objects of user-defined classes also could be used. In [Chapter 23](#), Standard Template Library (STL), we discuss the STL `map` class that allows noninteger subscripts.

## Array Class Definition

Now that we have seen how this program operates, let us walk through the class header ([Fig. 11.6](#)). As we refer to each member function in the header, we discuss that function's implementation in [Fig. 11.7](#). In [Fig. 11.6](#), lines 3536 represent the private data members of class `Array`. Each `Array` object consists of a `size` member indicating the number of elements in the `Array` and an `int` pointer`ptr`that points to the dynamically allocated pointer-based array of integers managed by the `Array` object.

## Overloading the Stream Insertion and Stream Extraction Operators as friends

Lines 1213 of [Fig. 11.6](#) declare the overloaded stream insertion operator and the overloaded stream extraction operator to be `friends` of class `Array`. When the compiler sees an expression like `cout << arrayObject`, it invokes global function `operator<<` with the call

```
operator<<(cout, arrayObject)
```

When the compiler sees an expression like `cin >> arrayObject`, it invokes global function `operator>>` with the call

```
operator>>(cin, arrayObject)
```

We note again that these stream insertion and stream extraction operator functions cannot be members of class `Array`, because the `Array` object is always mentioned on the right side of the stream insertion operator and the stream extraction operator. If these operator functions were to be members of class `Array`, the following awkward statements would have to be used to output and input an `Array`:

```
arrayObject << cout;
arrayObject >> cin;
```

Such statements would be confusing to most C++ programmers, who are familiar with `cout` and `cin` appearing as the left operands of `<<` and `>>`, respectively.

Function `operator<<` (defined in Fig. 11.7, lines 127144) prints the number of elements indicated by `size` from the integer array to which `ptr` points. Function `operator>>` (defined in Fig. 11.7, lines 118124) inputs directly into the array to which `ptr` points. Each of these operator functions returns an appropriate reference to enable cascaded output or input statements, respectively. Note that each of these functions has access to an `Array`'s private data because these functions are declared as friends of class `Array`. Also, note that class `Array`'s `getSize` and `operator[]` functions could be used by `operator<<` and `operator>>`, in which case these operator functions would not need to be friends of class `Array`. However, the additional function calls might increase execution-time overhead.

## Array Default Constructor

Line 15 of Fig. 11.6 declares the default constructor for the class and specifies a default size of 10 elements. When the compiler sees a declaration like line 13 in Fig. 11.8, it invokes class `Array`'s default constructor (remember that the default constructor in this example actually receives a single `int` argument that has a default value of 10). The default constructor (defined in Fig. 11.7, lines 1825) validates and assigns the argument to data member `size`, uses `new` to obtain the memory for the internal pointer-based representation of this array and assigns the pointer returned by `new` to data member `ptr`. Then the constructor uses a `for` statement to set all the elements of the array to zero. It is possible to have an `Array` class that does not initialize its members if, for example, these members are to be read at some later time; but this is considered to be a poor programming practice. Arrays, and objects in general, should be properly initialized and maintained in a consistent state.

## Array Copy Constructor

Line 16 of Fig. 11.6 declares a **copy constructor** (defined in Fig. 11.7, lines 2936) that initializes an `Array` by making a copy of an existing `Array` object. Such copying must be done carefully to avoid the pitfall of leaving both `Array` objects pointing to the same dynamically allocated memory. This is exactly the problem that would occur with default memberwise copying, if the compiler is allowed to define a default copy constructor for this class. Copy constructors are invoked whenever a copy of an object is needed, such as in passing an object by value to a function, returning an object by value from a function or initializing an object with a copy of another object of the same class. The copy constructor is called in a declaration when an object of class `Array` is instantiated and initialized with another object of class `Array`, as in the declaration in line 41 of Fig. 11.8.

### Software Engineering Observation 11.4



The argument to a copy constructor should be a `const` reference to allow a `const` object to be copied.

## Common Programming Error 11.6



Note that a copy constructor must receive its argument by reference, not by value. Otherwise, the copy constructor call results in infinite recursion (a fatal logic error) because receiving an object by value requires the copy constructor to make a copy of the argument object. Recall that any time a copy of an object is required, the class's copy constructor is called. If the copy constructor received its argument by value, the copy constructor would call itself recursively to make a copy of its argument!

[Page 592]

The copy constructor for `Array` uses a member initializer (Fig. 11.7, line 30) to copy the `size` of the initializer `Array` into data member `size`, uses `new` (line 32) to obtain the memory for the internal pointer-based representation of this `Array` and assigns the pointer returned by `new` to data member `ptr`.

<sup>[1]</sup> Then the copy constructor uses a `for` statement to copy all the elements of the initializer `Array` into the new `Array` object. Note that an object of a class can look at the `private` data of any other object of that class (using a handle that indicates which object to access).

<sup>[1]</sup> Note that `new` could fail to obtain the needed memory. We deal with `new` failures in Chapter 16, Exception Handling.

## Common Programming Error 11.7



If the copy constructor simply copied the pointer in the source object to the target object's pointer, then both objects would point to the same dynamically allocated memory. The first destructor to execute would then delete the dynamically allocated memory, and the other object's `ptr` would be undefined, a situation called a `dangling pointer`—this would likely result in a serious runtime error (such as early program termination) when the pointer was used.

## Array Destructor

Line 17 of Fig. 11.6 declares the destructor for the class (defined in Fig. 11.7, lines 3942). The destructor is invoked when an object of class `Array` goes out of scope. The destructor uses `delete []` to release the memory allocated dynamically by `new` in the constructor.

## getSize Member Function

Line 18 of Fig. 11.6 declares function `getSize` (defined in Fig. 11.7, lines 4548) that returns the number of elements in the `Array`.

## Overloaded Assignment Operator

Line 20 of Fig. 11.6 declares the overloaded assignment operator function for the class. When the compiler sees the expression `integers1 = integers2` in line 49 of Fig. 11.8, the compiler invokes member function `operator=` with the call

```
integers1.operator=(integers2)
```

The implementation of member function `operator=` (Fig. 11.7, lines 5270) tests for **self assignment** (line 54) in which an object of class `Array` is being assigned to itself. When `this` is equal to the address of the right operand, a self-assignment is being attempted, so the assignment is skipped (i.e., the object already is itself; in a moment we will see why self-assignment is dangerous). If it is not a self-assignment, then the member function determines whether the sizes of the two arrays are identical (line 58); in that case, the original array of integers in the left-side `Array` object is not reallocated. Otherwise, `operator=` uses `delete` (line 60) to release the memory originally allocated to the target array, copies the size of the source array to the size of the target array (line 61), uses `new` to allocate memory for the target array and places the pointer returned by `new` into the array's `ptr` member.<sup>[2]</sup> Then the `for` statement at lines 6566 copies the array elements from the source array to the target array. Regardless of whether this is a self-assignment, the member function returns the current object (i.e., `*this` at line 69) as a constant reference; this enables cascaded `Array` assignments such as `x = y = z`. If self-assignment occurs, and function `operator=` did not test for this case, `operator=` would delete the dynamic memory associated with the `Array` object before the assignment was complete. This would leave `ptr` pointing to memory that had been deallocated, which could lead to fatal runtime errors.

<sup>[2]</sup> Once again, `new` could fail. We discuss `new` failures in Chapter 16.

[Page 593]

### Software Engineering Observation 11.5



A copy constructor, a destructor and an overloaded assignment operator are usually provided as a group for any class that uses dynamically allocated memory.

### Common Programming Error 11.8



Not providing an overloaded assignment operator and a copy constructor for a class when objects of that class contain pointers to dynamically allocated memory is a logic error.

## Software Engineering Observation 11.6



It is possible to prevent one object of a class from being assigned to another. This is done by declaring the assignment operator as a `private` member of the class.

## Software Engineering Observation 11.7



It is possible to prevent class objects from being copied; to do this, simply make both the overloaded assignment operator and the copy constructor of that class `private`.

## Overloaded Equality and Inequality Operators

Line 21 of Fig. 11.6 declares the overloaded equality operator (`==`) for the class. When the compiler sees the expression `integers1 == integers2` in line 57 of Fig. 11.8, the compiler invokes member function `operator==` with the call

```
integers1.operator==(integers2)
```

Member function `operator==` (defined in Fig. 11.7, lines 7484) immediately returns `false` if the `size` members of the arrays are not equal. Otherwise, `operator==` compares each pair of elements. If they are all equal, the function returns `TRUE`. The first pair of elements to differ causes the function to return `false` immediately.

Lines 2427 of the header file define the overloaded inequality operator (`!=`) for the class. Member function `operator!=` uses the overloaded `operator==` function to determine whether one `Array` is equal to another, then returns the opposite of that result. Writing `operator!=` in this manner enables the programmer to reuse `operator==`, which reduces the amount of code that must be written in the class. Also, note that the full function definition for `operator!=` is in the `Array` header file. This allows the compiler to inline the definition of `operator!=` to eliminate the overhead of the extra function call.

## Overloaded Subscript Operators

Lines 30 and 33 of Fig. 11.6 declare two overloaded subscript operators (defined in Fig. 11.7 at lines 8899 and 103114, respectively). When the compiler sees the expression `integers1[ 5 ]` (Fig. 11.8, line 61), the compiler invokes the appropriate overloaded `operator[]` member function by generating the call

```
integers1.operator[](5)
```

The compiler creates a call to the `const` version of `operator[]` (Fig. 11.7, lines 103114) when the subscript operator is used on a `const` `Array` object. For example, if `const` object `z` is instantiated with the statement

[Page 594]

```
const Array z(5);
```

then the `const` version of `operator[]` is required to execute a statement such as

```
cout << z[3] << endl;
```

Remember, a program can invoke only the `const` member functions of a `const` object.

Each definition of `operator[]` determines whether the subscript it receives as an argument is in range. If it is not, each function prints an error message and terminates the program with a call to function `exit` (header `<cstdlib>`).<sup>[3]</sup> If the subscript is in range, the non-`const` version of `operator[]` returns the appropriate array element as a reference so that it may be used as a modifiable lvalue (e.g., on the left side of an assignment statement). If the subscript is in range, the `const` version of `operator[]` returns a copy of the appropriate element of the array. The returned character is an rvalue.

<sup>[3]</sup> Note that it is more appropriate when a subscript is out of range to "throw an exception" indicating the out-of-range subscript. Then the program can "catch" that exception, process it and possibly continue execution. See Chapter 16 for more information on exceptions.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 595]

Overloaded cast operator functions can be defined to convert objects of user-defined types into fundamental types or into objects of other user-defined types. The prototypes

```
A::operator int() const;
A::operator OtherClass() const;
```

declare overloaded cast operator functions that can convert an object of user-defined type A into an integer or into an object of user-defined type OtherClass, respectively.

One of the nice features of cast operators and conversion constructors is that, when necessary, the compiler can call these functions implicitly to create temporary objects. For example, if an object s of a user-defined String class appears in a program at a location where an ordinary `char *` is expected, such as

```
cout << s;
```

the compiler can call the overloaded cast-operator function `operator char *` to convert the object into a `char *` and use the resulting `char *` in the expression. With this cast operator provided for our String class, the stream insertion operator does not have to be overloaded to output a String using `cout`.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 603]

## Overloading the Stream Insertion and Stream Extraction Operators as friends

Lines 1213 (Fig. 11.9) declare the overloaded stream insertion operator function `operator<<` (defined in Fig. 11.10, lines 170174) and the overloaded stream extraction operator function `operator>>` (defined in Fig. 11.10, lines 177183) as `friends` of the class. The implementation of `operator<<` is straightforward. Note that `operator>>` restricts the total number of characters that can be read into array `temp` to 99 with `setw` (line 180); the 100th position is reserved for the string's terminating null character. [ Note: We did not have this restriction for `operator>>` in class `Array` (Figs. 11.611.7), because that class's `operator>>` read one array element at a time and stopped reading values when the end of the array was reached. Object `cin` does not know how to do this by default for input of character arrays.] Also, note the use of `operator=` (line 181) to assign the C-style string `temp` to the `String` object to which `s` refers. This statement invokes the conversion constructor to create a temporary `String` object containing the C-style string; the temporary `String` is then assigned to `s`. We could eliminate the overhead of creating the temporary `String` object here by providing another overloaded assignment operator that receives a parameter of type `const char *`.

**Figure 11.10. String class member-function and friend-function definitions.**

(This item is displayed on pages 597 - 600 in the print version)

```

1 // Fig. 11.10: String.cpp
2 // Member-function definitions for class String.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <cstring> // strcpy and strcat prototypes
12 using std::strcmp;
13 using std::strcpy;
14 using std::strcat;
15
16 #include <cstdlib> // exit prototype
17 using std::exit;
18
19 #include "String.h" // String class definition

```

```

20
21 // conversion (and default) constructor converts char * to String
22 String::String(const char *s)
23 : length((s != 0) ? strlen(s) : 0)
24 {
25 cout << "Conversion (and default) constructor: " << s << endl;
26 setString(s); // call utility function
27 } // end String conversion constructor
28
29 // copy constructor
30 String::String(const String ©)
31 : length(copy.length)
32 {
33 cout << "Copy constructor: " << copy.sPtr << endl;
34 setString(copy.sPtr); // call utility function
35 } // end String copy constructor
36
37 // Destructor
38 String::~String()
39 {
40 cout << "Destructor: " << sPtr << endl;
41 delete [] sPtr; // release pointer-based string memory
42 } // end ~String destructor
43
44 // overloaded = operator; avoids self assignment
45 const String &String::operator=(const String &right)
46 {
47 cout << "operator= called" << endl;
48
49 if (&right != this) // avoid self assignment
50 {
51 delete [] sPtr; // prevents memory leak
52 length = right.length; // new String length
53 setString(right.sPtr); // call utility function
54 } // end if
55 else
56 cout << "Attempted assignment of a String to itself" << endl;
57
58 return *this; // enables cascaded assignments
59 } // end function operator=
60
61 // concatenate right operand to this object and store in this object
62 const String &String::operator+=(const String &right)
63 {
64 size_t newLength = length + right.length; // new length
65 char *tempPtr = new char[newLength + 1]; // create memory
66
67 strcpy(tempPtr, sPtr); // copy sPtr

```

```

68 strcpy(tempPtr + length, right.sPtr); // copy right.sPtr
69
70 delete [] sPtr; // reclaim old space
71 sPtr = tempPtr; // assign new array to sPtr
72 length = newLength; // assign new length to length
73 return *this; // enables cascaded calls
74 } // end function operator+=
75
76 // is this String empty?
77 bool String::operator!() const
78 {
79 return length == 0;
80 } // end function operator!
81
82 // Is this String equal to right String?
83 bool String::operator==(const String &right) const
84 {
85 return strcmp(sPtr, right.sPtr) == 0;
86 } // end function operator==
87
88 // Is this String less than right String?
89 bool String::operator<(const String &right) const
90 {
91 return strcmp(sPtr, right.sPtr) < 0;
92 } // end function operator<
93
94 // return reference to character in String as a modifiable lvalue
95 char &String::operator[](int subscript)
96 {
97 // test for subscript out of range
98 if (subscript < 0 || subscript >= length)
99 {
100 cerr << "Error: Subscript " << subscript
101 << " out of range" << endl;
102 exit(1); // terminate program
103 } // end if
104
105 return sPtr[subscript]; // non-const return; modifiable lvalue
106 } // end function operator[]
107
108 // return reference to character in String as rvalue
109 char String::operator[](int subscript) const
110 {
111 // test for subscript out of range
112 if (subscript < 0 || subscript >= length)
113 {
114 cerr << "Error: Subscript " << subscript
115 << " out of range" << endl;

```

```

116 exit(1); // terminate program
117 } // end if
118
119 return sPtr[subscript]; // returns copy of this element
120 } // end function operator[]
121
122 // return a substring beginning at index and of length subLength
123 String String::operator()(int index, int subLength) const
124 {
125 // if index is out of range or substring length < 0,
126 // return an empty String object
127 if (index < 0 || index >= length || subLength < 0)
128 return ""; // converted to a String object automatically
129
130 // determine length of substring
131 int len;
132
133 if ((subLength == 0) || (index + subLength > length))
134 len = length - index;
135 else
136 len = subLength;
137
138 // allocate temporary array for substring and
139 // terminating null character
140 char *tempPtr = new char[len + 1];
141
142 // copy substring into char array and terminate string
143 strncpy(tempPtr, &sPtr[index], len);
144 tempPtr[len] = '\0';
145
146 // create temporary String object containing the substring
147 String tempString(tempPtr);
148 delete [] tempPtr; // delete temporary array
149 return tempString; // return copy of the temporary String
150 } // end function operator()
151
152 // return string length
153 int String::getLength() const
154 {
155 return length;
156 } // end function getLength
157
158 // utility function called by constructors and operator=
159 void String::setString(const char *string2)
160 {
161 sPtr = new char[length + 1]; // allocate memory
162
163 if (string2 != 0) // if string2 is not null pointer, copy contents

```

```

164 strcpy(sPtr, string2); // copy literal to object
165 else // if string2 is a null pointer, make this an empty string
166 sPtr[0] = '\0'; // empty string
167 } // end function setString
168
169 // overloaded output operator
170 ostream &operator<<(ostream &output, const String &s)
171 {
172 output << s.sPtr;
173 return output; // enables cascading
174 } // end function operator<<
175
176 // overloaded input operator
177 istream &operator>>(istream &input, String &s)
178 {
179 char temp[100]; // buffer to store input
180 input >> setw(100) >> temp;
181 s = temp; // use String class assignment operator
182 return input; // enables cascading
183 } // end function operator>>

```

## String Conversion Constructor

Line 15 (Fig. 11.9) declares a conversion constructor. This constructor (defined in Fig. 11.10, lines 2227) takes a `const char *` argument (that defaults to the empty string; Fig. 11.9, line 15) and initializes a `String` object containing that same character string. Any [single-argument constructor](#) can be thought of as a conversion constructor. As we will see, such constructors are helpful when we are doing any `String` operation using `char *` arguments. The conversion constructor can convert a `char *` string into a `String` object, which can then be assigned to the target `String` object. The availability of this conversion constructor means that it is not necessary to supply an overloaded assignment operator for specifically assigning character strings to `String` objects. The compiler invokes the conversion constructor to create a temporary `String` object containing the character string; then the overloaded assignment operator is invoked to assign the temporary `String` object to another `String` object.

### Software Engineering Observation 11.8



When a conversion constructor is used to perform an implicit conversion, C++ can apply only one implicit constructor call (i.e., a single user-defined conversion) to try to match the needs of another overloaded operator. The compiler will not match an overloaded operator's needs by performing a series of implicit, user-defined conversions.

The `String` conversion constructor could be invoked in such a declaration as `String s1( "happy" )`. The conversion constructor calculates the length of its character-string argument and assigns it to data member `length` in the member-initializer list. Then, line 26 calls utility function `setString` (defined in Fig. 11.10, lines 159167), which uses `new` to allocate a sufficient amount of memory to private data member `sPtr` and uses `strcpy` to copy the character string into the memory to which `sPtr` points.<sup>[4]</sup>

<sup>[4]</sup> There is a subtle issue in the implementation of this conversion constructor. As implemented, if a null pointer (i.e., 0) is passed to the constructor, the program will fail. The proper way to implement this constructor would be to detect whether the constructor argument is a null pointer, then "throw an exception." Chapter 16 discusses how we can make classes more robust in this manner. Also, note that a null pointer (0) is not the same as the empty string (""). A null pointer is a pointer that does not point to anything. An empty string is an actual string that contains only a null character ('\0').

## String Copy Constructor

Line 16 in Fig. 11.9 declares a copy constructor (defined in Fig. 11.10, lines 3035) that initializes a `String` object by making a copy of an existing `String` object. As with our class `Array` (Figs. 11.611.7), such copying must be done carefully to avoid the pitfall in which both `String` objects point to the same dynamically allocated memory. The copy constructor operates similarly to the conversion constructor, except that it simply copies the `length` member from the source `String` object to the target `String` object. Note that the copy constructor calls `setString` to create new space for the target object's internal character string. If it simply copied the `sPtr` in the source object to the target object's `sPtr`, then both objects would point to the same dynamically allocated memory. The first destructor to execute would then delete the dynamically allocated memory, and the other object's `sPtr` would be undefined (i.e., `sPtr` would be a dangling pointer), a situation likely to cause a serious runtime error.

## String Destructor

Line 17 of Fig. 11.9 declares the `String` destructor (defined in Fig. 11.10, lines 3842). The destructor uses `delete []` to release the dynamic memory to which `sPtr` points.

## Overloaded Assignment Operator

Line 19 (Fig. 11.9) declares the overloaded assignment operator function `operator=` (defined in Fig. 11.10, lines 4559). When the compiler sees an expression like `string1 = string2`, the compiler generates the function call

```
string1.operator=(string2);
```

The overloaded assignment operator function `operator=` tests for self-assignment. If this is a self-assignment, the function does not need to change the object. If this test were omitted, the function would immediately delete the space in the target object and thus lose the character string, such that the pointer would no longer be pointing to valid dataa classic example of a dangling pointer. If there is no self-assignment, the function deletes the memory and copies the `length` field of the source object to the target object. Then `operator=` calls `setString` to create new space for the target object and copy the character string from the source object to the target object. Whether or not this is a self-assignment, `operator=` returns `*this` to enable cascaded assignments.

---

[Page 605]

## Overloaded Addition Assignment Operator

Line 20 of Fig. 11.9 declares the overloaded string-concatenation operator `+=` (defined in Fig. 11.10, lines 6274). When the compiler sees the expression `s1 += s2` (line 40 of Fig. 11.11), the compiler generates the member-function call

```
s1.operator+=(s2)
```

**Figure 11.11. String class test program.**

(This item is displayed on pages 600 - 603 in the print version)

```

1 // Fig. 11.11: fig11_11.cpp
2 // String class test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::boolalpha;
7
8 #include "String.h"
9
10 int main()
11 {
12 String s1("happy");
13 String s2(" birthday");
14 String s3;
15
16 // test overloaded equality and relational operators
17 cout << "s1 is \" " << s1 << "\"; s2 is \" " << s2
18 << "\"; s3 is \" " << s3 << '\"'
19 << boolalpha << "\n\nThe results of comparing s2 and s1:"
20 << "\ns2 == s1 yields " << (s2 == s1)

```

```

21 << "\ns2 != s1 yields " << (s2 != s1)
22 << "\ns2 > s1 yields " << (s2 > s1)
23 << "\ns2 < s1 yields " << (s2 < s1)
24 << "\ns2 >= s1 yields " << (s2 >= s1)
25 << "\ns2 <= s1 yields " << (s2 <= s1);
26
27
28 // test overloaded String empty (!) operator
29 cout << "\n\nTesting !s3:" << endl;
30
31 if (!s3)
32 {
33 cout << "s3 is empty; assigning s1 to s3;" << endl;
34 s3 = s1; // test overloaded assignment
35 cout << "s3 is \" " << s3 << "\" ";
36 } // end if
37
38 // test overloaded String concatenation operator
39 cout << "\n\ns1 += s2 yields s1 = ";
40 s1 += s2; // test overloaded concatenation
41 cout << s1;
42
43 // test conversion constructor
44 cout << "\n\ns1 += \" to you\" yields" << endl;
45 s1 += " to you"; // test conversion constructor
46 cout << "s1 = " << s1 << "\n\n";
47
48 // test overloaded function call operator () for substring
49 cout << "The substring of s1 starting at\n"
50 << "location 0 for 14 characters, s1(0, 14), is:\n"
51 << s1(0, 14) << "\n\n";
52
53 // test substring "to-end-of-String" option
54 cout << "The substring of s1 starting at\n"
55 << "location 15, s1(15), is: "
56 << s1(15) << "\n\n";
57
58 // test copy constructor
59 String *s4Ptr = new String(s1);
60 cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
61
62 // test assignment (=) operator with self-assignment
63 cout << "assigning *s4Ptr to *s4Ptr" << endl;
64 *s4Ptr = *s4Ptr; // test overloaded assignment
65 cout << "*s4Ptr = " << *s4Ptr << endl;
66
67 // test destructor
68 delete s4Ptr;

```

```

69
70 // test using subscript operator to create a modifiable lvalue
71 s1[0] = 'H';
72 s1[6] = 'B';
73 cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
74 << s1 << "\n\n";
75
76 // test subscript out of range
77 cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
78 s1[30] = 'd'; // ERROR: subscript out of range
79 return 0;
80 } // end main

```

Conversion (and default) constructor: happy  
 Conversion (and default) constructor: birthday  
 Conversion (and default) constructor:  
 s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:

```

s2 == s1 yields false
s2 != s1 yields true
s2 > s1 yields false
s2 < s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

```

Testing !s3:

```

s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

```

```
s1 += s2 yields s1 = happy birthday
```

```
s1 += " to you" yields
Conversion (and default) constructor: to you
Destructor: to you
s1 = happy birthday to you
```

Conversion (and default) constructor: happy birthday  
 Copy constructor: happy birthday  
 Destructor: happy birthday  
 The substring of s1 starting at  
 location 0 for 14 characters, s1(0, 14), is:  
 happy birthday

Destructor: happy birthday  
 Conversion (and default) constructor: to you

```

Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15), is: to you

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Error: Subscript 30 out of range

```

Function `operator+=` calculates the combined length of the concatenated string and stores it in local variable `newLength`, then creates a temporary pointer (`tempPtr`) and allocates a new character array in which the concatenated string will be stored. Next, `operator+=` uses `strcpy` to copy the original character strings from `sPtr` and `right.sPtr` into the memory to which `tempPtr` points. Note that the location into which `strcpy` will copy the first character of `right.sPtr` is determined by the pointer-arithmetic calculation `tempPtr + length`. This calculation indicates that the first character of `right.sPtr` should be placed at location `length` in the array to which `tempPtr` points. Next, `operator+=` uses `delete []` to release the space occupied by this object's original character string, assigns `tempPtr` to `sPtr` so that this `String` object points to the new character string, assigns `newLength` to `length` so that this `String` object contains the new string length and returns `*this` as a `const String &` to enable cascading of `+=` operators.

Do we need a second overloaded concatenation operator to allow concatenation of a `String` and a `char *`? No. The `const char *` conversion constructor converts a C-style string into a temporary `String` object, which then matches the existing overloaded concatenation operator. This is exactly what the compiler does when it encounters line 44 in Fig. 11.11. Again, C++ can perform such conversions only one level deep to facilitate a match. C++ can also perform an implicit compiler-defined conversion between fundamental types before it performs the conversion between a fundamental type and a class. Note that, when a temporary `String` object is created in this case, the conversion constructor and the destructor are called (see the output resulting from line 45, `s1 += "to you"`, in Fig. 11.11). This is an example of function-call overhead that is hidden from the client of the class when temporary class objects

are created and destroyed during implicit conversions. Similar overhead is generated by copy constructors in call-by-value parameter passing and in returning class objects by value.

### Performance Tip 11.2



Overloading the `+=` concatenation operator with an additional version that takes a single argument of type `const char *` executes more efficiently than having only a version that takes a `String` argument. Without the `const char *` version of the `+=` operator, a `const char *` argument would first be converted to a `String` object with class `String`'s conversion constructor, then the `+=` operator that receives a `String` argument would be called to perform the concatenation.

### Software Engineering Observation 11.9



Using implicit conversions with overloaded operators, rather than overloading operators for many different operand types, often requires less code, which makes a class easier to modify, maintain and debug.

---

[Page 606]

## Overloaded Negation Operator

Line 22 of Fig. 11.9 declares the overloaded negation operator (defined in Fig. 11.10, lines 7780). This operator determines whether an object of our `String` class is empty. For example, when the compiler sees the expression `!string1`, it generates the function call

```
string1.operator!()
```

This function simply returns the result of testing whether `length` is equal to zero.

## Overloaded Equality and Relational Operators

Lines 2324 of Fig. 11.9 declare the overloaded equality operator (defined in Fig. 11.10, lines 8386) and the overloaded less-than operator (defined in Fig. 11.10, lines 8992) for class `String`. These are similar, so let us discuss only one example, namely, overloading the `==` operator. When the compiler sees the expression `string1 == string2`, the compiler generates the member-function call

```
string1.operator==(string2)
```

which returns `true` if `string1` is equal to `string2`. Each of these operators uses function `strcmp` (from `<cstring>`) to compare the character strings in the `String` objects. Many C++ programmers advocate using some of the overloaded operator functions to implement others. So, the `!=`, `>`, `<=` and `>=` operators are implemented (Fig. 11.9, lines 2748) in terms of `operator==` and `operator<`. For example, overloaded function `operator>=` (implemented at lines 4548 in the header file) uses the overloaded `<` operator to determine whether one `String` object is greater than or equal to another. Note that the operator functions for `!=`, `>`, `<=` and `>=` are defined in the header file. The compiler inlines these definitions to eliminate the overhead of the extra function calls.

### Software Engineering Observation 11.10



By implementing member functions using previously defined member functions, the programmer reuses code to reduce the amount of code that must be written and maintained.

## Overloaded Subscript Operators

Lines 5051 in the header file declare two overloaded subscript operators (defined in Fig. 11.10, lines 95106 and 109120, respectively) one for `non-const Strings` and one for `const Strings`. When the compiler sees an expression like `string1[ 0 ]`, the compiler generates the member-function call

```
string1.operator[](0)
```

(using the appropriate version of `operator[]` based on whether the `String` is `const`). Each implementation of `operator[]` first validates the subscript to ensure that it is in range. If the subscript is out of range, each function prints an error message and terminates the program with a call to `exit`.<sup>[5]</sup> If the subscript is in range, the `non-const` version of `operator[]` returns a `char &` to the appropriate character of the `String` object; this `char &` may be used as an `lvalue` to modify the designated character of the `String` object. The `const` version of `operator[]` returns the appropriate character of the `String` object; this can be used only as an `rvalue` to read the value of the character.

<sup>[5]</sup> Again, it is more appropriate when a subscript is out of range to "throw an exception" indicating the out-of-range subscript.



Returning a non-const `char` reference from an overloaded subscript operator in a `String` class is dangerous. For example, the client could use this reference to insert a null ('`\0`') anywhere in the string.

## Overloaded Function Call Operator

Line 52 of [Fig. 11.9](#) declares the **overloaded function call operator** (defined in [Fig. 11.10](#), lines 123150). We overload this operator to select a substring from a `String`. The two integer parameters specify the start location and the length of the substring being selected from the `String`. If the start location is out of range or the substring length is negative, the operator simply returns an empty `String`. If the substring length is 0, then the substring is selected to the end of the `String` object. For example, suppose `string1` is a `String` object containing the string "AEIOU". For the expression `string1( 2, 2 )`, the compiler generates the member-function call

```
string1.operator()(2, 2)
```

When this call executes, it produces a `String` object containing the string "IO" and returns a copy of that object.

Overloading the function call operator () is powerful, because functions can take arbitrarily long and complex parameter lists. So we can use this capability for many interesting purposes. One such use of the function call operator is an alternate array-subscripting notation: Instead of using C's awkward double-square-bracket notation for pointer-based two-dimensional arrays, such as in `a[ b ][ c ]`, some programmers prefer to overload the function call operator to enable the notation `a( b, c )`. The overloaded function call operator must be a `non-static` member function. This operator is used only when the "function name" is an object of class `String`.

## String Member Function `getLength`

Line 53 in [Fig. 11.9](#) declares function `getLength` (defined in [Fig. 11.10](#), lines 153156), which returns the length of a `String`.

## Notes on Our `String` Class

At this point, you should step through the code in `main`, examine the output window and check each use of an overloaded operator. As you study the output, pay special attention to the implicit constructor calls that are generated to create temporary `String` objects throughout the program. Many of these calls introduce additional overhead into the program that can be avoided if the class provides overloaded operators that take `char *` arguments. However, additional operator functions can make the class harder to maintain, modify and debug.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 608]

## Overloading the Prefix Increment Operator

Suppose, for example, that we want to add 1 to the day in `Date` object `d1`. When the compiler sees the preincrementing expression `++d1`, the compiler generates the member-function call

```
d1.operator++()
```

The prototype for this operator function would be

```
Date &operator++();
```

If the prefix increment operator is implemented as a global function, then, when the compiler sees the expression `++d1`, the compiler generates the function call

```
operator++(d1)
```

The prototype for this operator function would be declared in the `Date` class as

```
Date &operator++(Date &);
```

## Overloading the Postfix Increment Operator

Overloading the postfix increment operator presents a challenge, because the compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions. The convention that has been adopted in C++ is that, when the compiler sees the postincrementing expression `d1++`, it generates the member-function call

```
d1.operator++(0)
```

The prototype for this function is

```
Date operator++(int)
```

The argument `0` is strictly a "dummy value" that enables the compiler to distinguish between the prefix and postfix increment operator functions.

If the postfix increment is implemented as a global function, then, when the compiler sees the expression `d1++`, the compiler generates the function call

```
operator++(d1, 0)
```

The prototype for this function would be

```
Date operator++(Date &, int);
```

Once again, the `0` argument is used by the compiler to distinguish between the prefix and postfix increment operators implemented as global functions. Note that the postfix increment operator returns `Date` objects by value, whereas the prefix increment operator returns `Date` objects by reference, because the postfix increment operator typically returns a temporary object that contains the original value of the object before the increment occurred. C++ treats such objects as rvalues, which cannot be used on the left side of an assignment. The prefix increment operator returns the actual incremented object with its new value. Such an object can be used as an lvalue in a continuing expression.

[Page 609]

### Performance Tip 11.3



The extra object that is created by the postfix increment (or decrement) operator can result in a significant performance problem especially when the operator is used in a loop. For this reason, you should use the postfix increment (or decrement) operator only when the logic of the program requires postincrementing (or postdecrementing).

Everything stated in this section for overloading prefix and postfix increment operators applies to overloading predecrement and postdecrement operators. Next, we examine a `Date` class with overloaded prefix and postfix increment operators.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 613]

Function `main` (Fig. 11.14) creates three `Date` objects (lines 1113)`d1` is initialized by default to January 1, 1900; `d2` is initialized to December 27, 1992; and `d3` is initialized to an invalid date. The `Date` constructor (defined in Fig. 11.13, lines 1114) calls `setDate` to validate the month, day and year specified. An invalid month is set to 1, an invalid year is set to 1900 and an invalid day is set to 1.

**Figure 11.13. Date class member- and friend-function definitions.**

(This item is displayed on pages 610 - 611 in the print version)

```

1 // Fig. 11.13: Date.cpp
2 // Date class member-function definitions.
3 #include <iostream>
4 #include "Date.h"
5
6 // initialize static member at file scope; one classwide copy
7 const int Date::days[] =
8 { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 31 };
9
10 // Date constructor
11 Date::Date(int m, int d, int y)
12 {
13 setDate(m, d, y);
14 } // end Date constructor
15
16 // set month, day and year
17 void Date::setDate(int mm, int dd, int yy)
18 {
19 month = (mm >= 1 && mm <= 12) ? mm : 1;
20 year = (yy >= 1900 && yy <= 2100) ? yy : 1900;
21
22 // test for a leap year
23 if (month == 2 && leapYear(year))
24 day = (dd >= 1 && dd <= 29) ? dd : 1;
25 else
26 day = (dd >= 1 && dd <= days[month]) ? dd : 1;
27 } // end function setDate
28
29 // overloaded prefix increment operator
30 Date &Date::operator++()
31 {

```

```

32 helpIncrement(); // increment date
33 return *this; // reference return to create an lvalue
34 } // end function operator++
35
36 // overloaded postfix increment operator; note that the
37 // dummy integer parameter does not have a parameter name
38 Date Date::operator++(int)
39 {
40 Date temp = *this; // hold current state of object
41 helpIncrement();
42
43 // return unincremented, saved, temporary object
44 return temp; // value return; not a reference return
45 } // end function operator++
46
47 // add specified number of days to date
48 const Date &Date::operator+=(int additionalDays)
49 {
50 for (int i = 0; i < additionalDays; i++)
51 helpIncrement();
52
53 return *this; // enables cascading
54 } // end function operator+=
55
56 // if the year is a leap year, return true; otherwise, return false
57 bool Date::leapYear(int testYear) const
58 {
59 if (testYear % 400 == 0 ||
60 (testYear % 100 != 0 && testYear % 4 == 0))
61 return true; // a leap year
62 else
63 return false; // not a leap year
64 } // end function leapYear
65
66 // determine whether the day is the last day of the month
67 bool Date::endOfDay(int testDay) const
68 {
69 if (month == 2 && leapYear(year))
70 return testDay == 29; // last day of Feb. in leap year
71 else
72 return testDay == days[month];
73 } // end function endOfDay
74
75 // function to help increment the date
76 void Date::helpIncrement()
77 {
78 // day is not end of month
79 if (!endOfDay(day))

```

```

80 day++; // increment day
81 else
82 if (month < 12) // day is end of month and month < 12
83 {
84 month++; // increment month
85 day = 1; // first day of new month
86 } // end if
87 else // last day of year
88 {
89 year++; // increment year
90 month = 1; // first month of new year
91 day = 1; // first day of new month
92 } // end else
93 } // end function helpIncrement
94
95 // overloaded output operator
96 ostream &operator<<(ostream &output, const Date &d)
97 {
98 static char *monthName[13] = { "", "January", "February",
99 "March", "April", "May", "June", "July", "August",
100 "September", "October", "November", "December" };
101 output << monthName[d.month] << ' ' << d.day << ", " << d.year;
102 return output; // enables cascading
103 } // end function operator<<

```

**Figure 11.14. Date class test program.**

(This item is displayed on page 612 in the print version)

```

1 // Fig. 11.14: fig11_14.cpp
2 // Date class test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Date.h" // Date class definition
8
9 int main()
10 {
11 Date d1; // defaults to January 1, 1900
12 Date d2(12, 27, 1992); // December 27, 1992
13 Date d3(0, 99, 8045); // invalid date
14
15 cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;

```

```

16 cout << "\n\nnd2 += 7 is " << (d2 += 7);
17
18 d3.setDate(2, 28, 1992);
19 cout << "\n\n d3 is " << d3;
20 cout << "\n++d3 is " << ++d3 << " (leap year allows 29th)";
21
22 Date d4(7, 13, 2002);
23
24 cout << "\n\nTesting the prefix increment operator:\n"
25 << " d4 is " << d4 << endl;
26 cout << "++d4 is " << ++d4 << endl;
27 cout << " d4 is " << d4;
28
29 cout << "\n\nTesting the postfix increment operator:\n"
30 << " d4 is " << d4 << endl;
31 cout << "d4++ is " << d4++ << endl;
32 cout << " d4 is " << d4 << endl;
33
34 } // end main

```

```

d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

 d3 is February 28, 1992
++d3 is February 29, 1992 (leap year allows 29th)

Testing the prefix increment operator:
 d4 is July 13, 2002
++d4 is July 14, 2002
 d4 is July 14, 2002

Testing the postfix increment operator:
 d4 is July 14, 2002
d4++ is July 14, 2002
 d4 is July 15, 2002

```

Lines 1516 of `main` output each of the constructed `Date` objects, using the overloaded stream insertion operator (defined in Fig. 11.13, lines 96103). Line 16 of `main` uses the overloaded operator `+=` to add

seven days to `d2`. Line 18 uses function `setDate` to set `d3` to February 28, 1992, which is a leap year. Then, line 20 preincrements `d3` to show that the date increments properly to February 29. Next, line 22 creates a `Date` object, `d4`, which is initialized with the date July 13, 2002. Then line 26 increments `d4` by 1 with the overloaded prefix increment operator. Lines 2427 output `d4` before and after the preincrement operation to confirm that it worked correctly. Finally, line 31 increments `d4` with the overloaded postfix increment operator. Lines 2932 output `d4` before and after the postincrement operation to confirm that it worked correctly.

Overloading the prefix increment operator is straightforward. The prefix increment operator (defined in Fig. 11.13, lines 3034) calls utility function `helpIncrement` (defined in Fig. 11.13, lines 7693) to increment the date. This function deals with "wraparounds" or "carries" that occur when we increment the last day of the month. These carries require incrementing the month. If the month is already 12, then the year must also be incremented and the month must be set to 1. Function `helpIncrement` uses function `endOfMonth` to increment the day correctly.

The overloaded prefix increment operator returns a reference to the current `Date` object (i.e., the one that was just incremented). This occurs because the current object, `*this`, is returned as a `Date &`. This enables a preincremented `Date` object to be used as an lvalue, which is how the built-in prefix increment operator works for fundamental types.

Overloading the postfix increment operator (defined in Fig. 11.13, lines 3845) is trickier. To emulate the effect of the postincrement, we must return an unincremented copy of the `Date` object. For example, that `int` variable `x` has the value 7, the statement

```
cout << x++ << endl;
```

outputs the original value of variable `x`. So we'd like our postfix increment operator to operate the same way on a `Date` object. On entry to `operator++`, we save the current object (`*this`) in `temp` (line 40). Next, we call `helpIncrement` to increment the current `Date` object. Then, line 44 returns the unincremented copy of the object previously stored in `temp`. Note that this function cannot return a reference to the local `Date` object `temp`, because a local variable is destroyed when the function in which it is declared exits. Thus, declaring the return type to this function as `Date &` would return a reference to an object that no longer exists. Returning a reference (or a pointer) to a local variable is a common error for which most compilers will issue a warning.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 614]

Building useful, reusable classes such as `String` and `Array` takes work. However, once such classes are tested and debugged, they can be reused by you, your colleagues, your company, many companies, an entire industry or even many industries (if they are placed in public or for-sale libraries). The designers of C++ did exactly that, building class `string` (which we have been using since [Chapter 3](#)) and class template `vector` (which we introduced in [Chapter 7](#)) into standard C++. These classes are available to anyone building applications with C++. As you will see in [Chapter 23](#), Standard Template Library (STL), the C++ Standard Library provides several predefined class templates for use in your programs.

To close this chapter, we redo our `String` ([Figs. 11.911.11](#)) example, using the standard C++ `string` class. We rework our example to demonstrate similar functionality provided by standard class `string`. We also demonstrate three member functions of standard class `string`: `empty`, `substr` and `at` that were not part of our `String` example. Function `empty` determines whether a `string` is empty, function `substr` returns a `string` that represents a portion of an existing `string` and function `at` returns the character at a specific index in a `string` (after checking that the index is in range). [Chapter 18](#) presents class `string` in detail.

## Standard Library Class `string`

The program of [Fig. 11.15](#) reimplements the program of [Fig. 11.11](#), using standard class `string`. As you will see in this example, class `string` provides all the functionality of our class `String` presented in [Figs. 11.911.10](#). Class `string` is defined in header `<string>` (line 7) and belongs to namespace `std` (line 8).

### Figure 11.15. Standard Library class `string`.

(This item is displayed on pages 614 - 616 in the print version)

```

1 // Fig. 11.15: fig11_15.cpp
2 // Standard Library string class test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string s1("happy");
13 string s2(" birthday");
14 string s3;
15
16 // test overloaded equality and relational operators
17 cout << "s1 is \" " << s1 << "\"; s2 is \" " << s2
18 << "\"; s3 is \" " << s3 << '\"'
19 << "\n\nThe results of comparing s2 and s1:"
20 << "\ns2 == s1 yields " << (s2 == s1 ? "true" : "false")
21 << "\ns2 != s1 yields " << (s2 != s1 ? "true" : "false")
22 << "\ns2 > s1 yields " << (s2 > s1 ? "true" : "false")
23 << "\ns2 < s1 yields " << (s2 < s1 ? "true" : "false")
24 << "\ns2 >= s1 yields " << (s2 >= s1 ? "true" : "false")
25 << "\ns2 <= s1 yields " << (s2 <= s1 ? "true" : "false");
26
27 // test string member-function empty
28 cout << "\n\nTesting s3.empty():" << endl;
29
30 if (s3.empty())
31 {
32 cout << "s3 is empty; assigning s1 to s3;" << endl;
33 s3 = s1; // assign s1 to s3
34 cout << "s3 is \" " << s3 << "\"";
35 } // end if
36
37 // test overloaded string concatenation operator
38 cout << "\n\ns1 += s2 yields s1 = ";
39 s1 += s2; // test overloaded concatenation
40 cout << s1;
41
42 // test overloaded string concatenation operator with C-style string
43 cout << "\n\ns1 += \" to you\" yields" << endl;
44 s1 += " to you";
45 cout << "s1 = " << s1 << "\n\n";
46
47 // test string member function substr
48 cout << "The substring of s1 starting at location 0 for\n"

```

```

49 << "14 characters, s1.substr(0, 14), is:\n"
50 << s1.substr(0, 14) << "\n\n";
51
52 // test substr "to-end-of-string" option
53 cout << "The substring of s1 starting at\n"
54 << "location 15, s1.substr(15), is:\n"
55 << s1.substr(15) << endl;
56
57 // test copy constructor
58 string *s4Ptr = new string(s1);
59 cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
60
61 // test assignment (=) operator with self-assignment
62 cout << "assigning *s4Ptr to *s4Ptr" << endl;
63 *s4Ptr = *s4Ptr;
64 cout << "*s4Ptr = " << *s4Ptr << endl;
65
66 // test destructor
67 delete s4Ptr;
68
69 // test using subscript operator to create lvalue
70 s1[0] = 'H';
71 s1[6] = 'B';
72 cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
73 << s1 << "\n\n";
74
75 // test subscript out of range with string member function "at"
76 cout << "Attempt to assign 'd' to s1.at(30) yields:" << endl;
77 s1.at(30) = 'd'; // ERROR: subscript out of range
78 return 0;
79 } // end main

```

s1 is "happy"; s2 is " birthday"; s3 is "

The results of comparing s2 and s1:

```

s2 == s1 yields false
s2 != s1 yields true
s2 > s1 yields false
s2 < s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

```

Testing s3.empty():

```

s3 is empty; assigning s1 to s3;
s3 is "happy"

```

```

s1 += s2 yields s1 = happy birthday

```

```
s1 += " to you" yields
s1 = happy birthday to you
```

The substring of s1 starting at location 0 for 14 characters, s1.substr(0, 14), is:  
 happy birthday

The substring of s1 starting at location 15, s1.substr(15), is:  
 to you

```
*s4Ptr = happy birthday to you
```

```
assigning *s4Ptr to *s4Ptr
*s4Ptr = happy birthday to you
```

```
s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you
```

Attempt to assign 'd' to s1.at( 30 ) yields:

```
abnormal program termination
```

[Page 616]

Lines 1214 create three string objects: s1 is initialized with the literal "happy", s2 is initialized with the literal " birthday" and s3 uses the default string constructor to create an empty string. Lines 1718 output these three objects, using cout and operator <<, which the string class designers overloaded to handle string objects. Then lines 1925 show the results of comparing s2 to s1 by using class string's overloaded equality and relational operators.

Our class String (Figs. 11.911.10) provided an overloaded operator ! that tested a String to determine whether it was empty. Standard class string does not provide this functionality as an overloaded operator; instead, it provides member function empty, which we demonstrate on line 30. Member function empty returns true if the string is empty; otherwise, it returns false.

Line 33 demonstrates class string's overloaded assignment operator by assigning s1 to s3. Line 34 outputs s3 to demonstrate that the assignment worked correctly.

Line 39 demonstrates class `string`'s overloaded `+=` operator for string concatenation. In this case, the contents of `s2` are appended to `s1`. Then line 40 outputs the resulting string that is stored in `s1`. Line 44 demonstrates that a C-style string literal can be appended to a `string` object by using operator `+=`. Line 45 displays the result.

Our class `String` ([Figs. 11.911.10](#)) provided overloaded `operator()` to obtain substrings. Standard class `string` does not provide this functionality as an overloaded operator; instead, it provides member function `substr` (lines 50 and 55). The call to `substr` in line 50 obtains a 14-character substring (specified by the second argument) of `s1` starting at position 0 (specified by the first argument). The call to `substr` in line 55 obtains a substring starting from position 15 of `s1`. When the second argument is not specified, `substr` returns the remainder of the `string` on which it is called.

Line 58 dynamically allocates a `string` object and initializes it with a copy of `s1`. This results in a call to class `string`'s copy constructor. Line 63 uses class `string`'s overloaded `=` operator to demonstrate that it handles self-assignment properly.

Lines 7071 used class `string`'s overloaded `[ ]` operator to create lvalues that enable new characters to replace existing characters in `s1`. Line 73 outputs the new value of `s1`. In our class `String` ([Figs. 11.911.10](#)), the overloaded `[ ]` operator performed bounds checking to determine whether the subscript it received as an argument was a valid subscript in the string. If the subscript was invalid, the operator printed an error message and terminated the program. Standard class `string`'s overloaded `[ ]` operator does not perform any bounds checking. Therefore, the programmer must ensure that operations using standard class `string`'s overloaded `[ ]` operator do not accidentally manipulate elements outside the bounds of the `string`. Standard class `string` does provide bounds checking in its member function `at`, which "throws an exception" if its argument is an invalid subscript. By default, this causes a C++ program to terminate.<sup>[6]</sup> If the subscript is valid, function `at` returns the character at the specified location as a modifiable lvalue or an unmodifiable lvalue (i.e., a `const` reference), depending on the context in which the call appears. Line 77 demonstrates a call to function `at` with an invalid subscript.

[6] Again, [Chapter 16](#), Exception Handling, demonstrates how to "catch" such exceptions.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 618]

## Accidentally Using a Single-Argument Constructor as a Conversion Constructor

The program (Fig. 11.16) uses the `Array` class of Figs. 11.611.7 to demonstrate an improper implicit conversion.

**Figure 11.16. Single-argument constructors and implicit conversions.**

```

1 // Fig. 11.16: Fig11_16.cpp
2 // Driver for simple class Array.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Array.h"
8
9 void outputArray(const Array &); // prototype
10
11 int main()
12 {
13 Array integers1(7); // 7-element array
14 outputArray(integers1); // output Array integers1
15 outputArray(3); // convert 3 to an Array and output Array's contents
16 return 0;
17 } // end main
18
19 // print Array contents
20 void outputArray(const Array &arrayToOutput)
21 {
22 cout << "The Array received has " << arrayToOutput.getSize()
23 << " elements. The contents are:\n" << arrayToOutput << endl;
24 } // end outputArray

```

The Array received has 7 elements. The contents are:

|   |   |   |   |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 |   |

The Array received has 3 elements. The contents are:

|   |   |   |
|---|---|---|
| 0 | 0 | 0 |
|---|---|---|

Line 13 in `main` instantiates `Array` object `integers1` and calls the single argument constructor with the `int` value 7 to specify the number of elements in the `Array`. Recall from Fig. 11.7 that the `Array` constructor that receives an `int` argument initializes all the array elements to 0. Line 14 calls function `outputArray` (defined in lines 2024), which receives as its argument a `const Array &` to an `Array`. The function outputs the number of elements in its `Array` argument and the contents of the `Array`. In this case, the size of the `Array` is 7, so seven 0s are output.

Line 15 calls function `outputArray` with the `int` value 3 as an argument. However, this program does not contain a function called `outputArray` that takes an `int` argument. So, the compiler determines whether class `Array` provides a conversion constructor that can convert an `int` into an `Array`. Since any constructor that receives a single argument is considered to be a conversion constructor, the compiler assumes the `Array` constructor that receives a single `int` is a conversion constructor and uses it to convert the argument 3 into a temporary `Array` object that contains three elements. Then, the compiler passes the temporary `Array` object to function `outputArray` to output the `Array`'s contents. Thus, even though we do not explicitly provide an `outputArray` function that receives an `int` argument, the compiler is able to compile line 15. The output shows the contents of the three-element `Array` containing 0s.

---

[Page 619]

## Preventing Accidental Use of a Single-Argument Constructor as a Conversion Constructor

C++ provides the keyword `explicit` to suppress implicit conversions via conversion constructors when such conversions should not be allowed. A constructor that is declared `explicit` cannot be used in an implicit conversion. Figure 11.17 declares an `explicit` constructor in class `Array`. The only modification to `Array.h` was the addition of the keyword `explicit` to the declaration of the single-argument constructor at line 15. No modifications are required to the source-code file containing class `Array`'s member-function definitions.

---

[Page 620]

### Figure 11.17. Array class definition with explicit constructor.

(This item is displayed on page 619 in the print version)

```

1 // Fig. 11.17: Array.h
2 // Array class for storing arrays of integers.
3 #ifndef ARRAY_H
4 #define ARRAY_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 class Array
11 {
12 friend ostream &operator<<(ostream &, const Array &);
13 friend istream &operator>>(istream &, Array &);
14 public:
15 explicit Array(int = 10); // default constructor
16 Array(const Array &); // copy constructor
17 ~Array(); // destructor
18 int getSize() const; // return size
19
20 const Array &operator=(const Array &); // assignment operator
21 bool operator==(const Array &) const; // equality operator
22
23 // inequality operator; returns opposite of == operator
24 bool operator!=(const Array &right) const
25 {
26 return ! (*this == right); // invokes Array::operator==
27 } // end function operator!=
28
29 // subscript operator for non-const objects returns lvalue
30 int &operator[](int);
31
32 // subscript operator for const objects returns rvalue
33 const int &operator[](int) const;
34 private:
35 int size; // pointer-based array size
36 int *ptr; // pointer to first element of pointer-based array
37 }; // end class Array
38
39 #endif

```

Figure 11.18 presents a slightly modified version of the program in Fig. 11.16. When this program is compiled, the compiler produces an error message indicating that the integer value passed to `outputArray` at line 15 cannot be converted to a `const Array &`. The compiler error message is shown in the output window. Line 16 demonstrates how the explicit constructor can be used to create a temporary `Array` of 3 elements and pass it to function `outputArray`.

**Figure 11.18. Demonstrating an explicit constructor.**

```

1 // Fig. 11.18: Fig11_18.cpp
2 // Driver for simple class Array.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Array.h"
8
9 void outputArray(const Array &); // prototype
10
11 int main()
12 {
13 Array integers1(7); // 7-element array
14 outputArray(integers1); // output Array integers1
15 outputArray(3); // convert 3 to an Array and output Array's contents
16 outputArray(Array(3)); // explicit single-argument constructor call
17 return 0;
18 } // end main
19
20 // print array contents
21 void outputArray(const Array &arrayToOutput)
22 {
23 cout << "The Array received has " << arrayToOutput.getSize()
24 << " elements. The contents are:\n" << arrayToOutput << endl;
25 } // end outputArray

```

```

c:\cpphttp5_examples\ch11\Fig11_17_18\Fig11_18.cpp(15) : error C2664:
 'outputArray' : cannot convert parameter 1 from 'int' to 'const Array &'
 Reason: cannot convert from 'int' to 'const Array'
 Constructor for class 'Array' is declared 'explicit'

```

**Common Programming Error 11.10**

Attempting to invoke an explicit constructor for an implicit conversion is a compilation error.

## Common Programming Error 11.11



Using the `explicit` keyword on data members or member functions other than a single-argument constructor is a compilation error.

---

[Page 621]

## Error-Prevention Tip 11.3



Use the `explicit` keyword on single-argument constructors that should not be used by the compiler to perform implicit conversions.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 621 (continued)]

## 11.15. Wrap-Up

In this chapter, you learned how to build more robust classes by defining overloaded operators that enable programmers to treat objects of your classes as if they were fundamental C++ data types. We presented the basic concepts of operator overloading, as well as several restrictions that the C++ standard places on overloaded operators. You learned reasons for implementing overloaded operators as member functions or as global functions. We discussed the differences between overloading unary and binary operators as member functions and global functions. With global functions, we showed how to enable objects of our classes to be input and output using the overloaded stream extraction and stream insertion operators, respectively. We showed a special syntax that is required to differentiate between the prefix and postfix versions of the increment (++) operator. We also demonstrated standard C++ class `string`, which makes extensive use of overloaded operators to create a robust, reusable class that can replace C-style, pointer-based strings. Finally, you learned how to use keyword `explicit` to prevent the compiler from using a single-argument constructor to perform implicit conversions. In the next chapter, we continue our discussion of classes by introducing a form of software reuse called inheritance. We will see that classes often share common attributes and behaviors. In such cases, it is possible to define those attributes and behaviors in a common "base" class and "inherit" those capabilities into new class definitions.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 622]

- You cannot change the precedence and associativity of an operator by overloading.
- You cannot change the "arity" of an operator (i.e., the number of operands an operator takes).
- You cannot create new operators; only existing operators can be overloaded.
- You cannot change the meaning of how an operator works on objects of fundamental types.
- Overloading an assignment operator and an addition operator for a class does not imply that the `+=` operator is also overloaded. Such behavior can be achieved only by explicitly overloading operator `+=` for that class.
- Operator functions can be member functions or global functions; global functions are often made friends for performance reasons. Member functions use the `this` pointer implicitly to obtain one of their class object arguments (the left operand for binary operators). Arguments for both operands of a binary operator must be explicitly listed in a global function call.
- When overloading `( )`, `[ ]`, `->` or any of the assignment operators, the operator overloading function must be declared as a class member. For the other operators, the operator overloading functions can be class members or global functions.
- When an operator function is implemented as a member function, the leftmost (or only) operand must be an object (or a reference to an object) of the operator's class.
- If the left operand must be an object of a different class or a fundamental type, this operator function must be implemented as a global function.
- A global operator function can be made a `friend` of a class if that function must access `private` or `protected` members of that class directly.
- The overloaded stream insertion operator (`<<`) is used in an expression in which the left operand has type `ostream &`. For this reason, it must be overloaded as a global function. To be a member function, operator `<<` would have to be a member of the `ostream` class, but this is not possible, since we are not allowed to modify C++ Standard Library classes. Similarly, the overloaded stream extraction operator (`>>`) must be a global function.
- Another reason to choose a global function to overload an operator is to enable the operator to be commutative.
- When used with `cin` and `strings`, `setw` restricts the number of characters read to the number of characters specified by its argument.
- `istream` member function `ignore` discards the specified number of characters in the input stream (one character by default).
- Overloaded input and output operators are declared as `friends` if they need to access non-public class members directly for performance reasons.
- A unary operator for a class can be overloaded as a `non-static` member function with no arguments or as a global function with one argument; that argument must be either an object of the class or a reference to an object of the class.
- Member functions that implement overloaded operators must be `non-static` so that they can access the `non-static` data in each object of the class.
- A binary operator can be overloaded as a `non-static` member function with one argument or as a global function with two arguments (one of those arguments must be either a class object or a reference to a class object).
- A copy constructor initializes a new object of a class by copying the members of an existing object of

that class. When objects of a class contain dynamically allocated memory, the class should provide a copy constructor to ensure that each copy of an object has its own separate copy of the dynamically allocated memory. Typically, such a class would also provide a destructor and an overloaded assignment operator.

### [Page 623]

- The implementation of member function `operator=` should test for self-assignment, in which an object is being assigned to itself.
- The compiler calls the `const` version of `operator[ ]` when the subscript operator is used on a `const` object and calls the non-`const` version of the operator when it is used on a non-`const` object.
- The array subscript operator (`[]`) is not restricted for use with arrays. It can be used to select elements from other types of container classes. Also, with overloading, the index values no longer need to be integers; characters or strings could be used, for example.
- The compiler cannot know in advance how to convert among user-defined types, and between user-defined types and fundamental types, so the programmer must specify how to do this. Such conversions can be performed with conversion constructorssingle-argument constructors that turn objects of other types (including fundamental types) into objects of a particular class.
- A conversion operator (also called a cast operator) can be used to convert an object of one class into an object of another class or into an object of a fundamental type. Such a conversion operator must be a non-static member function. Overloaded cast-operator functions can be defined for converting objects of user-defined types into fundamental types or into objects of other user-defined types.
- An overloaded cast operator function does not specify a return typethe return type is the type to which the object is being converted.
- One of the nice features of cast operators and conversion constructors is that, when necessary, the compiler can call these functions implicitly to create temporary objects.
- Any single-argument constructor can be thought of as a conversion constructor.
- Overloading the function call operator (`()`) is powerful, because functions can take arbitrarily long and complex parameter lists.
- The prefix and postfix increment and decrement operator can all be overloaded.
- To overload the increment operator to allow both preincrement and postincrement usage, each overloaded operator function must have a distinct signature, so that the compiler will be able to determine which version of `++` is intended. The prefix versions are overloaded exactly as any other prefix unary operator would be. Providing a unique signature to the postfix increment operator is accomplished by providing a second argument, which must be of type `int`. This argument is not supplied in the client code. It is used implicitly by the compiler to distinguish between the prefix and postfix versions of the increment operator.
- Standard class `string` is defined in header `<string>` and belongs to namespace `std`.
- Class `string` provides many overloaded operators, including equality, relational, assignment, addition assignment (for concatenation) and subscript operators.
- Class `string` provides member function `empty`, which returns `true` if the `string` is empty; otherwise, it returns `false`.
- Standard class `string` member function `substr` obtains a substring of a length specified by the second argument, starting at the position specified by the first argument. When the second argument is not specified, `substr` returns the remainder of the string on which it is called.
- Class `string`'s overloaded `[]` operator does not perform any bounds checking. Therefore, the programmer must ensure that operations using standard class `string`'s overloaded `[]` operator do not accidentally manipulate elements outside the bounds of the `string`.
- Standard class `string` provides bounds checking with member function `at`, which "throws an

exception" if its argument is an invalid subscript. By default, this causes a C++ program to terminate. If the subscript is valid, function `at` returns the character at the specified location as an *lvalue* or an *rvalue*, depending on the context in which the call appears.

---

[Page 624]

- C++ provides the keyword `explicit` to suppress implicit conversions via conversion constructors when such conversions should not be allowed. A constructor that is declared `explicit` cannot be used in an implicit conversion.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 624 (continued)]

## Terminology

"arity" of an operator

Array class

assignment-operator functions

associativity not changed by overloading

cast operator function

commutative operation

const version of operator[ ]

conversion between fundamental and class types

conversion constructor

conversion operator

copy constructor

empty member function of string

explicit constructor

function call operator ( )

global function to overload an operator

ignore member function of istream

implicit user-defined conversions

lvalue ("left value")

operator function

operator keyword

operator overloading

operator!

operator!=

operator()

operator+

operator++

operator++( int )

operator<

operator<<

operator=

operator==

operator>=

operator>>

operator[]

overloadable operators

overloaded ! operator

overloaded != operator

overloaded () operator

overloaded + operator

overloaded ++ operator

overloaded ++( int ) operator

overloaded += operator

overloaded < operator

overloaded << operator

overloaded <= operator

overloaded == operator

overloaded > operator

overloaded >= operator

overloaded >> operator

overloaded assignment (=) operator

overloaded [ ] operator

overloaded stream insertion and stream extraction operators

overloading a binary operator

overloading a unary operator

self-assignment

string (standard C++ class)

string concatenation

substr member function of string

substring

user-defined conversion

user-defined type

 PREV

NEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 625]

**11.4**

(True/False) In C++, only existing operators can be overloaded.

**11.5**

How does the precedence of an overloaded operator in C++ compare with the precedence of the original operator?

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 625 (continued)]

## Answers to Self-Review Exercises

- 11.1** a) operator overloading. b) `operator.` c) assignment (`=`), address (`&`), comma (`,`). d) precedence, associativity, "arity."
- 11.2** Operator `>>` is both the right-shift operator and the stream extraction operator, depending on its context. Operator `<<` is both the left-shift operator and the stream insertion operator, depending on its context.
- 11.3** For operator overloading: It would be the name of a function that would provide an overloaded version of the `/` operator for a specific class.
- 11.4** True.
- 11.5** The precedence is identical.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 626]

for an array of objects, overload the function call operator to allow the alternate form

```
chessBoard(row, column)
```

Create a class `DoubleSubscriptedArray` that has similar features to class `Array` in [Figs. 11.611.7](#). At construction time, the class should be able to create an array of any number of rows and any number of columns. The class should supply `operator()` to perform double-subscripting operations. For example, in a 3-by-5 `DoubleSubscriptedArray` called `a`, the user could write `a( 1, 3 )` to access the element at row 1 and column 3. Remember that `operator()` can receive any number of arguments (see class `String` in [Figs. 11.911.10](#) for an example of `operator()`). The underlying representation of the double-subscripted array should be a single-subscripted array of integers with `rows * columns` number of elements. Function `operator()` should perform the proper pointer arithmetic to access each element of the array. There should be two versions of `operator()` one that returns `int &` (so that an element of a `DoubleSubscriptedArray` can be used as an lvalue) and one that returns `const int &` (so that an element of a `const DoubleSubscriptedArray` can be used only as an rvalue). The class should also provide the following operators: `==`, `!=`, `=`, `<<` (for outputting the array in row and column format) and `>>` (for inputting the entire array contents).

## 11.12

Overload the subscript operator to return the largest element of a collection, the second largest, the third largest, and so on.

## 11.13

Consider class `Complex` shown in [Figs. 11.1911.21](#). The class enables operations on so-called complex numbers. These are numbers of the form `realPart + imaginaryPart * i`, where `i` has the value

$$\sqrt{-1}$$

a.

Modify the class to enable input and output of complex numbers through the overloaded `>>` and `<<` operators, respectively (you should remove the `print` function from the class).

b.

Overload the multiplication operator to enable multiplication of two complex numbers as in algebra.

c.

Overload the == and != operators to allow comparisons of complex numbers.

**Figure 11.19. Complex class definition.**

```
1 // Fig. 11.19: Complex.h
2 // Complex class definition.
3 #ifndef COMPLEX_H
4 #define COMPLEX_H
5
6 class Complex
7 {
8 public:
9 Complex(double = 0.0, double = 0.0); // constructor
10 Complex operator+(const Complex &) const; // addition
11 Complex operator-(const Complex &) const; // subtraction
12 void print() const; // output
13 private:
14 double real; // real part
15 double imaginary; // imaginary part
16 }; // end class Complex
17
18 #endif
```

---

[Page 627]

**Figure 11.20. Complex class member-function definitions.**

```
1 // Fig. 11.20: Complex.cpp
2 // Complex class member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 #include "Complex.h" // Complex class definition
7
8 // Constructor
9 Complex::Complex(double realPart, double imaginaryPart)
10 : real(realPart),
11 imaginary(imaginaryPart)
12 {
13 // empty body
14 } // end Complex constructor
15
16 // addition operator
17 Complex Complex::operator+(const Complex &operand2) const
18 {
19 return Complex(real + operand2.real,
20 imaginary + operand2.imaginary);
21 } // end function operator+
22
23 // subtraction operator
24 Complex Complex::operator-(const Complex &operand2) const
25 {
26 return Complex(real - operand2.real,
27 imaginary - operand2.imaginary);
28 } // end function operator-
29
30 // display a Complex object in the form: (a, b)
31 void Complex::print() const
32 {
33 cout << '(' << real << ", " << imaginary << ')';
34 } // end function print
```

**Figure 11.21. Complex numbers.**

(This item is displayed on pages 627 - 628 in the print version)

```
1 // Fig. 11.21: fig11_21.cpp
2 // Complex class test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Complex.h"
8
9 int main()
10 {
11 Complex x;
12 Complex y(4.3, 8.2);
13 Complex z(3.3, 1.1);
14
15 cout << "x: ";
16 x.print();
17 cout << "\ny: ";
18 y.print();
19 cout << "\nz: ";
20 z.print();
21
22 x = y + z;
23 cout << "\nx = y + z:" << endl;
24 x.print();
25 cout << " = ";
26 y.print();
27 cout << " + ";
28 z.print();
29
30 x = y - z;
31 cout << "\nx = y - z:" << endl;
32 x.print();
33 cout << " = ";
34 y.print();
35 cout << " - ";
36 z.print();
37 cout << endl;
38 return 0;
39 } // end main
```

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)
```

---

[Page 628]

### 11.14

A machine with 32-bit integers can represent integers in the range of approximately 2 billion to +2 billion. This fixed-size restriction is rarely troublesome, but there are applications in which we would like to be able to use a much wider range of integers. This is what C++ was built to do, namely, create powerful new data types. Consider class `HugeInt` of Figs. 11.2211.24. Study the class carefully, then answer the following:

a.

Describe precisely how it operates.

b.

What restrictions does the class have?

c.

Overload the \* multiplication operator.

d.

Overload the / division operator.

e.

Overload all the relational and equality operators.

[Note: We do not show an assignment operator or copy constructor for class HugeInteger, because the assignment operator and copy constructor provided by the compiler are capable of copying the entire array data member properly.]

---

[Page 629]

**Figure 11.22. HugeInt class definition.**

```
1 // Fig. 11.22: Hugeint.h
2 // HugeInt class definition.
3 #ifndef HUGEINT_H
4 #define HUGEINT_H
5
6 #include <iostream>
7 using std::ostream;
8
9 class HugeInt
10 {
11 friend ostream &operator<<(ostream &, const HugeInt &);
12 public:
13 HugeInt(long = 0); // conversion/default constructor
14 HugeInt(const char *); // conversion constructor
15
16 // addition operator; HugeInt + HugeInt
17 HugeInt operator+(const HugeInt &) const;
18
19 // addition operator; HugeInt + int
20 HugeInt operator+(int) const;
21
22 // addition operator;
23 // HugeInt + string that represents large integer value
24 HugeInt operator+(const char *) const;
25 private:
26 short integer[30];
27 }; // end class HugeInt
28
29 #endif
```

## 11.15

Create a class RationalNumber (fractions) with the following capabilities:

Create a constructor that prevents a 0 denominator in a fraction, reduces or simplifies fractions that are not in reduced form and avoids negative denominators.

**b.**

Overload the addition, subtraction, multiplication and division operators for this class.

**c.**

Overload the relational and equality operators for this class.

## 11.16

Study the C string-handling library functions and implement each of the functions as part of class `String` ([Figs. 11.911.10](#)). Then, use these functions to perform text manipulations.

## 11.17

Develop class `Polynomial`. The internal representation of a `Polynomial` is an array of terms. Each term contains a coefficient and an exponent. The term

$$2x^4$$

has the coefficient 2 and the exponent 4. Develop a complete class containing proper constructor and destructor functions as well as set and get functions. The class should also provide the following overloaded operator capabilities:

**a.**

Overload the addition operator (+) to add two `Polynomials`.

**b.**

Overload the subtraction operator (-) to subtract two `Polynomials`.

**c.**

Overload the assignment operator to assign one `Polynomial` to another.

**d.**

Overload the multiplication operator (\*) to multiply two `Polynomials`.

Overload the addition assignment operator (`+=`), subtraction assignment operator (`-=`), and multiplication assignment operator (`*=`).

[Page 630]

**Figure 11.23. HugeInt class member-function and friend-function definitions.**

(This item is displayed on pages 630 - 631 in the print version)

```
1 // Fig. 11.23: Hugeint.cpp
2 // HugeInt member-function and friend-function definitions.
3 #include <cctype> // isdigit function prototype
4 #include <cstring> // strlen function prototype
5 #include "Hugeint.h" // HugeInt class definition
6
7 // default constructor; conversion constructor that converts
8 // a long integer into a HugeInt object
9 HugeInt::HugeInt(long value)
10 {
11 // initialize array to zero
12 for (int i = 0; i <= 29; i++)
13 integer[i] = 0;
14
15 // place digits of argument into array
16 for (int j = 29; value != 0 && j >= 0; j--)
17 {
18 integer[j] = value % 10;
19 value /= 10;
20 } // end for
21 } // end HugeInt default/conversion constructor
22
23 // conversion constructor that converts a character string
24 // representing a large integer into a HugeInt object
25 HugeInt::HugeInt(const char *string)
26 {
27 // initialize array to zero
28 for (int i = 0; i <= 29; i++)
29 integer[i] = 0;
30
31 // place digits of argument into array
32 int length = strlen(string);
33
34 for (int j = 30 - length, k = 0; j <= 29; j++, k++)
35
36 if (isdigit(string[k]))
37 integer[j] = string[k] - '0';
```

```
38 } // end HugeInt conversion constructor
39
40 // addition operator; HugeInt + HugeInt
41 HugeInt HugeInt::operator+(const HugeInt &op2) const
42 {
43 HugeInt temp; // temporary result
44 int carry = 0;
45
46 for (int i = 29; i >= 0; i--)
47 {
48 temp.integer[i] =
49 integer[i] + op2.integer[i] + carry;
50
51 // determine whether to carry a 1
52 if (temp.integer[i] > 9)
53 {
54 temp.integer[i] %= 10; // reduce to 0-9
55 carry = 1;
56 } // end if
57 else // no carry
58 carry = 0;
59 } // end for
60
61 return temp; // return copy of temporary object
62 } // end function operator+
63
64 // addition operator; HugeInt + int
65 HugeInt HugeInt::operator+(int op2) const
66 {
67 // convert op2 to a HugeInt, then invoke
68 // operator+ for two HugeInt objects
69 return *this + HugeInt(op2);
70 } // end function operator+
71
72 // addition operator;
73 // HugeInt + string that represents large integer value
74 HugeInt HugeInt::operator+(const char *op2) const
75 {
76 // convert op2 to a HugeInt, then invoke
77 // operator+ for two HugeInt objects
78 return *this + HugeInt(op2);
79 } // end operator+
80
81 // overloaded output operator
82 ostream& operator<<(ostream &output, const HugeInt &num)
83 {
84 int i;
```

```

86 for (i = 0; (num.integer[i] == 0) && (i <= 29); i++)
87 ; // skip leading zeros
88
89 if (i == 30)
90 output << 0;
91 else
92
93 for (; i <= 29; i++)
94 output << num.integer[i];
95
96 return output;
97 } // end function operator<<

```

---

[Page 631]

## 11.18

In the program of Figs. 11.311.5, Fig. 11.4 contains the comment "overloaded stream insertion operator; cannot be a member function if we would like to invoke it with cout << somePhoneNumber;." Actually, the stream insertion operator could be a PhoneNumber class member function if we were willing to invoke it either as somePhoneNumber.operator<<( cout ); or as somePhoneNumber << cout;. Rewrite the program of Fig. 11.5 with the overloaded stream insertion operator<< as a member function and try the two preceding statements in the program to demonstrate that they work.

---

[Page 632]

## Figure 11.24. Huge integers.

```

1 // Fig. 11.24: fig11_24.cpp
2 // HugeInt test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Hugeint.h"
8
9 int main()
10 {
11 HugeInt n1(7654321);
12 HugeInt n2(7891234);
13 HugeInt n3("99999999999999999999999999999999");

```

```
14 HugeInt n4("1");
15 HugeInt n5;
16
17 cout << "n1 is " << n1 << "\nn2 is " << n2
18 << "\nn3 is " << n3 << "\nn4 is " << n4
19 << "\nn5 is " << n5 << "\n\n";
20
21 n5 = n1 + n2;
22 cout << n1 << " + " << n2 << " = " << n5 << "\n\n";
23
24 cout << n3 << " + " << n4 << "\n= " << (n3 + n4) << "\n\n";
25
26 n5 = n1 + 9;
27 cout << n1 << " + " << 9 << " = " << n5 << "\n\n";
28
29 n5 = n2 + "10000";
30 cout << n2 << " + " << "10000" << " = " << n5 << endl;
31 return 0;
32 } // end main
```

$$7654321 + 7891234 = 15545555$$

$$7654321 + 9 = 7654330$$

$$7891234 + 10000 = 7901234$$

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 634]

## Outline

### [12.1 Introduction](#)

### [12.2 Base Classes and Derived Classes](#)

### [12.3 protected Members](#)

### [12.4 Relationship between Base Classes and Derived Classes](#)

#### [12.4.1 Creating and Using a CommissionEmployee Class](#)

#### [12.4.2 Creating a BasePlusCommissionEmployee Class Without Using Inheritance](#)

#### [12.4.3 Creating a CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy](#)

#### [12.4.4 CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy Using protected Data](#)

#### [12.4.5 CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy Using private Data](#)

### [12.5 Constructors and Destructors in Derived Classes](#)

### [12.6 public, protected and private Inheritance](#)

### [12.7 Software Engineering with Inheritance](#)

### [12.8 Wrap-Up](#)

## [Summary](#)

## [Terminology](#)

## Self-Review Exercises

### Answers to Self-Review Exercises

#### Exercises

 PREV

NEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 635]

Experience in building software systems indicates that significant amounts of code deal with closely related special cases. When programmers are preoccupied with special cases, the details can obscure the big picture. With object-oriented programming, programmers focus on the commonalities among objects in the system rather than on the special cases.

We distinguish between the **is-a relationship** and the has-a relationship. The is-a relationship represents inheritance. In an is-a relationship, an object of a derived class also can be treated as an object of its base class for example, a car is a vehicle, so any properties and behaviors of a vehicle are also properties of a car. By contrast, the *has-a* relationship represents composition. (Composition was discussed in [Chapter 10](#).) In a has-a relationship, an object contains one or more objects of other classes as members. For example, a car includes many components it has a steering wheel, has a brake pedal, has a transmission and has many other components.

Derived-class member functions might require access to base-class data members and member functions. A derived class can access the non-private members of its base class. Base-class members that should not be accessible to the member functions of derived classes should be declared `private` in the base class. A derived class can effect state changes in `private` base-class members, but only through non-private member functions provided in the base class and inherited into the derived class.

### Software Engineering Observation 12.1



Member functions of a derived class cannot directly access private members of the base class.

### Software Engineering Observation 12.2



If a derived class could access its base class's `private` members, classes that inherit from that derived class could access that data as well. This would propagate access to what should be `private` data, and the benefits of information hiding would be lost.

One problem with inheritance is that a derived class can inherit data members and member functions it does not need or should not have. It is the class designer's responsibility to ensure that the capabilities provided by a class are appropriate for future derived classes. Even when a base-class member function is appropriate for a derived class, the derived class often requires that member function to behave in a

manner specific to the derived class. In such cases, the base-class member function can be redefined in the derived class with an appropriate implementation.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 636]

**Figure 12.1. Inheritance examples.**

| Base class | Derived classes                               |
|------------|-----------------------------------------------|
| Student    | GraduateStudent, UndergraduateStudent         |
| Shape      | Circle, Triangle, Rectangle, Sphere, Cube     |
| Loan       | CarLoan, HomeImprovementLoan,<br>MortgageLoan |
| Employee   | Faculty, Staff                                |
| Account    | CheckingAccount, SavingsAccount               |

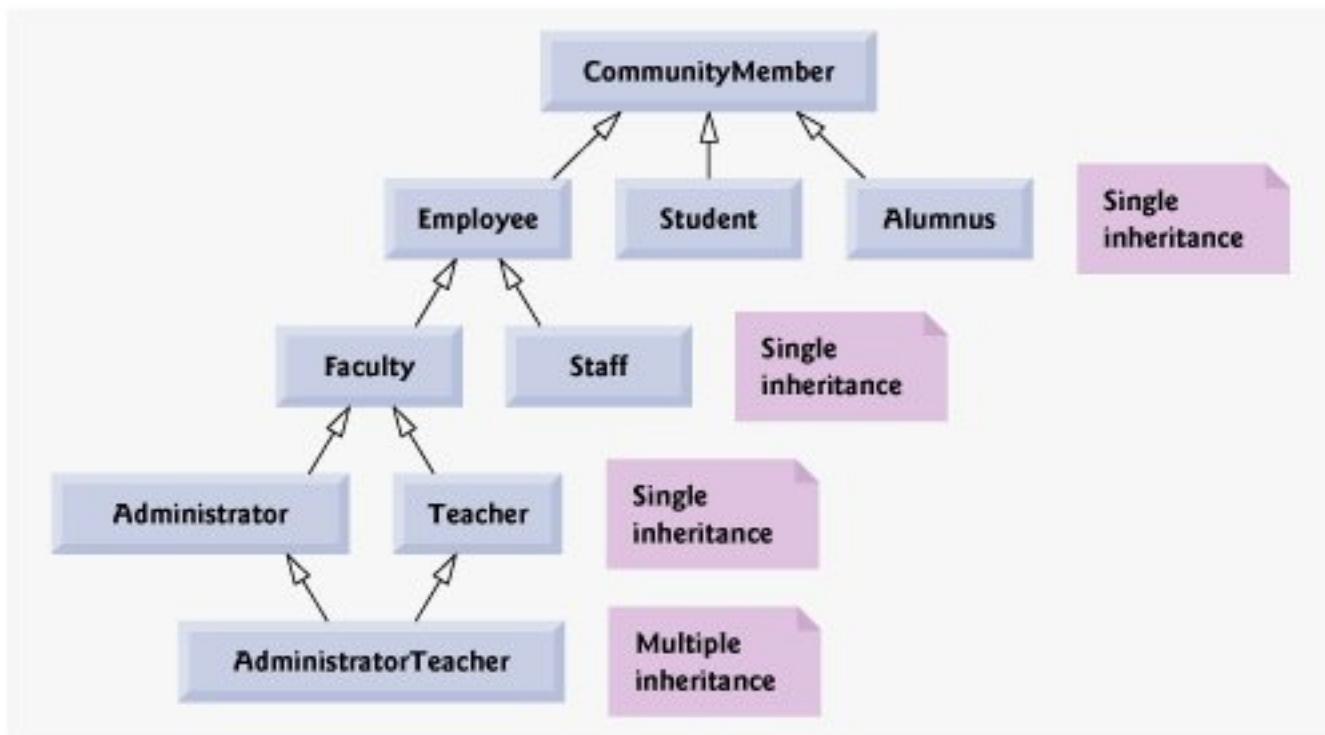
Because every derived-class object is an object of its base class, and one base class can have many derived classes, the set of objects represented by a base class typically is larger than the set of objects represented by any of its derived classes. For example, the base class `Vehicle` represents all vehicles, including cars, trucks, boats, airplanes, bicycles and so on. By contrast, derived class `Car` represents a smaller, more specific subset of all vehicles.

Inheritance relationships form treelike hierarchical structures. A base class exists in a hierarchical relationship with its derived classes. Although classes can exist independently, once they are employed in inheritance relationships, they become affiliated with other classes. A class becomes either a base class supplying members to other classes, a derived class inheriting its members from other classes, or both.

Let us develop a simple inheritance hierarchy with five levels (represented by the UML class diagram in Fig. 12.2). A university community has thousands of members.

**Figure 12.2. Inheritance hierarchy for university CommunityMembers.**

[\[View full size image\]](#)



[Page 637]

These members consist of employees, students and alumni. Employees are either faculty members or staff members. Faculty members are either administrators (such as deans and department chairpersons) or teachers. Some administrators, however, also teach classes. Note that we have used multiple inheritance to form class **AdministratorTeacher**. Also note that this inheritance hierarchy could contain many other classes. For example, students can be graduate or undergraduate students. Undergraduate students can be freshmen, sophomores, juniors and seniors.

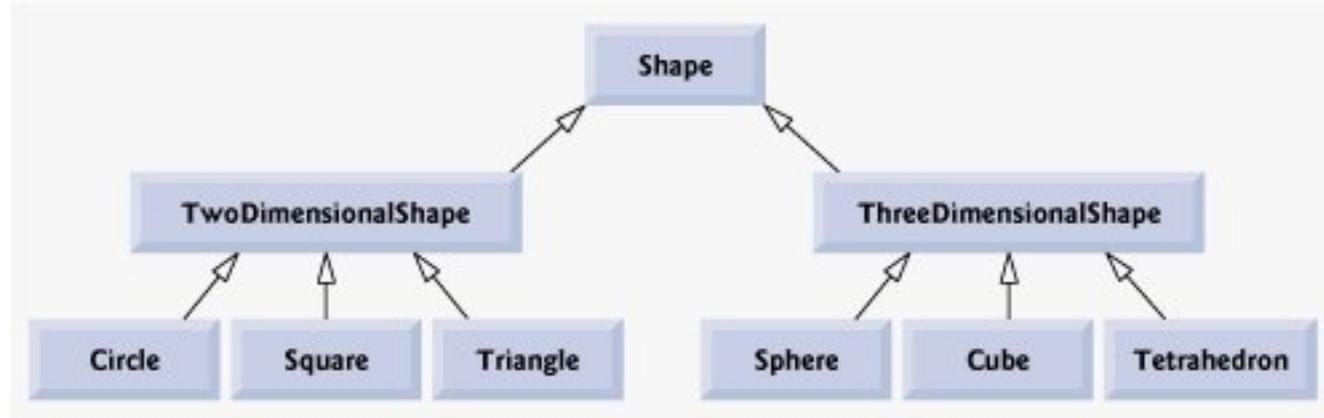
Each arrow in the hierarchy (Fig. 12.2) represents an is-a relationship. For example, as we follow the arrows in this class hierarchy, we can state "an **Employee** is a **CommunityMember**" and "a **Teacher** is a **Faculty member**." **CommunityMember** is the direct base class of **Employee**, **Student** and **Alumnus**. In addition, **CommunityMember** is an indirect base class of all the other classes in the diagram. Starting from the bottom of the diagram, the reader can follow the arrows and apply the is-a relationship to the topmost base class. For example, an **AdministratorTeacher** is an **Administrator**, is a **Faculty member**, is an **Employee** and is a **CommunityMember**.

Now consider the Shape inheritance hierarchy in Fig. 12.3. This hierarchy begins with base class **Shape**. Classes **TwoDimensionalShape** and **ThreeDimensionalShape** derive from base class **Shape**. **Shapes** are either **TwoDimensionalShapes** or **ThreeDimensionalShapes**. The third level of this hierarchy contains some more specific types of **TwoDimensionalShapes** and **ThreeDimensionalShapes**. As in Fig. 12.2, we can follow the arrows from the bottom of the diagram to the topmost base class in this class hierarchy to identify several is-a relationships. For instance, a triangle is a **TwoDimensionalShape** and is a **Shape**, while a **Sphere** is a **ThreeDimensionalShape** and is a **Shape**. Note that this hierarchy could contain many other classes, such as **Rectangles**, **Ellipses** and **trapezoids**, which

are all `TwoDimensionalShapes`.

**Figure 12.3. Inheritance hierarchy for shapes.**

[View full size image]



To specify that class `TwoDimensionalShape` (Fig. 12.3) is derived from (or inherits from) class `Shape`, class `TwoDimensionalShape` could be defined in C++ as follows:

```
class TwoDimensionalShape : public Shape
```

This is an example of **public inheritance**, the most commonly used form. We also will discuss **private inheritance** and **protected inheritance** (Section 12.6). With all forms of inheritance, private members of a base class are not accessible directly from that class's derived classes, but these private base-class members are still inherited (i.e., they are still considered parts of the derived classes). With **public inheritance**, all other base-class members retain their original member access when they become members of the derived class (e.g., public members of the base class become public members of the derived class, and, as we will soon see, protected members of the base class become protected members of the derived class). Through these inherited base-class members, the derived class can manipulate private members of the base class (if these inherited members provide such functionality in the base class). Note that **friend** functions are not inherited.

[Page 638]

Inheritance is not appropriate for every class relationship. In Chapter 10, we discussed the has-a relationship, in which classes have members that are objects of other classes. Such relationships create classes by composition of existing classes. For example, given the classes `Employee`, `BirthDate` and `PhoneNumber`, it is improper to say that an `Employee` is a `BirthDate` or that an `Employee` is a `PhoneNumber`. However, it is appropriate to say that an `Employee` has a `BirthDate` and that an `Employee` has a `PhoneNumber`.

It is possible to treat base-class objects and derived-class objects similarly; their commonalities are expressed in the members of the base class. Objects of all classes derived from a common base class can be treated as objects of that base class (i.e., such objects have an is-a relationship with the base class). In [Chapter 13, Object-Oriented Programming: Polymorphism](#), we consider many examples that take advantage of this relationship.

 PREVNEXT 

page footer

---

The CHM file was converted to HTML by [chm2web](#) software.

# A!K RESEARCH LABS

**SOFTWARE SOLUTIONS**

**Site  
Home**

**SWTT**

**chm2web**

**NotesHolder**

**Pocket  
Icon**

**Forum**

**Contacts**

**Menu:**

**Product Info**

Screenshots

Download

Purchase

Online Help

**Newsletter:**

Please subscribe to our newsletters.

**Your Name:**

**Email:**

**Subscribe to:**

chm2web

NotesHolder

SWTT

Pocket Icon

To be removed from this Newsletter, go to the [unsubscribe page](#).

**Our Partners:**



**Convert chm to html browser-based help system with chm2web**

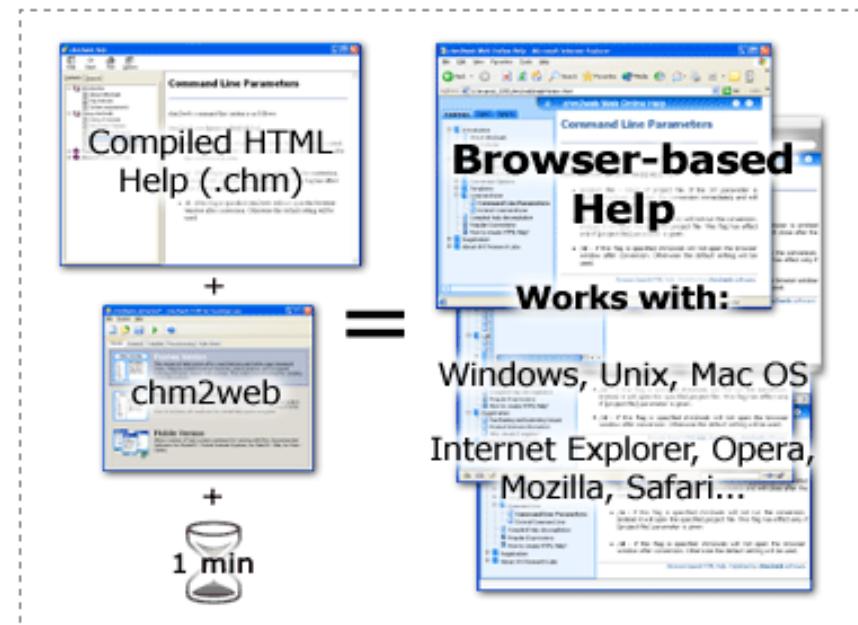
Using chm2web you can convert documentation in the chm format (html help) into a full-featured browser-based platform-independent html help system that can be viewed in any modern web browser. The set of HTML files created by chm2web can be as well uploaded to your web server or used locally as a help system for your products.

Using chm2web you can also convert a compiled HTML Help file of any size to a format readable on your PDA (PocketPC, HPC, Palm or Psion). An example of such converted file is available [here](#).

**Why create web online help systems?**

1. To make your potential customers stay with your product right after the first visit of your web site. Let them see that your product is exactly what they need and that they need not to search further.
2. To drive more traffic to your web site from search engines. There is no other material like your help file that contains as many keywords and key phrases of high relevance to your software.
3. To be able to give direct links to your online help system web pages to your customers instead of trying to explain how to navigate your offline help file.
4. If your software features web-interface it must have an online help system. Even if you do not have a ready HTML help file you can easily create one using freeware Microsoft html help workshop and then convert it to online help system using chm2web within several seconds.

Finally, web based help system increases interest value of your software. For instance, Tucows.com, one of the most popular software archives, gives additional points to the software that offers online help system.





### **chm2web Key Features:**

- There are no limits to customizing the view of your online help system that you can build into the existing sites creating your own templates and using the standard ones.
- Help systems created by chm2web are platform- and browser-independent.
- Chm2web makes use of some solutions enabling searching engines to index the created documentation much better.
- Online web help systems created by chm2web have a tree-like table of contents, a help index, and the full-text search feature.
- It enables the creation of frame-based, frame-free or mobile (for viewing on a Pocket PC or Palm PDA) versions of online help systems;
- The built-in pre-processor is capable of deleting unwanted headers and footers and adding customizable html code to help pages;
- The conversion process can be automated using command line options;
- There is a built-in chm decompiler enabling the extraction of files, including service ones.
- Free to try.
- Affordable price.

Copyright © 2000-2005, AIK Research Labs.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 638 (continued)]

## 12.3. protected Members

Chapter 3 introduced access specifiers `public` and `private`. A base class's `public` members are accessible within the body of that base class and anywhere that the program has a handle (i.e., a name, reference or pointer) to an object of that base class or one of its derived classes. A base class's `private` members are accessible only within the body of that base class and the `friends` of that base class. In this section, we introduce an additional access specifier: `protected`.

Using `protected` access offers an intermediate level of protection between `public` and `private` access. A base class's `protected` members can be accessed within the body of that base class, by members and `friends` of that base class, and by members and `friends` of any classes derived from that base class.

Derived-class member functions can refer to `public` and `protected` members of the base class simply by using the member names. When a derived-class member function redefines a base-class member function, the base-class member can be accessed from the derived class by preceding the base-class member name with the base-class name and the binary scope resolution operator (`::`). We discuss accessing redefined members of the base class in [Section 12.4](#) and using `protected` data in [Section 12.4.4](#).

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 639]

In the first example, we create class `CommissionEmployee`, which contains as `private` data members a first name, last name, social security number, commissionrate (percentage) and gross (i.e., total) sales amount.

The second example defines class `BasePlusCommissionEmployee`, which contains as `private` data members a first name, last name, social security number, commission rate, gross sales amount and base salary. We create the latter class by writing every line of code the class requireswe will soon see that it is much more efficient to create this class simply by inheriting from class `CommissionEmployee`.

The third example defines a new version of class `BasePlusCommissionEmployee` class that inherits directly from class `CommissionEmployee` (i.e., a `BasePlusCommissionEmployee` is a `CommissionEmployee` who also has a base salary) and attempts to access class `CommissionEmployee`'s `private` membersthis results in compilation errors, because the derived class does not have access to the base class's `private` data.

The fourth example shows that if `CommissionEmployee`'s data is declared as `protected`, a new version of class `BasePlusCommissionEmployee` that inherits from class `CommissionEmployee` can access that data directly. For this purpose, we define a new version of class `CommissionEmployee` with `protected` data. Both the inherited and noninherited `BasePlusCommissionEmployee` classes contain identical functionality, but we show how the version of `BasePlusCommissionEmployee` that inherits from class `CommissionEmployee` is easier to create and manage.

After we discuss the convenience of using `protected` data, we create the fifth example, which sets the `CommissionEmployee` data members back to `private` to enforce good software engineering. This example demonstrates that derived class `BasePlusCommissionEmployee` can use base class `CommissionEmployee`'s `public` member functions to manipulate `CommissionEmployee`'s `private` data.

## 12.4.1. Creating and Using a `CommissionEmployee` Class

Let us first examine `CommissionEmployee`'s class definition (Figs. 12.412.5). The `CommissionEmployee` header file (Fig. 12.4) specifies class `CommissionEmployee`'s public services, which include a constructor (lines 1213) and member functions `earnings` (line 30) and `print` (line 31). Lines 1528 declare public get and set functions for manipulating the class's data members (declared in lines 3337) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`. The `CommissionEmployee` header file specifies each of these data members as `private`, so objects of other classes cannot directly access this data. Declaring data members as `private` and providing non-private get and set functions to manipulate and validate the data members helps enforce good software engineering. Member functions `setGrossSales` (defined in lines 5760 of Fig. 12.5) and `setCommissionRate` (defined in lines 6972 of Fig. 12.5), for example, validate their arguments before assigning the values to data members `grossSales` and `commissionRate`, respectively.

**Figure 12.4. `CommissionEmployee` class header file.**

(This item is displayed on page 640 in the print version)

```

1 // Fig. 12.4: CommissionEmployee.h
2 // CommissionEmployee class definition represents a commission employee.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // C++ standard string class
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12 CommissionEmployee(const string &, const string &, const string &,
13 double = 0.0, double = 0.0);
14
15 void setFirstName(const string &); // set first name
16 string getFirstName() const; // return first name
17
18 void setLastName(const string &); // set last name
19 string getLastname() const; // return last name
20
21 void setSocialSecurityNumber(const string &); // set SSN
22 string getSocialSecurityNumber() const; // return SSN
23
24 void setGrossSales(double); // set gross sales amount
25 double getGrossSales() const; // return gross sales amount
26
27 void setCommissionRate(double); // set commission rate (percentage)
28 double getCommissionRate() const; // return commission rate
29
30 double earnings() const; // calculate earnings
31 void print() const; // print CommissionEmployee object

```

```

32 private:
33 string firstName;
34 string lastName;
35 string socialSecurityNumber;
36 double grossSales; // gross weekly sales
37 double commissionRate; // commission percentage
38 };// end class CommissionEmployee
39
40 #endif

```

**Figure 12.5. Implementation file for `CommissionEmployee` class that represents an employee who is paid a percentage of gross sales.**

(This item is displayed on pages 640 - 642 in the print version)

```

1 // Fig. 12.5: CommissionEmployee.cpp
2 // Class CommissionEmployee member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8 // constructor
9 CommissionEmployee::CommissionEmployee(
10 const string &first, const string &last, const string &ssn,
11 double sales, double rate)
12 {
13 firstName = first; // should validate
14 lastName = last; // should validate
15 socialSecurityNumber = ssn; // should validate
16 setGrossSales(sales); // validate and store gross sales
17 setCommissionRate(rate); // validate and store commission rate
18 } // end CommissionEmployee constructor
19
20 // set first name
21 void CommissionEmployee::setFirstName(const string &first)
22 {
23 firstName = first; // should validate
24 } // end function setFirstName
25
26 // return first name
27 string CommissionEmployee::getFirstName() const
28 {
29 return firstName;
30 } // end function getFirstName

```

```
31
32 // set last name
33 void CommissionEmployee::setLastName(const string &last)
34 {
35 lastName = last; // should validate
36 } // end function setLastName
37
38 // return last name
39 string CommissionEmployee::getLastName() const
40 {
41 return lastName;
42 } // end function getLastname
43
44 // set social security number
45 void CommissionEmployee::setSocialSecurityNumber(const string &ssn)
46 {
47 socialSecurityNumber = ssn; // should validate
48 } // end function setSocialSecurityNumber
49
50 // return social security number
51 string CommissionEmployee::getSocialSecurityNumber() const
52 {
53 return socialSecurityNumber;
54 } // end function getSocialSecurityNumber
55
56 // set gross sales amount
57 void CommissionEmployee::setGrossSales(double sales)
58 {
59 grossSales = (sales < 0.0) ? 0.0 : sales;
60 } // end function setGrossSales
61
62 // return gross sales amount
63 double CommissionEmployee::getGrossSales() const
64 {
65 return grossSales;
66 } // end function getGrossSales
67
68 // set commission rate
69 void CommissionEmployee::setCommissionRate(double rate)
70 {
71 commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;
72 } // end function setCommissionRate
73
74 // return commission rate
75 double CommissionEmployee::getCommissionRate() const
76 {
77 return commissionRate;
78 } // end function getCommissionRate
79
```

```

80 // calculate earnings
81 double CommissionEmployee::earnings() const
82 {
83 return commissionRate * grossSales;
84 } // end function earnings
85
86 // print CommissionEmployee object
87 void CommissionEmployee::print() const
88 {
89 cout << "commission employee: " << firstName << ' ' << lastName
90 << "\nsocial security number: " << socialSecurityNumber
91 << "\ngross sales: " << grossSales
92 << "\ncommission rate: " << commissionRate;
93 } // end function print

```

The `CommissionEmployee` constructor definition purposely does not use member-initializer syntax in the first several examples of this section, so that we can demonstrate how `private` and `protected` specifiers affect member access in derived classes. As shown in Fig. 12.5, lines 1315, we assign values to data members `firstName`, `lastName` and `socialSecurityNumber` in the constructor body. Later in this section, we will return to using member-initializer lists in the constructors.

---

[Page 642]

Note that we do not validate the values of the constructor's arguments `first`, `last` and `ssn` before assigning them to the corresponding data members. We certainly could validate the `first` and `last` names perhaps by ensuring that they are of a reasonable length. Similarly, a social security number could be validated to ensure that it contains nine digits, with or without dashes (e.g., 123-45-6789 or 123456789).

Member function `earnings` (lines 8184) calculates a `CommissionEmployee`'s earnings. Line 83 multiplies the `commissionRate` by the `grossSales` and returns the result. Member function `print` (lines 8793) displays the values of a `CommissionEmployee` object's data members.

**Figure 12.6** tests class `CommissionEmployee`. Lines 1617 instantiate object `employee` of class `CommissionEmployee` and invoke `CommissionEmployee`'s constructor to initialize the object with "Sue" as the first name, "Jones" as the last name, "222-22-2222" as the social security number, 10000 as the gross sales amount and .06 as the commission rate. Lines 2329 use `employee`'s get functions to display the values of its data members. Lines 3132 invoke the object's member functions `setGrossSales` and `setCommissionRate` to change the values of data members `grossSales` and `commissionRate`, respectively. Line 36 then calls `employee`'s `print` member function to output the updated `CommissionEmployee` information. Finally, line 39 displays the `CommissionEmployee`'s earnings, calculated by the object's `earnings` member function using the updated values of data members `grossSales` and `commissionRate`.

**Figure 12.6. CommissionEmployee class test program.**

(This item is displayed on pages 643 - 644 in the print version)

```
1 // Fig. 12.6: fig12_06.cpp
2 // Testing class CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include "CommissionEmployee.h" // CommissionEmployee class definition
12
13 int main()
14 {
15 // instantiate a CommissionEmployee object
16 CommissionEmployee employee(
17 "Sue", "Jones", "222-22-2222", 10000, .06);
18
19 // set floating-point output formatting
20 cout << fixed << setprecision(2);
21
22 // get commission employee data
23 cout << "Employee information obtained by get functions: \n"
24 << "\nFirst name is " << employee.getFirstName()
25 << "\nLast name is " << employee.getLastName()
26 << "\nSocial security number is "
27 << employee.getSocialSecurityNumber()
28 << "\nGross sales is " << employee.getGrossSales()
29 << "\nCommission rate is " << employee.getCommissionRate() << endl;
30
31 employee.setGrossSales(8000); // set gross sales
32 employee.setCommissionRate(.1); // set commission rate
33
34 cout << "\nUpdated employee information output by print function: \n"
35 << endl;
36 employee.print(); // display the new employee information
37
38 // display the employee's earnings
39 cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
40
41 return 0;
```

```
42 } // end main
```

Employee information obtained by get functions:

```
First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06
```

Updated employee information output by print function:

```
commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 8000.00
commission rate: 0.10
```

```
Employee's earnings: $800.00
```

[Page 644]

#### 12.4.2. Creating a `BasePlusCommissionEmployee` Class Without Using Inheritance

We now discuss the second part of our introduction to inheritance by creating and testing (a completely new and independent) class `BasePlusCommissionEmployee` ([Figs. 12.7](#)[12.8](#)), which contains a first name, last name, social security number, gross sales amount, commission rate and base salary.

**Figure 12.7. `BasePlusCommissionEmployee` class header file.**

(This item is displayed on pages 644 - 645 in the print version)

```

1 // Fig. 12.7: BasePlusCommissionEmployee.h
2 // BasePlusCommissionEmployee class definition represents an employee
3 // that receives a base salary in addition to commission.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // C++ standard string class
8 using std::string;
9
10 class BasePlusCommissionEmployee
11 {
12 public:
13 BasePlusCommissionEmployee(const string &, const string &,
14 const string &, double = 0.0, double = 0.0, double = 0.0);
15
16 void setFirstName(const string &); // set first name
17 string getFirstName() const; // return first name
18
19 void setLastName(const string &); // set last name
20 string getLastName() const; // return last name
21
22 void setSocialSecurityNumber(const string &); // set SSN
23 string getSocialSecurityNumber() const; // return SSN
24
25 void setGrossSales(double); // set gross sales amount
26 double getGrossSales() const; // return gross sales amount
27
28 void setCommissionRate(double); // set commission rate
29 double getCommissionRate() const; // return commission rate
30
31 void setBaseSalary(double); // set base salary
32 double getBaseSalary() const; // return base salary
33
34 double earnings() const; // calculate earnings
35 void print() const; // print BasePlusCommissionEmployee object
36 private:
37 string firstName;
38 string lastName;
39 string socialSecurityNumber;
40 double grossSales; // gross weekly sales
41 double commissionRate; // commission percentage
42 double baseSalary; // base salary
43 }; // end class BasePlusCommissionEmployee
44
45 #endif

```

**Figure 12.8. BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission.**

(This item is displayed on pages 646 - 648 in the print version)

```

1 // Fig. 12.8: BasePlusCommissionEmployee.cpp
2 // Class BasePlusCommissionEmployee member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 // BasePlusCommissionEmployee class definition
7 #include "BasePlusCommissionEmployee.h"
8
9 // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11 const string &first, const string &last, const string &ssn,
12 double sales, double rate, double salary)
13 {
14 firstName = first; // should validate
15 lastName = last; // should validate
16 socialSecurityNumber = ssn; // should validate
17 setGrossSales(sales); // validate and store gross sales
18 setCommissionRate(rate); // validate and store commission rate
19 setBaseSalary(salary); // validate and store base salary
20 } // end BasePlusCommissionEmployee constructor
21
22 // set first name
23 void BasePlusCommissionEmployee::setFirstName(const string &first)
24 {
25 firstName = first; // should validate
26 } // end function setFirstName
27
28 // return first name
29 string BasePlusCommissionEmployee::getFirstName() const
30 {
31 return firstName;
32 } // end function getFirstName
33
34 // set last name
35 void BasePlusCommissionEmployee::setLastName(const string &last)
36 {
37 lastName = last; // should validate
38 } // end function setLastName
39
40 // return last name
41 string BasePlusCommissionEmployee::getLastName() const
42 {
43 return lastName;

```

```
44 } // end function getLastname
45
46 // set social security number
47 void BasePlusCommissionEmployee::setSocialSecurityNumber(
48 const string &ssn)
49 {
50 socialSecurityNumber = ssn; // should validate
51 } // end function setSocialSecurityNumber
52
53 // return social security number
54 string BasePlusCommissionEmployee::getSocialSecurityNumber() const
55 {
56 return socialSecurityNumber;
57 } // end function getSocialSecurityNumber
58
59 // set gross sales amount
60 void BasePlusCommissionEmployee::setGrossSales(double sales)
61 {
62 grossSales = (sales < 0.0) ? 0.0 : sales;
63 } // end function setGrossSales
64
65 // return gross sales amount
66 double BasePlusCommissionEmployee::getGrossSales() const
67 {
68 return grossSales;
69 } // end function getGrossSales
70
71 // set commission rate
72 void BasePlusCommissionEmployee::setCommissionRate(double rate)
73 {
74 commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;
75 } // end function setCommissionRate
76
77 // return commission rate
78 double BasePlusCommissionEmployee::getCommissionRate() const
79 {
80 return commissionRate;
81 } // end function getCommissionRate
82
83 // set base salary
84 void BasePlusCommissionEmployee::setBaseSalary(double salary)
85 {
86 baseSalary = (salary < 0.0) ? 0.0 : salary;
87 } // end function setBaseSalary
88
89 // return base salary
90 double BasePlusCommissionEmployee::getBaseSalary() const
91 {
92 return baseSalary;
```

```

93 } // end function getBaseSalary
94
95 // calculate earnings
96 double BasePlusCommissionEmployee::earnings() const
97 {
98 return baseSalary + (commissionRate * grossSales);
99 } // end function earnings
100
101 // print BasePlusCommissionEmployee object
102 void BasePlusCommissionEmployee::print() const
103 {
104 cout << "base-salaried commission employee: " << firstName << ' '
105 << lastName << "\nsocial security number: " << socialSecurityNumber
106 << "\ngross sales: " << grossSales
107 << "\ncommission rate: " << commissionRate
108 << "\nbase salary: " << baseSalary;
109 } // end function print

```

## Defining Class `BasePlusCommissionEmployee`

The `BasePlusCommissionEmployee` header file (Fig. 12.7) specifies class `BasePlusCommissionEmployee`'s public services, which include the `BasePlusCommissionEmployee` constructor (lines 1314) and member functions `earnings` (line 34) and `print` (line 35). Lines 1632 declare public get and set functions for the class's private data members (declared in lines 3742) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` and `baseSalary`. These variables and member functions encapsulate all the necessary features of a base-salaried commission employee. Note the similarity between this class and class `CommissionEmployee` (Figs. 12.412.5) in this example, we will not yet exploit that similarity.

---

[Page 645]

Class `BasePlusCommissionEmployee`'s `earnings` member function (defined in lines 9699 of Fig. 12.8) computes the earnings of a base-salaried commission employee. Line 98 returns the result of adding the employee's base salary to the product of the commission rate and the employee's gross sales.

## Testing Class `BasePlusCommissionEmployee`

Figure 12.9 tests class `BasePlusCommissionEmployee`. Lines 1718 instantiate object `employee` of class `BasePlusCommissionEmployee`, passing "Bob", "Lewis", "333-33-3333", 5000, .04 and 300 to the constructor as the first name, last name, social security number, gross sales, commission rate and base salary, respectively. Lines 2431 use `BasePlusCommissionEmployee`'s get functions to retrieve the values of the object's data members for output. Line 33 invokes the object's `setBaseSalary` member function to change the base salary. Member function `setBaseSalary` (Fig. 12.8, lines 8487) ensures that

data member `baseSalary` is not assigned a negative value, because an employee's base salary cannot be negative. Line 37 of Fig. 12.9 invokes the object's `print` member function to output the updated `BasePlusCommissionEmployee`'s information, and line 40 calls member function `earnings` to display the `BasePlusCommissionEmployee`'s earnings.

[Page 646]

[Page 648]

**Figure 12.9. `BasePlusCommissionEmployee` class test program.**

(This item is displayed on pages 648 - 649 in the print version)

```

1 // Fig. 12.9: fig12_09.cpp
2 // Testing class BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // BasePlusCommissionEmployee class definition
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16 // instantiate BasePlusCommissionEmployee object
17 BasePlusCommissionEmployee
18 employee("Bob", "Lewis", "333-33-3333", 5000, .04, 300);
19
20 // set floating-point output formatting
21 cout << fixed << setprecision(2);
22
23 // get commission employee data
24 cout << "Employee information obtained by get functions: \n"
25 << "\nFirst name is " << employee.getFirstName()
26 << "\nLast name is " << employee.getLastName()
27 << "\nSocial security number is "
28 << employee.getSocialSecurityNumber()
29 << "\nGross sales is " << employee.getGrossSales()
30 << "\nCommission rate is " << employee.getCommissionRate()
31 << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33 employee.setBaseSalary(1000); // set base salary

```

```

34
35 cout << "\nUpdated employee information output by print function: \n"
36 << endl;
37 employee.print(); // display the new employee information
38
39 // display the employee's earnings
40 cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42 return 0;
43 } // end main

```

Employee information obtained by get functions:

```

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

```

Updated employee information output by print function:

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

```

Employee's earnings: \$1200.00

[Page 649]

## **Exploring the Similarities Between Class `BasePlusCommissionEmployee` and Class `CommissionEmployee`**

Note that much of the code for class `BasePlusCommissionEmployee` (Figs. 12.712.8) is similar, if not identical, to the code for class `CommissionEmployee` (Figs. 12.412.5). For example, in class `BasePlusCommissionEmployee`, private data members `firstName` and `lastName` and member functions `setFirstName`, `getFirstName`, `setLastName` and `getLastName` are identical to those of class `CommissionEmployee`. Classes `CommissionEmployee` and `BasePlusCommissionEmployee`

also both contain private data members `socialSecurityNumber`, `commissionRate` and `grossSales`, as well as get and set functions to manipulate these members. In addition, the `BasePlusCommissionEmployee` constructor is almost identical to that of class `CommissionEmployee`, except that `BasePlusCommissionEmployee`'s constructor also sets the `baseSalary`. The other additions to class `BasePlusCommissionEmployee` are private data member `baseSalary` and member functions `setBaseSalary` and `getBaseSalary`. Class `BasePlusCommissionEmployee`'s print member function is nearly identical to that of class `CommissionEmployee`, except that `BasePlusCommissionEmployee`'s print also outputs the value of data member `baseSalary`.

We literally copied code from class `CommissionEmployee` and pasted it into class `BasePlusCommissionEmployee`, then modified class `BasePlusCommissionEmployee` to include a base salary and member functions that manipulate the base salary. This "copy-and-paste" approach is often error prone and time consuming. Worse yet, it can spread many physical copies of the same code throughout a system, creating a code-maintenance nightmare. Is there a way to "absorb" the data members and member functions of a class in a way that makes them part of other classes without duplicating code? In the next several examples, we do exactly this, using inheritance.

[Page 650]

### Software Engineering Observation 12.3



Copying and pasting code from one class to another can spread errors across multiple source code files. To avoid duplicating code (and possibly errors), use inheritance, rather than the "copy-and-paste" approach, in situations where you want one class to "absorb" the data members and member functions of another class.

### Software Engineering Observation 12.4



With inheritance, the common data members and member functions of all the classes in the hierarchy are declared in a base class. When changes are required for these common features, software developers need to make the changes only in the base class derived classes then inherit the changes. Without inheritance, changes would need to be made to all the source code files that contain a copy of the code in question.

### 12.4.3. Creating a `CommissionEmployee``BasePlusCommissionEmployee` Inheritance Hierarchy

Now we create and test a new version of class `BasePlusCommissionEmployee` (Figs. 12.1012.11) that derives from class `CommissionEmployee` (Figs. 12.412.5). In this example, a `BasePlusCommissionEmployee` object is a `CommissionEmployee` (because inheritance passes on the capabilities of class `CommissionEmployee`), but class `BasePlusCommissionEmployee` also has data

member `baseSalary` (Fig. 12.10, line 24). The colon (:) in line 12 of the class definition indicates inheritance. Keyword `public` indicates the type of inheritance. As a derived class (formed with `public` inheritance), `BasePlusCommissionEmployee` inherits all the members of class `CommissionEmployee`, except for the constructors each class provides its own constructors that are specific to the class. [Note that destructors, too, are not inherited.] Thus, the `public` services of `BasePlusCommissionEmployee` include its constructor (lines 1516) and the `public` member functions inherited from class `CommissionEmployee` although we cannot see these inherited member functions in `BasePlusCommissionEmployee`'s source code, they are nevertheless a part of derived class `BasePlusCommissionEmployee`. The derived class's `public` services also include member functions `setBaseSalary`, `getBaseSalary`, `earnings` and `print` (lines 1822).

[Page 651]

**Figure 12.10. `BasePlusCommissionEmployee` class definition indicating inheritance relationship with class `CommissionEmployee`.**

(This item is displayed on page 650 in the print version)

```

1 // Fig. 12.10: BasePlusCommissionEmployee.h
2 // BasePlusCommissionEmployee class derived from class
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // C++ standard string class
8 using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15 BasePlusCommissionEmployee(const string &, const string &,
16 const string &, double = 0.0, double = 0.0, double = 0.0);
17
18 void setBaseSalary(double); // set base salary
19 double getBaseSalary() const; // return base salary
20
21 double earnings() const; // calculate earnings
22 void print() const; // print BasePlusCommissionEmployee object
23 private:
24 double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif

```

**Figure 12.11. `BasePlusCommissionEmployee` implementation file: private base-class data cannot be accessed from derived class.**

(This item is displayed on pages 651 - 653 in the print version)

```

1 // Fig. 12.11: BasePlusCommissionEmployee.cpp
2 // Class BasePlusCommissionEmployee member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 // BasePlusCommissionEmployee class definition
7 #include "BasePlusCommissionEmployee.h"
8
9 // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11 const string &first, const string &last, const string &ssn,
12 double sales, double rate, double salary)
13 // explicitly call base-class constructor
14 : CommissionEmployee(first, last, ssn, sales, rate)
15 {
16 setBaseSalary(salary); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary(double salary)
21 {
22 baseSalary = (salary < 0.0) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28 return baseSalary;
29 } // end function getBaseSalary
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const
33 {
34 // derived class cannot access the base class's private data
35 return baseSalary + (commissionRate * grossSales);
36 } // end function earnings
37
38 // print BasePlusCommissionEmployee object
39 void BasePlusCommissionEmployee::print() const
40 {

```

```

41 // derived class cannot access the base class's private data
42 cout << "base-salaried commission employee: " << firstName << ' '
43 << lastName << "\nsocial security number: " << socialSecurityNumber
44 << "\ngross sales: " << grossSales
45 << "\ncommission rate: " << commissionRate
46 << "\nbase salary: " << baseSalary;
47 } // end function print

```

```

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(35) :
error C2248: 'CommissionEmployee::commissionRate' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(37) :
 see declaration of 'CommissionEmployee::commissionRate'
C:\cpphttp5e_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
 see declaration of 'CommissionEmployee'

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp (35) :
error C2248: 'CommissionEmployee::grossSales' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h (36) :
 see declaration of 'CommissionEmployee::grossSales'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
 see declaration of 'CommissionEmployee'

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(42) :
error C2248: 'CommissionEmployee::firstName' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(33) :
 see declaration of 'CommissionEmployee::firstName'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
 see declaration of 'CommissionEmployee'

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(43) :
error C2248: 'CommissionEmployee::lastName' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(34) :
 see declaration of 'CommissionEmployee::lastName'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
 see declaration of 'CommissionEmployee'

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(43) :
error C2248: 'CommissionEmployee::socialSecurity-Number' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(35) :
 see declaration of 'CommissionEmployee::socialSecurityNumber'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
 see declaration of 'CommissionEmployee'

```

```
C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(44) :
error C2248: 'CommissionEmployee::grossSales' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(36) :
 see declaration of 'CommissionEmployee::grossSales'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
 see declaration of 'CommissionEmployee'

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(45) :
error C2248: 'CommissionEmployee::commissionRate' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(37) :
 see declaration of 'CommissionEmployee::commissionRate'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
 see declaration of 'CommissionEmployee'
```

Figure 12.11 shows `BasePlusCommissionEmployee`'s member-function implementations. The constructor (lines 1017) introduces **base-class initializer syntax** (line 14), which uses a member initializer to pass arguments to the base-class (`CommissionEmployee`) constructor. C++ requires a derived-class constructor to call its base-class constructor to initialize the base-class data members that are inherited into the derived class. Line 14 accomplishes this task by invoking the `CommissionEmployee` constructor by name, passing the constructor's parameters `first`, `last`, `ssn`, `sales` and `rate` as arguments to initialize base-class data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`. If `BasePlusCommissionEmployee`'s constructor did not invoke class `CommissionEmployee`'s constructor explicitly, C++ would attempt to invoke class `CommissionEmployee`'s default constructor but the class does not have such a constructor, so the compiler would issue an error. Recall from Chapter 3 that the compiler provides a default constructor with no parameters in any class that does not explicitly include a constructor. However, `CommissionEmployee` does explicitly include a constructor, so a default constructor is not provided and any attempts to implicitly call `CommissionEmployee`'s default constructor would result in compilation errors.

---

[Page 653]

## Common Programming Error 12.1



A compilation error occurs if a derived-class constructor calls one of its base-class constructors with arguments that are inconsistent with the number and types of parameters specified in one of the base-class constructor definitions.

## Performance Tip 12.1



In a derived-class constructor, initializing member objects and invoking base-class constructors explicitly in the member initializer list prevents duplicate initialization in which a default constructor is called, then data members are modified again in the derived-class constructor's body.

The compiler generates errors for line 35 of Fig. 12.11 because base class `CommissionEmployee`'s data members `commissionRate` and `grossSales` are private. The derived class `BasePlusCommissionEmployee`'s member functions are not allowed to access base class `CommissionEmployee`'s private data. Note that we used red text in Fig. 12.11 to indicate erroneous code. The compiler issues additional errors at lines 4245 of `BasePlusCommissionEmployee`'s print member function for the same reason. As you can see, C++ rigidly enforces restrictions on accessing private data members, so that even a derived class (which is intimately related to its base class) cannot access the base class's private data. [Note: To save space, we show only the error messages from Visual C++ .NET in this example. The error messages produced by your compiler may differ from those shown here. Also notice that we highlight key portions of the lengthy error messages in bold.]

[Page 654]

We purposely included the erroneous code in Fig. 12.11 to demonstrate that a derived class's member functions cannot access its base class's private data. The errors in `BasePlusCommissionEmployee` could have been prevented by using the get member functions inherited from class `CommissionEmployee`. For example, line 35 could have invoked `getCommissionRate` and `getGrossSales` to access `CommissionEmployee`'s private data members `commissionRate` and `grossSales`, respectively. Similarly, lines 4245 could have used appropriate get member functions to retrieve the values of the base class's data members. In the next example, we show how using `protected` data also allows us to avoid the errors encountered in this example.

### Including the Base Class Header File in the Derived Class Header File with #include

Notice that we `#include` the base class's header file in the derived class's header file (line 10 of Fig. 12.10). This is necessary for three reasons. First, for the derived class to use the base class's name in line 12, we must tell the compiler that the base class exists—the class definition in `CommissionEmployee.h` does exactly that.

The second reason is that the compiler uses a class definition to determine the size of an object of that class (as we discussed in Section 3.8). A client program that creates an object of a class must `#include` the class definition to enable the compiler to reserve the proper amount of memory for the object. When using inheritance, a derived-class object's size depends on the data members declared explicitly in its class definition and the data members inherited from its direct and indirect base classes. Including the base class's definition in line 10 allows the compiler to determine the memory requirements for the base class's data members that become part of a derived-class object and thus contribute to the total size of the derived-class object.

The last reason for line 10 is to allow the compiler to determine whether the derived class uses the base class's inherited members properly. For example, in the program of Figs. 12.1012.11, the compiler uses the base-class header file to determine that the data members being accessed by the derived class are private in the base class. Since these are inaccessible to the derived class, the compiler generates errors. The compiler also uses the base class's function prototypes to validate function calls made by the derived class to the inherited base-class functions you will see an example of such a function call in Fig. 12.16.

## Linking Process in an Inheritance Hierarchy

In Section 3.9, we discussed the linking process for creating an executable GradeBook application. In that example, you saw that the client's object code was linked with the object code for class GradeBook, as well as the object code for any C++ Standard Library classes used in either the client code or in class GradeBook.

The linking process is similar for a program that uses classes in an inheritance hierarchy. The process requires the object code for all classes used in the program and the object code for the direct and indirect base classes of any derived classes used by the program. Suppose a client wants to create an application that uses class BasePlusCommissionEmployee, which is a derived class of CommissionEmployee (we will see an example of this in Section 12.4.4). When compiling the client application, the client's object code must be linked with the object code for classes BasePlusCommissionEmployee and CommissionEmployee, because BasePlusCommissionEmployee inherits member functions from its base class CommissionEmployee. The code is also linked with the object code for any C++ Standard Library classes used in class CommissionEmployee, class BasePlusCommissionEmployee or the client code. This provides the program with access to the implementations of all of the functionality that the program may use.

[Page 655]

### 12.4.4. CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy Using protected Data

To enable class BasePlusCommissionEmployee to directly access CommissionEmployee data members firstName, lastName, socialSecurityNumber, grossSales and commissionRate, we can declare those members as `protected` in the base class. As we discussed in Section 12.3, a base class's `protected` members can be accessed by members and `friends` of the base class and by members and `friends` of any classes derived from that base class.

#### Good Programming Practice 12.1



Declare public members first, `protected` members second and `private` members last.

## Defining Base Class `CommissionEmployee` with `protected` Data

Class `CommissionEmployee` (Figs. 12.1212.13) now declares data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` as `protected` (Fig. 12.12, lines 3337) rather than `private`. The member-function implementations in Fig. 12.13 are identical to those in Fig. 12.5.

**Figure 12.12. `CommissionEmployee` class definition that declares `protected` data to allow access by derived classes.**

(This item is displayed on pages 655 - 656 in the print version)

```

1 // Fig. 12.12: CommissionEmployee.h
2 // CommissionEmployee class definition with protected data.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // C++ standard string class
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12 CommissionEmployee(const string &, const string &, const string &,
13 double = 0.0, double = 0.0);
14
15 void setFirstName(const string &); // set first name
16 string getFirstName() const; // return first name
17
18 void setLastName(const string &); // set last name
19 string getLastname() const; // return last name
20
21 void setSocialSecurityNumber(const string &); // set SSN
22 string getSocialSecurityNumber() const; // return SSN
23
24 void setGrossSales(double); // set gross sales amount
25 double getGrossSales() const; // return gross sales amount
26
27 void setCommissionRate(double); // set commission rate
28 double getCommissionRate() const; // return commission rate
29
30 double earnings() const; // calculate earnings
31 void print() const; // print CommissionEmployee object
32 protected:
33 string firstName;
```

```

34 string lastName;
35 string socialSecurityNumber;
36 double grossSales; // gross weekly sales
37 double commissionRate; // commission percentage
38 } // end class CommissionEmployee
39
40 #endif

```

[Page 656]

**Figure 12.13. CommissionEmployee class with protected data.**

(This item is displayed on pages 656 - 658 in the print version)

```

1 // Fig. 12.13: CommissionEmployee.cpp
2 // Class CommissionEmployee member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8 // constructor
9 CommissionEmployee::CommissionEmployee(
10 const string &first, const string &last, const string &ssn,
11 double sales, double rate)
12 {
13 firstName = first; // should validate
14 lastName = last; // should validate
15 socialSecurityNumber = ssn; // should validate
16 setGrossSales(sales); // validate and store gross sales
17 setCommissionRate(rate); // validate and store commission rate
18 } // end CommissionEmployee constructor
19
20 // set first name
21 void CommissionEmployee::setFirstName(const string &first)
22 {
23 firstName = first; // should validate
24 } // end function setFirstName
25
26 // return first name
27 string CommissionEmployee::getFirstName() const
28 {
29 return firstName;
30 } // end function getFirstName

```

```
31
32 // set last name
33 void CommissionEmployee::setLastName(const string &last)
34 {
35 lastName = last; // should validate
36 } // end function setLastName
37
38 // return last name
39 string CommissionEmployee::getLastName() const
40 {
41 return lastName;
42 } // end function getLastname
43
44 // set social security number
45 void CommissionEmployee::setSocialSecurityNumber(const string &ssn)
46 {
47 socialSecurityNumber = ssn; // should validate
48 } // end function setSocialSecurityNumber
49
50 // return social security number
51 string CommissionEmployee::getSocialSecurityNumber() const
52 {
53 return socialSecurityNumber;
54 } // end function getSocialSecurityNumber
55
56 // set gross sales amount
57 void CommissionEmployee::setGrossSales(double sales)
58 {
59 grossSales = (sales < 0.0) ? 0.0 : sales;
60 } // end function setGrossSales
61
62 // return gross sales amount
63 double CommissionEmployee::getGrossSales() const
64 {
65 return grossSales;
66 } // end function getGrossSales
67
68 // set commission rate
69 void CommissionEmployee::setCommissionRate(double rate)
70 {
71 commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;
72 } // end function setCommissionRate
73
74 // return commission rate
75 double CommissionEmployee::getCommissionRate() const
76 {
77 return commissionRate;
78 } // end function getCommissionRate
79
```

```

80 // calculate earnings
81 double CommissionEmployee::earnings() const
82 {
83 return commissionRate * grossSales;
84 } // end function earnings
85
86 // print CommissionEmployee object
87 void CommissionEmployee::print() const
88 {
89 cout << "commission employee: " << firstName << ' ' << lastName
90 << "\nsocial security number: " << socialSecurityNumber
91 << "\ngross sales: " << grossSales
92 << "\ncommission rate: " << commissionRate;
93 } // end function print

```

---

[Page 658]

## Modifying Derived Class `BasePlusCommissionEmployee`

We now modify class `BasePlusCommissionEmployee` ([Figs. 12.14](#)[12.15](#)) so that it inherits from the version of class `CommissionEmployee` in [Figs. 12.12](#)[12.13](#). Because class `BasePlusCommissionEmployee` inherits from this version of class `CommissionEmployee`, objects of class `BasePlusCommissionEmployee` can access inherited data members that are declared `protected` in class `CommissionEmployee` (i.e., data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate`). As a result, the compiler does not generate errors when compiling the `BasePlusCommissionEmployee` `earnings` and `print` member-function definitions in [Fig. 12.15](#) (lines 3236 and 3947, respectively). This shows the special privileges that a derived class is granted to access protected baseclass data members. Objects of a derived class also can access `protected` members in any of that derived class's indirect base classes.

---

[Page 659]

### Figure 12.14. `BasePlusCommissionEmployee` class header file.

(This item is displayed on page 658 in the print version)

```

1 // Fig. 12.14: BasePlusCommissionEmployee.h
2 // BasePlusCommissionEmployee class derived from class
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // C++ standard string class
8 using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15 BasePlusCommissionEmployee(const string &, const string &,
16 const string &, double = 0.0, double = 0.0, double = 0.0);
17
18 void setBaseSalary(double); // set base salary
19 double getBaseSalary() const; // return base salary
20
21 double earnings() const; // calculate earnings
22 void print() const; // print BasePlusCommissionEmployee object
23 private:
24 double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif

```

**Figure 12.15. BasePlusCommissionEmployee implementation file for BasePlusCommissionEmployee class that inherits protected data from CommissionEmployee.**

```

1 // Fig. 12.15: BasePlusCommissionEmployee.cpp
2 // Class BasePlusCommissionEmployee member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 // BasePlusCommissionEmployee class definition
7 #include "BasePlusCommissionEmployee.h"
8
9 // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11 const string &first, const string &last, const string &ssn,
12 double sales, double rate, double salary)
13 // explicitly call base-class constructor
14 : CommissionEmployee(first, last, ssn, sales, rate)

```

```

15 {
16 setBaseSalary(salary); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary(double salary)
21 {
22 baseSalary = (salary < 0.0) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28 return baseSalary;
29 } // end function getBaseSalary
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const
33 {
34 // can access protected data of base class
35 return baseSalary + (commissionRate * grossSales);
36 } // end function earnings
37
38 // print BasePlusCommissionEmployee object
39 void BasePlusCommissionEmployee::print() const
40 {
41 // can access protected data of base class
42 cout << "base-salaried commission employee: " << firstName << ' '
43 << lastName << "\nsocial security number: " << socialSecurityNumber
44 << "\ngross sales: " << grossSales
45 << "\ncommission rate: " << commissionRate
46 << "\nbase salary: " << baseSalary;
47 } // end function print

```

[Page 660]

Class `BasePlusCommissionEmployee` does not inherit class `CommissionEmployee`'s constructor. However, class `BasePlusCommissionEmployee`'s constructor (Fig. 12.15, lines 1017) calls class `CommissionEmployee`'s constructor explicitly (line 14). Recall that `BasePlusCommissionEmployee`'s constructor must explicitly call the constructor of class `CommissionEmployee`, because `CommissionEmployee` does not contain a default constructor that could be invoked implicitly.

## Testing the Modified `BasePlusCommissionEmployee` Class

Figure 12.16 uses a `BasePlusCommissionEmployee` object to perform the same tasks that Fig. 12.9 performed on an object of the first version of class `BasePlusCommissionEmployee` (Figs. 12.712.8). Note that the outputs of the two programs are identical. We created the first class `BasePlusCommissionEmployee` without using inheritance and created this version of `BasePlusCommissionEmployee` using inheritance; however, both classes provide the same functionality. Note that the code for class `BasePlusCommissionEmployee` (i.e., the header and implementation files), which is 74 lines, is considerably shorter than the code for the noninherited version of the class, which is 154 lines, because the inherited version absorbs part of its functionality from `CommissionEmployee`, whereas the noninherited version does not absorb any functionality. Also, there is now only one copy of the `CommissionEmployee` functionality declared and defined in class `CommissionEmployee`. This makes the source code easier to maintain, modify and debug, because the source code related to a `CommissionEmployee` exists only in the files of Figs. 12.1212.13.

[Page 661]

### Figure 12.16. protected base-class data can be accessed from derived class.

(This item is displayed on pages 660 - 661 in the print version)

```

1 // Fig. 12.16: fig12_16.cpp
2 // Testing class BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // BasePlusCommissionEmployee class definition
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16 // instantiate BasePlusCommissionEmployee object
17 BasePlusCommissionEmployee
18 employee("Bob", "Lewis", "333-33-3333", 5000, .04, 300);
19
20 // set floating-point output formatting
21 cout << fixed << setprecision(2);
22
23 // get commission employee data
24 cout << "Employee information obtained by get functions: \n"
25 << "\nFirst name is " << employee.getFirstName()
26 << "\nLast name is " << employee.getLastName()
27 << "\nSocial security number is "

```

```

28 << employee.getSocialSecurityNumber()
29 << "\nGross sales is " << employee.getGrossSales()
30 << "\nCommission rate is " << employee.getCommissionRate()
31 << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33 employee.setBaseSalary(1000); // set base salary
34
35 cout << "\nUpdated employee information output by print function: \n"
36 << endl;
37 employee.print(); // display the new employee information
38
39 // display the employee's earnings
40 cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42 return 0;
43 } // end main

```

Employee information obtained by get functions:

First name is Bob  
 Last name is Lewis  
 Social security number is 333-33-3333  
 Gross sales is 5000.00  
 Commission rate is 0.04  
 Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis  
 social security number: 333-33-3333  
 gross sales: 5000.00  
 commission rate: 0.04  
 base salary: 1000.00

Employee's earnings: \$1200.00

## Notes on Using **protected** Data

In this example, we declared base-class data members as **protected**, so that derived classes could modify the data directly. Inheriting **protected** data members slightly increases performance, because we can directly access the members without incurring the overhead of calls to set or get member functions. In most cases, however, it is better to use **private** data members to encourage proper software engineering.

and leave code optimization issues to the compiler. Your code will be easier to maintain, modify and debug.

Using `protected` data members creates two major problems. First, the derived-class object does not have to use a member function to set the value of the base-class's `protected` data member. Therefore, a derived-class object easily can assign an invalid value to the `protected` data member, thus leaving the object in an inconsistent state. For example, with `CommissionEmployee`'s data member `grossSales` declared as `protected`, a derived-class (e.g., `BasePlusCommissionEmployee`) object can assign a negative value to `grossSales`. The second problem with using `protected` data members is that derived-class member functions are more likely to be written so that they depend on the base-class implementation. In practice, derived classes should depend only on the base-class services (i.e., non `private` member functions) and not on the base-class implementation. With `protected` data members in the base class, if the base-class implementation changes, we may need to modify all derived classes of that base class. For example, if for some reason we were to change the names of data members `firstName` and `lastName` to `first` and `last`, then we would have to do so for all occurrences in which a derived class references these base-class data members directly. In such a case, the software is said to be **fragile** or **brittle**, because a small change in the base class can "break" derived-class implementation. The programmer should be able to change the base-class implementation while still providing the same services to derived classes. (Of course, if the base-class services change, we must reimplement our derived classes—good object-oriented design attempts to prevent this.)

[Page 662]

### Software Engineering Observation 12.5



It is appropriate to use the `protected` access specifier when a base class should provide a service (i.e., a member function) only to its derived classes (and friends), not to other clients.

### Software Engineering Observation 12.6



Declaring base-class data members `private` (as opposed to declaring them `protected`) enables programmers to change the base-class implementation without having to change derived-class implementations.

### Error-Prevention Tip 12.1



When possible, avoid including `protected` data members in a base class. Rather, include non-`private` member functions that access `private` data members, ensuring that the object maintains a consistent state.

## 12.4.5. CommissionEmployee-BasePlusCommissionEmployee Inheritance Hierarchy Using private Data

We now reexamine our hierarchy once more, this time using the best software engineering practices. Class `CommissionEmployee` (Figs. 12.1712.18) now declares data members `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` and `commissionRate` as `private` (Fig. 12.17, lines 3337) and provides `public` member functions `setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber`, `getSocialSecurityNumber`, `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` and `print` for manipulating these values. If we decide to change the data member names, the `earnings` and `print` definitions will not require modificationonly the definitions of the `get` and `set` member functions that directly manipulate the data members will need to change. Note that these changes occur solely within the base classno changes to the derived class are needed. Localizing the effects of changes like this is a good software engineering practice. Derived class `BasePlusCommissionEmployee` (Figs. 12.1912.20) inherits `CommissionEmployee`'s non-`private` member functions and can access the `private` base-class members via those member functions.

---

[Page 663]

**Figure 12.17. CommissionEmployee class defined using good software engineering practices.**

```

1 // Fig. 12.17: CommissionEmployee.h
2 // CommissionEmployee class definition with good software engineering.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // C++ standard string class
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12 CommissionEmployee(const string &, const string &, const string &,
13 double = 0.0, double = 0.0);
14
15 void setFirstName(const string &); // set first name
16 string getFirstName() const; // return first name
17
18 void setLastName(const string &); // set last name
19 string getLastname() const; // return last name
20
21 void setSocialSecurityNumber(const string &); // set SSN
22 string getSocialSecurityNumber() const; // return SSN
23
24 void setGrossSales(double); // set gross sales amount

```

```

25 double getGrossSales() const; // return gross sales amount
26
27 void setCommissionRate(double); // set commission rate
28 double getCommissionRate() const; // return commission rate
29
30 double earnings() const; // calculate earnings
31 void print() const; // print CommissionEmployee object
32 private:
33 string firstName;
34 string lastName;
35 string socialSecurityNumber;
36 double grossSales; // gross weekly sales
37 double commissionRate; // commission percentage
38 } // end class CommissionEmployee
39
40 #endif

```

**Figure 12.18. CommissionEmployee class implementation file: CommissionEmployee class uses member functions to manipulate its private data.**

(This item is displayed on pages 663 - 665 in the print version)

```

1 // Fig. 12.18: CommissionEmployee.cpp
2 // Class CommissionEmployee member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8 // constructor
9 CommissionEmployee::CommissionEmployee(
10 const string &first, const string &last, const string &ssn,
11 double sales, double rate)
12 : firstName(first), lastName(last), socialSecurityNumber(ssn)
13 {
14 setGrossSales(sales); // validate and store gross sales
15 setCommissionRate(rate); // validate and store commission rate
16 } // end CommissionEmployee constructor
17
18 // set first name
19 void CommissionEmployee::setFirstName(const string &first)
20 {
21 firstName = first; // should validate
22 } // end function setFirstName
23

```

```
24 // return first name
25 string CommissionEmployee::getFirstName() const
26 {
27 return firstName;
28 } // end function getFirstName
29
30 // set last name
31 void CommissionEmployee::setLastName(const string &last)
32 {
33 lastName = last; // should validate
34 } // end function setLastName
35
36 // return last name
37 string CommissionEmployee::getLastName() const
38 {
39 return lastName;
40 } // end function getLastname
41
42 // set social security number
43 void CommissionEmployee::setSocialSecurityNumber(const string &ssn)
44 {
45 socialSecurityNumber = ssn; // should validate
46 } // end function setSocialSecurityNumber
47
48 // return social security number
49 string CommissionEmployee::getSocialSecurityNumber() const
50 {
51 return socialSecurityNumber;
52 } // end function getSocialSecurityNumber
53
54 // set gross sales amount
55 void CommissionEmployee::setGrossSales(double sales)
56 {
57 grossSales = (sales < 0.0) ? 0.0 : sales;
58 } // end function setGrossSales
59
60 // return gross sales amount
61 double CommissionEmployee::getGrossSales() const
62 {
63 return grossSales;
64 } // end function getGrossSales
65
66 // set commission rate
67 void CommissionEmployee::setCommissionRate(double rate)
68 {
69 commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;
70 } // end function setCommissionRate
71
72 // return commission rate
```

```

73 double CommissionEmployee::getCommissionRate() const
74 {
75 return commissionRate;
76 } // end function getCommissionRate
77
78 // calculate earnings
79 double CommissionEmployee::earnings() const
80 {
81 return getCommissionRate() * getGrossSales();
82 } // end function earnings
83
84 // print CommissionEmployee object
85 void CommissionEmployee::print() const
86 {
87 cout << "commission employee: "
88 << getFirstName() << ' ' << getLastName()
89 << "\nsocial security number: " << getSocialSecurityNumber()
90 << "\ngross sales: " << getGrossSales()
91 << "\ncommission rate: " << getCommissionRate();
92 } // end function print

```

**Figure 12.19. BasePlusCommissionEmployee class header file.**

(This item is displayed on page 666 in the print version)

```

1 // Fig. 12.19: BasePlusCommissionEmployee.h
2 // BasePlusCommissionEmployee class derived from class
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // C++ standard string class
8 using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15 BasePlusCommissionEmployee(const string &, const string &,
16 const string &, double = 0.0, double = 0.0, double = 0.0);
17
18 void setBaseSalary(double); // set base salary
19 double getBaseSalary() const; // return base salary
20

```

```

21 double earnings() const; // calculate earnings
22 void print() const; // print BasePlusCommissionEmployee object
23 private:
24 double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif

```

**Figure 12.20. BasePlusCommissionEmployee class that inherits from class CommissionEmployee but cannot directly access the class's private data.**

(This item is displayed on pages 666 - 667 in the print version)

```

1 // Fig. 12.20: BasePlusCommissionEmployee.cpp
2 // Class BasePlusCommissionEmployee member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 // BasePlusCommissionEmployee class definition
7 #include "BasePlusCommissionEmployee.h"
8
9 // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11 const string &first, const string &last, const string &ssn,
12 double sales, double rate, double salary)
13 // explicitly call base-class constructor
14 : CommissionEmployee(first, last, ssn, sales, rate)
15 {
16 setBaseSalary(salary); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary(double salary)
21 {
22 baseSalary = (salary < 0.0) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28 return baseSalary;
29 } // end function getBaseSalary
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const

```

```

33 {
34 return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee object
38 void BasePlusCommissionEmployee::print() const
39 {
40 cout << "base-salaried ";
41
42 // invoke CommissionEmployee's print function
43 CommissionEmployee::print();
44
45 cout << "\nbase salary: " << getBaseSalary();
46 } // end function print

```

[Page 665]

In the CommissionEmployee constructor implementation (Fig. 12.18, lines 916), note that we use member initializers (line 12) to set the values of members `firstName`, `lastName` and `socialSecurityNumber`. We show how derived-class `BasePlusCommissionEmployee` (Figs. 12.1912.20) can invoke non-private base-class member functions (`setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber` and `getSocialSecurityNumber`) to manipulate these data members.

#### Performance Tip 12.2



Using a member function to access a data member's value can be slightly slower than accessing the data directly. However, today's optimizing compilers are carefully designed to perform many optimizations implicitly (such as inlining set and get member-function calls). As a result, programmers should write code that adheres to proper software engineering principles, and leave optimization issues to the compiler. A good rule is, "Do not second-guess the compiler."

[Page 666]

Class `BasePlusCommissionEmployee` (Figs. 12.1912.20) has several changes to its member-function implementations (Fig. 12.20) that distinguish it from the previous version of the class (Figs. 12.1412.15). Member functions `earnings` (Fig. 12.20, lines 3235) and `print` (lines 3846) each invoke member function `getBaseSalary` to obtain the base salary value, rather than accessing `baseSalary` directly. This insulates `earnings` and `print` from potential changes to the implementation of data member `baseSalary`. For example, if we decide to rename data member `baseSalary` or change its type, only

member functions `setBaseSalary` and `getBaseSalary` will need to change.

[Page 667]

Class `BasePlusCommissionEmployee`'s `earnings` function (Fig. 12.20, lines 3235) redefines class `CommissionEmployee`'s `earnings` member function (Fig. 12.18, lines 7982) to calculate the earnings of a base-salaried commission employee. Class `BasePlusCommissionEmployee`'s version of `earnings` obtains the portion of the employee's earnings based on commission alone by calling base-class `CommissionEmployee`'s `earnings` function with the expression `CommissionEmployee::earnings()` (Fig. 12.20, line 34). `BasePlusCommissionEmployee`'s `earnings` function then adds the base salary to this value to calculate the total earnings of the employee. Note the syntax used to invoke a redefined base-class member function from a derived classplace the base-class name and the binary scope resolution operator (`::`) before the base-class member-function name. This member-function invocation is a good software engineering practice: Recall from Software Engineering Observation 9.9 that, if an object's member function performs the actions needed by another object, we should call that member function rather than duplicating its code body. By having `BasePlusCommissionEmployee`'s `earnings` function invoke `CommissionEmployee`'s `earnings` function to calculate part of a `BasePlusCommissionEmployee` object's `earnings`, we avoid duplicating the code and reduce code-maintenance problems.

[Page 668]

## Common Programming Error 12.2



When a base-class member function is redefined in a derived class, the derived-class version often calls the base-class version to do additional work. Failure to use the `::` operator prefixed with the name of the base class when referencing the base class's member function causes infinite recursion, because the derived-class member function would then call itself.

## Common Programming Error 12.3



Including a base-class member function with a different signature in the derived class hides the base-class version of the function. Attempts to call the base-class version through the `public` interface of a derived-class object result in compilation errors.

Similarly, `BasePlusCommissionEmployee`'s `print` function (Fig. 12.20, lines 3846) redefines class `CommissionEmployee`'s `print` member function (Fig. 12.18, lines 8592) to output information that is appropriate for a base-salaried commission employee. Class `BasePlusCommissionEmployee`'s version displays part of a `BasePlusCommissionEmployee` object's information (i.e., the string "commission employee" and the values of class `CommissionEmployee`'s private data members) by calling

CommissionEmployee's print member function with the qualified name CommissionEmployee::print() (Fig. 12.20, line 43). BasePlusCommissionEmployee's print function then outputs the remainder of a BasePlusCommissionEmployee object's information (i.e., the value of class BasePlusCommissionEmployee's base salary).

Figure 12.21 performs the same manipulations on a BasePlusCommissionEmployee object as did Fig. 12.9 and Fig. 12.16 on objects of classes CommissionEmployee and BasePlusCommissionEmployee, respectively. Although each "base-salaried commission employee" class behaves identically, class BasePlusCommissionEmployee is the best engineered. By using inheritance and by calling member functions that hide the data and ensure consistency, we have efficiently and effectively constructed a well-engineered class.

**Figure 12.21. Base-class private data is accessible to a derived class via public or protected member function inherited by the derived class.**

(This item is displayed on pages 668 - 669 in the print version)

```

1 // Fig. 12.21: fig12_21.cpp
2 // Testing class BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // BasePlusCommissionEmployee class definition
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16 // instantiate BasePlusCommissionEmployee object
17 BasePlusCommissionEmployee
18 employee("Bob", "Lewis", "333-33-3333", 5000, .04, 300);
19
20 // set floating-point output formatting
21 cout << fixed << setprecision(2);
22
23 // get commission employee data
24 cout << "Employee information obtained by get functions: \n"
25 << "\nFirst name is " << employee.getFirstName()
26 << "\nLast name is " << employee.getLastName()
27 << "\nSocial security number is "
28 << employee.getSocialSecurityNumber()
29 << "\nGross sales is " << employee.getGrossSales()
30 << "\nCommission rate is " << employee.getCommissionRate()
```

```

31 << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33 employee.setBaseSalary(1000); // set base salary
34
35 cout << "\nUpdated employee information output by print function: \n"
36 << endl;
37 employee.print(); // display the new employee information
38
39 // display the employee's earnings
40 cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42 return 0;
43 } // end main

```

Employee information obtained by get functions:

First name is Bob  
 Last name is Lewis  
 Social security number is 333-33-3333  
 Gross sales is 5000.00  
 Commission rate is 0.04  
 Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis  
 social security number: 333-33-3333  
 gross sales: 5000.00  
 commission rate: 0.04  
 base salary: 1000.00

Employee's earnings: \$1200.00

[Page 670]

In this section, you saw an evolutionary set of examples that was carefully designed to teach key capabilities for good software engineering with inheritance. You learned how to create a derived class using inheritance, how to use protected base-class members to enable a derived class to access inherited base-class data members and how to redefine baseclass functions to provide versions that are more appropriate for derived-class objects. In addition, you learned how to apply software engineering techniques from Chapters 910 and this chapter to create classes that are easy to maintain, modify and debug.

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 671]

Base-class constructors, destructors and overloaded assignment operators (see [Chapter 11](#), Operator Overloading; String and Array Objects) are not inherited by derived classes. Derived-class constructors, destructors and overloaded assignment operators, however, can call base-class constructors, destructors and overloaded assignment operators.

Our next example revisits the commission employee hierarchy by defining class `CommissionEmployee` ([Figs. 12.22](#)[12.23](#)) and class `BasePlusCommissionEmployee` ([Figs. 12.24](#)[12.25](#)) that contain constructors and destructors, each of which prints a message when it is invoked. As you will see in the output in [Fig. 12.26](#), these messages demonstrate the order in which the constructors and destructors are called for objects in an inheritance hierarchy.

**Figure 12.22. CommissionEmployee class header file.**

```

1 // Fig. 12.22: CommissionEmployee.h
2 // CommissionEmployee class definition represents a commission employee.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // C++ standard string class
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12 CommissionEmployee(const string &, const string &, const string &,
13 double = 0.0, double = 0.0);
14 ~CommissionEmployee(); // destructor
15
16 void setFirstName(const string &); // set first name
17 string getFirstName() const; // return first name
18
19 void setLastName(const string &); // set last name
20 string getLastName() const; // return last name
21
22 void setSocialSecurityNumber(const string &); // set SSN
23 string getSocialSecurityNumber() const; // return SSN
24
25 void setGrossSales(double); // set gross sales amount
26 double getGrossSales() const; // return gross sales amount
27

```

```

28 void setCommissionRate(double); // set commission rate
29 double getCommissionRate() const; // return commission rate
30
31 double earnings() const; // calculate earnings
32 void print() const; // print CommissionEmployee object
33 private:
34 string firstName;
35 string lastName;
36 string socialSecurityNumber;
37 double grossSales; // gross weekly sales
38 double commissionRate; // commission percentage
39 }; // end class CommissionEmployee
40
41 #endif

```

**Figure 12.23. CommissionEmployee's constructor outputs text.**

(This item is displayed on pages 672 - 674 in the print version)

```

1 // Fig. 12.23: CommissionEmployee.cpp
2 // Class CommissionEmployee member-function definitions.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "CommissionEmployee.h" // CommissionEmployee class definition
8
9 // constructor
10 CommissionEmployee::CommissionEmployee(
11 const string &first, const string &last, const string &ssn,
12 double sales, double rate)
13 : firstName(first), lastName(last), socialSecurityNumber(ssn)
14 {
15 setGrossSales(sales); // validate and store gross sales
16 setCommissionRate(rate); // validate and store commission rate
17
18 cout << "CommissionEmployee constructor: " << endl;
19 print();
20 cout << "\n\n";
21 } // end CommissionEmployee constructor
22
23 // destructor
24 CommissionEmployee::~CommissionEmployee()
25 {

```

```
26 cout << "CommissionEmployee destructor: " << endl;
27 print();
28 cout << "\n\n";
29 } // end CommissionEmployee destructor
30
31 // set first name
32 void CommissionEmployee::setFirstName(const string &first)
33 {
34 firstName = first; // should validate
35 } // end function setFirstName
36
37 // return first name
38 string CommissionEmployee::getFirstName() const
39 {
40 return firstName;
41 } // end function getFirstName
42
43 // set last name
44 void CommissionEmployee::setLastName(const string &last)
45 {
46 lastName = last; // should validate
47 } // end function setLastName
48
49 // return last name
50 string CommissionEmployee::getLastName() const
51 {
52 return lastName;
53 } // end function getLastName
54
55 // set social security number
56 void CommissionEmployee::setSocialSecurityNumber(const string &ssn)
57 {
58 socialSecurityNumber = ssn; // should validate
59 } // end function setSocialSecurityNumber
60
61 // return social security number
62 string CommissionEmployee::getSocialSecurityNumber() const
63 {
64 return socialSecurityNumber;
65 } // end function getSocialSecurityNumber
66
67 // set gross sales amount
68 void CommissionEmployee::setGrossSales(double sales)
69 {
70 grossSales = (sales < 0.0) ? 0.0 : sales;
71 } // end function setGrossSales
72
73 // return gross sales amount
```

```
74 double CommissionEmployee::getGrossSales() const
75 {
76 return grossSales;
77 } // end function getGrossSales
78
79 // set commission rate
80 void CommissionEmployee::setCommissionRate(double rate)
81 {
82 commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;
83 } // end function setCommissionRate
84
85 // return commission rate
86 double CommissionEmployee::getCommissionRate() const
87 {
88 return commissionRate;
89 } // end function getCommissionRate
90
91 // calculate earnings
92 double CommissionEmployee::earnings() const
93 {
94 return getCommissionRate() * getGrossSales();
95 } // end function earnings
96
97 // print CommissionEmployee object
98 void CommissionEmployee::print() const
99 {
100 cout << "commission employee: "
101 << getFirstName() << ' ' << getLastName()
102 << "\nsocial security number: " << getSocialSecurityNumber()
103 << "\ngross sales: " << getGrossSales()
104 << "\ncommission rate: " << getCommissionRate();
105 } // end function print
```

**Figure 12.24. BasePlusCommissionEmployee class header file.**

(This item is displayed on page 674 in the print version)

```

1 // Fig. 12.24: BasePlusCommissionEmployee.h
2 // BasePlusCommissionEmployee class derived from class
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // C++ standard string class
8 using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15 BasePlusCommissionEmployee(const string &, const string &,
16 const string &, double = 0.0, double = 0.0, double = 0.0);
17 ~BasePlusCommissionEmployee(); // destructor
18
19 void setBaseSalary(double); // set base salary
20 double getBaseSalary() const; // return base salary
21
22 double earnings() const; // calculate earnings
23 void print() const; // print BasePlusCommissionEmployee object
24 private:
25 double baseSalary; // base salary
26 }; // end class BasePlusCommissionEmployee
27
28 #endif

```

---

[Page 672]

In this example, we modified the `CommissionEmployee` constructor (lines 1021 of [Fig. 12.23](#)) and included a `CommissionEmployee` destructor (lines 2429), each of which outputs a line of text upon its invocation. We also modified the `BasePlusCommissionEmployee` constructor (lines 1122 of [Fig. 12.25](#)) and included a `BasePlusCommissionEmployee` destructor (lines 2530), each of which outputs a line of text upon its invocation.

---

[Page 674]

**Figure 12.25. `BasePlusCommissionEmployee`'s constructor outputs text.**

(This item is displayed on pages 674 - 675 in the print version)

```

1 // Fig. 12.25: BasePlusCommissionEmployee.cpp
2 // Class BasePlusCommissionEmployee member-function definitions.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // BasePlusCommissionEmployee class definition
8 #include "BasePlusCommissionEmployee.h"
9
10 // constructor
11 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
12 const string &first, const string &last, const string &ssn,
13 double sales, double rate, double salary)
14 // explicitly call base-class constructor
15 : CommissionEmployee(first, last, ssn, sales, rate)
16 {
17 setBaseSalary(salary); // validate and store base salary
18
19 cout << "BasePlusCommissionEmployee constructor: " << endl;
20 print();
21 cout << "\n\n";
22 } // end BasePlusCommissionEmployee constructor
23
24 // destructor
25 BasePlusCommissionEmployee::~BasePlusCommissionEmployee()
26 {
27 cout << "BasePlusCommissionEmployee destructor: " << endl;
28 print();
29 cout << "\n\n";
30 } // end BasePlusCommissionEmployee destructor
31
32 // set base salary
33 void BasePlusCommissionEmployee::setBaseSalary(double salary)
34 {
35 baseSalary = (salary < 0.0) ? 0.0 : salary;
36 } // end function setBaseSalary
37
38 // return base salary
39 double BasePlusCommissionEmployee::getBaseSalary() const
40 {
41 return baseSalary;
42 } // end function getBaseSalary
43
44 // calculate earnings
45 double BasePlusCommissionEmployee::earnings() const
46 {

```

```

47 return getBaseSalary() + CommissionEmployee::earnings();
48 } // end function earnings
49
50 // print BasePlusCommissionEmployee object
51 void BasePlusCommissionEmployee::print() const
52 {
53 cout << "base-salaried ";
54
55 // invoke CommissionEmployee's print function
56 CommissionEmployee::print();
57
58 cout << "\nbase salary: " << getBaseSalary();
59 } // end function print

```

[Page 676]

Figure 12.26 demonstrates the order in which constructors and destructors are called for objects of classes that are part of an inheritance hierarchy. Function `main` (lines 1534) begins by instantiating `CommissionEmployee` object `employee1` (lines 2122) in a separate block inside `main` (lines 2023). The object goes in and out of scope immediately (the end of the block is reached as soon as the object is created), so both the `CommissionEmployee` constructor and destructor are called. Next, lines 2627 instantiate `BasePlusCommissionEmployee` object `employee2`. This invokes the `CommissionEmployee` constructor to display outputs with values passed from the `BasePlusCommissionEmployee` constructor, then the output specified in the `BasePlusCommissionEmployee` constructor is performed. Lines 3031 then instantiate `BasePlusCommissionEmployee` object `employee3`. Again, the `CommissionEmployee` and `BasePlusCommissionEmployee` constructors are both called. Note that, in each case, the body of the `CommissionEmployee` constructor is executed before the body of the `BasePlusCommissionEmployee` constructor executes. When the end of `main` is reached, the destructors are called for objects `employee2` and `employee3`. But, because destructors are called in the reverse order of their corresponding constructors, the `BasePlusCommissionEmployee` destructor and `CommissionEmployee` destructor are called (in that order) for object `employee3`, then the `BasePlusCommissionEmployee` and `CommissionEmployee` destructors are called (in that order) for object `employee2`.

### Figure 12.26. Constructor and destructor call order.

(This item is displayed on pages 676 - 678 in the print version)

```
1 // Fig. 12.26: fig12_26.cpp
2 // Display order in which base-class and derived-class constructors
3 // and destructors are called.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 // BasePlusCommissionEmployee class definition
13 #include "BasePlusCommissionEmployee.h"
14
15 int main()
16 {
17 // set floating-point output formatting
18 cout << fixed << setprecision(2);
19
20 { // begin new scope
21 CommissionEmployee employee1(
22 "Bob", "Lewis", "333-33-3333", 5000, .04);
23 } // end scope
24
25 cout << endl;
26 BasePlusCommissionEmployee
27 employee2("Lisa", "Jones", "555-55-5555", 2000, .06, 800);
28
29 cout << endl;
30 BasePlusCommissionEmployee
31 employee3("Mark", "Sands", "888-88-8888", 8000, .15, 2000);
32 cout << endl;
33 return 0;
34 } // end main
```

```
CommissionEmployee constructor:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

```
CommissionEmployee destructor:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

```
CommissionEmployee constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
```

```
BasePlusCommissionEmployee constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00
```

```
CommissionEmployee constructor:
commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
```

```
BasePlusCommissionEmployee constructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00
```

```
BasePlusCommissionEmployee destructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00
```

```
CommissionEmployee destructor:
commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
```

```
BasePlusCommissionEmployee destructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00
```

```
CommissionEmployee destructor:
commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 678]

## 12.6. public, protected and private Inheritance

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. Use of protected and private inheritance is rare, and each should be used only with great care; we normally use public inheritance in this book. (Chapter 21 demonstrates private inheritance as an alternative to composition.) Figure 12.27 summarizes for each type of inheritance the accessibility of base-class members in a derived class. The first column contains the base-class access specifiers.

**Figure 12.27. Summary of base-class member accessibility in a derived class.**

(This item is displayed on page 679 in the print version)

[\[View full size image\]](#)

| Base-class member-access specifier | Type of inheritance                                                                                                                                                                  |                                                                                                                                                                                      |                                                                                                                                                                                      |
|------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                    | public inheritance                                                                                                                                                                   | protected inheritance                                                                                                                                                                | private inheritance                                                                                                                                                                  |
| public                             | <p><b>public</b> in derived class.</p> <p>Can be accessed directly by member functions, <b>friend</b> functions and nonmember functions.</p>                                         | <p><b>protected</b> in derived class.</p> <p>Can be accessed directly by member functions and <b>friend</b> functions.</p>                                                           | <p><b>private</b> in derived class.</p> <p>Can be accessed directly by member functions and <b>friend</b> functions.</p>                                                             |
| protected                          | <p><b>protected</b> in derived class.</p> <p>Can be accessed directly by member functions and <b>friend</b> functions.</p>                                                           | <p><b>protected</b> in derived class.</p> <p>Can be accessed directly by member functions and <b>friend</b> functions.</p>                                                           | <p><b>private</b> in derived class.</p> <p>Can be accessed directly by member functions and <b>friend</b> functions.</p>                                                             |
| private                            | <p>Hidden in derived class.</p> <p>Can be accessed by member functions and <b>friend</b> functions through <b>public</b> or <b>protected</b> member functions of the base class.</p> | <p>Hidden in derived class.</p> <p>Can be accessed by member functions and <b>friend</b> functions through <b>public</b> or <b>protected</b> member functions of the base class.</p> | <p>Hidden in derived class.</p> <p>Can be accessed by member functions and <b>friend</b> functions through <b>public</b> or <b>protected</b> member functions of the base class.</p> |

When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.

When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class. When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members (e.g., the functions become utility functions) of the derived class. **Private** and **protected** inheritance are not is-a relationships.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 679]

Sometimes it is difficult for students to appreciate the scope of problems faced by designers who work on large-scale software projects in industry. People experienced with such projects say that effective software reuse improves the software development process. Object-oriented programming facilitates software reuse, thus shortening development times and enhancing software quality.

The availability of substantial and useful class libraries delivers the maximum benefits of software reuse through inheritance. Just as shrink-wrapped software produced by independent software vendors became an explosive-growth industry with the arrival of the personal computer, so, too, interest in the creation and sale of class libraries is growing exponentially. Application designers build their applications with these libraries, and library designers are being rewarded by having their libraries included with the applications. The standard C++ libraries that are shipped with C++ compilers tend to be rather general purpose and limited in scope. However, there is massive worldwide commitment to the development of class libraries for a huge variety of applications arenas.

[Page 680]

#### Software Engineering Observation 12.9



At the design stage in an object-oriented system, the designer often determines that certain classes are closely related. The designer should "factor out" common attributes and behaviors and place these in a base class, then use inheritance to form derived classes, endowing them with capabilities beyond those inherited from the base class.

#### Software Engineering Observation 12.10



The creation of a derived class does not affect its base class's source code. Inheritance preserves the integrity of a base class.

#### Software Engineering Observation 12.11



Just as designers of non-object-oriented systems should avoid proliferation of functions, designers of object-oriented systems should avoid proliferation of classes. Proliferation of classes creates management problems and can hinder software reusability, because it becomes difficult for a client to locate the most appropriate class of a huge class library. The alternative is to create fewer classes that provide more substantial functionality, but such classes might provide too much functionality.

### Performance Tip 12.3



If classes produced through inheritance are larger than they need to be (i.e., contain too much functionality), memory and processing resources might be wasted. Inherit from the class whose functionality is "closest" to what is needed.

Reading derived-class definitions can be confusing, because inherited members are not shown physically in the derived classes, but nevertheless are present. A similar problem exists when documenting derived-class members.

[◀ PREV](#)**page footer**[NEXT ▶](#)

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 680 (continued)]

## 12.8. Wrap-Up

This chapter introduced inheritance—the ability to create a class by absorbing an existing class's data members and member functions and embellishing them with new capabilities. Through a series of examples using an employee inheritance hierarchy, you learned the notions of base classes and derived classes and used `public` inheritance to create a derived class that inherits members from a base class. The chapter introduced the access specifier `protected`; derived-class member functions can access `protected` base-class members. You learned how to access redefined base-class members by qualifying their names with the base-class name and binary scope resolution operator (`::`). You also saw the order in which constructors and destructors are called for objects of classes that are part of an inheritance hierarchy. Finally, we explained the three types of inheritance—`public`, `protected` and `private`—and the accessibility of base-class members in a derived class when using each type.

In [Chapter 13](#), Object-Oriented Programming: Polymorphism, we build upon our discussion of inheritance by introducing polymorphism—an object-oriented concept that enables us to write programs that handle, in a more general manner, objects of a wide variety of classes related by inheritance. After studying [Chapter 13](#), you will be familiar with classes, objects, encapsulation, inheritance and polymorphism—the essential aspects of object-oriented programming.

[◀ PREV](#)[NEXT ▶](#)

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 682]

- When an object of a derived class is instantiated, the base class's constructor is called immediately (either explicitly or implicitly) to initialize the base-class data members in the derived-class object (before the derived-class data members are initialized).
- Declaring data members `private`, while providing non-`private` member functions to manipulate and perform validation checking on this data, enforces good software engineering.
- When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors first the derived-class destructor is called, then the base-class destructor is called.
- When deriving a class from a base class, the base class may be declared as either `public`, `protected` or `private`.
- When deriving a class from a `public` base class, `public` members of the base class become `public` members of the derived class, and `protected` members of the base class become `protected` members of the derived class.
- When deriving a class from a `protected` base class, `public` and `protected` members of the base class become `protected` members of the derived class.
- When deriving a class from a `private` base class, `public` and `protected` members of the base class become `private` members of the derived class.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 682 (continued)]

## Terminology

base class

base-class constructor

base-class default constructor

base-class destructor

base-class initializer

brittle software

class hierarchy

composition

customize software

derived class

derived-class constructor

derived-class destructor

direct base class

fragile software

friend of a base class

friend of a derived class

has-a relationship

hierarchical relationship

indirect base class

inherit the members of an existing class

inheritance

is-a relationship

multiple inheritance

private base class

private inheritance

protected base class

protected inheritance

protected keyword

protected member of a class

public base class

public inheritance

qualified name

redefine a base-class member function

single inheritance

subclass

superclass

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 683]

- In single inheritance, a class exists in a(n) \_\_\_\_\_ relationship with its derived classes.
- A base class's \_\_\_\_\_ members are accessible within that base class and anywhere that the program has a handle to an object of that base class or to an object of one of its derived classes.
- A base class's protected access members have a level of protection between those of public and \_\_\_\_\_ access.
- C++ provides for \_\_\_\_\_, which allows a derived class to inherit from many base classes, even if these base classes are unrelated.
- When an object of a derived class is instantiated, the base class's \_\_\_\_\_ is called implicitly or explicitly to do any necessary initialization of the base-class data members in the derived-class object.
- When deriving a class from a base class with public inheritance, public members of the base class become \_\_\_\_\_ members of the derived class, and protected members of the base class become \_\_\_\_\_ members of the derived class.
- When deriving a class from a base class with protected inheritance, public members of the base class become \_\_\_\_\_ members of the derived class, and protected members of the base class become \_\_\_\_\_ members of the derived class.

## 12.2

State whether each of the following is true or false. If false, explain why.

a.

Base-class constructors are not inherited by derived classes.

b.

A has-a relationship is implemented via inheritance.

c.

A Car class has an is-a relationship with the SteeringWheel and Brakes classes.

d.

Inheritance encourages the reuse of proven high-quality software.

e.

When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 683 (continued)]

## Answers to Self-Review Exercises

**12.1** a) Inheritance. b) protected. c) is-a or inheritance. d) has-a or composition or aggregation. e) hierarchical. f) public. g) private. h) multiple inheritance. i) constructor. j) public, protected. k) protected, protected.

**12.2** a) True. b) False. A has-a relationship is implemented via composition. An is-a relationship is implemented via inheritance. c) False. This is an example of a has-a relationship. Class `Car` has an is-a relationship with class `Vehicle`. d) True. e) True.

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 684]

## 12.7

The world of shapes is much richer than the shapes included in the inheritance hierarchy of Fig. 12.3. Write down all the shapes you can think of both two-dimensional and three-dimensional and form them into a more complete Shape hierarchy with as many levels as possible. Your hierarchy should have base class Shape from which class TwoDimensionalShape and class ThreeDimensionalShape are derived. [Note: You do not need to write any code for this exercise.] We will use this hierarchy in the exercises of Chapter 13 to process a set of distinct shapes as objects of base-class Shape. (This technique, called polymorphism, is the subject of Chapter 13.)

## 12.8

Draw an inheritance hierarchy for classes Quadrilateral, TRapezoid, Parallelogram, Rectangle and Square. Use Quadrilateral as the base class of the hierarchy. Make the hierarchy as deep as possible.

## 12.9

(Package Inheritance Hierarchy) Package-delivery services, such as FedEx®, DHL® and UPS®, offer a number of different shipping options, each with specific costs associated. Create an inheritance hierarchy to represent various types of packages. Use Package as the base class of the hierarchy, then include classes TwoDayPackage and OvernightPackage that derive from Package. Base class Package should include data members representing the name, address, city, state and ZIP code for both the sender and the recipient of the package, in addition to data members that store the weight (in ounces) and cost per ounce to ship the package. Package's constructor should initialize these data members. Ensure that the weight and cost per ounce contain positive values. Package should provide a public member function calculateCost that returns a double indicating the cost associated with shipping the package. Package's calculateCost function should determine the cost by multiplying the weight by the cost per ounce. Derived class TwoDayPackage should inherit the functionality of base class Package, but also include a data member that represents a flat fee that the shipping company charges for two-day-delivery service. TwoDayPackage's constructor should receive a value to initialize this data member. TwoDayPackage should redefine member function calculateCost so that it computes the shipping cost by adding the flat fee to the weight-based cost calculated by base class Package's calculateCost function. Class OvernightPackage should inherit directly from class Package and contain an additional data member representing an additional fee per ounce charged for overnight-delivery service. OvernightPackage should redefine member function calculateCost so that it adds the additional fee per ounce to the standard cost per ounce before calculating the shipping cost. Write a test program that creates objects of each type of Package and tests member function calculateCost.

## 12.10

(Account Inheritance Hierarchy) Create an inheritance hierarchy that a bank might use to represent customers' bank accounts. All customers at this bank can deposit (i.e., credit) money into their accounts and withdraw (i.e., debit) money from their accounts. More specific types of accounts also exist. Savings accounts, for instance, earn interest on the money they hold. Checking accounts, on the other hand, charge a fee per transaction (i.e., credit or debit).

Create an inheritance hierarchy containing base class `Account` and derived classes `SavingsAccount` and `CheckingAccount` that inherit from class `Account`. Base class `Account` should include one data member of type `double` to represent the account balance. The class should provide a constructor that receives an initial balance and uses it to initialize the data member. The constructor should validate the initial balance to ensure that it is greater than or equal to `0.0`. If not, the balance should be set to `0.0` and the constructor should display an error message, indicating that the initial balance was invalid. The class should provide three member functions. Member function `credit` should add an amount to the current balance. Member function `debit` should withdraw money from the `Account` and ensure that the debit amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the function should print the message "Debit amount exceeded account balance." Member function `getBalance` should return the current balance.

---

[Page 685]

Derived class `SavingsAccount` should inherit the functionality of an `Account`, but also include a data member of type `double` indicating the interest rate (percentage) assigned to the `Account`. `SavingsAccount`'s constructor should receive the initial balance, as well as an initial value for the `SavingsAccount`'s interest rate. `SavingsAccount` should provide a public member function `calculateInterest` that returns a `double` indicating the amount of interest earned by an account. Member function `calculateInterest` should determine this amount by multiplying the interest rate by the account balance. [Note: `SavingsAccount` should inherit member functions `credit` and `debit` as is without redefining them.]

Derived class `CheckingAccount` should inherit from base class `Account` and include an additional data member of type `double` that represents the fee charged per transaction. `CheckingAccount`'s constructor should receive the initial balance, as well as a parameter indicating a fee amount. Class `CheckingAccount` should redefine member functions `credit` and `debit` so that they subtract the fee from the account balance whenever either transaction is performed successfully. `CheckingAccount`'s versions of these functions should invoke the base-class `Account` version to perform the updates to an account balance. `CheckingAccount`'s `debit` function should charge a fee only if money is actually withdrawn (i.e., the debit amount does not exceed the account balance). [Hint: Define `Account`'s `debit` function so that it returns a `bool` indicating whether money was withdrawn. Then use the return value to determine whether a fee should be charged.]

After defining the classes in this hierarchy, write a program that creates objects of each class and tests their member functions. Add interest to the `SavingsAccount` object by first invoking its `calculateInterest` function, then passing the returned interest amount to the object's `credit`

function.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 687]

## Outline

### 13.1 Introduction

### 13.2 Polymorphism Examples

### 13.3 Relationships Among Objects in an Inheritance Hierarchy

#### 13.3.1 Invoking Base-Class Functions from Derived-Class Objects

#### 13.3.2 Aiming Derived-Class Pointers at Base-Class Objects

#### 13.3.3 Derived-Class Member-Function Calls via Base-Class Pointers

#### 13.3.4 Virtual Functions

#### 13.3.5 Summary of the Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers

### 13.4 Type Fields and switch Statements

### 13.5 Abstract Classes and Pure virtual Functions

### 13.6 Case Study: Payroll System Using Polymorphism

#### 13.6.1 Creating Abstract Base Class Employee

#### 13.6.2 Creating Concrete Derived Class SalariedEmployee

#### 13.6.3 Creating Concrete Derived Class HourlyEmployee

#### 13.6.4 Creating Concrete Derived Class CommissionEmployee

#### 13.6.5 Creating Indirect Concrete Derived Class BasePlusCommissionEmployee

### 13.6.6 Demonstrating Polymorphic Processing

## 13.7 (Optional) Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

### 13.8 Case Study: Payroll System Using Polymorphism and Run-Time Type Information with Downcasting, dynamic\_cast, typeid and type\_info

## 13.9 Virtual Destructors

## 13.10 (Optional) Software Engineering Case Study: Incorporating Inheritance into the ATM System

## 13.11 Wrap-Up

### Summary

### Terminology

### Self-Review Exercises

### Answers to Self-Review Exercises

### Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 688]

With polymorphism, we can design and implement systems that are easily extensible new classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically. The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that the programmer adds to the hierarchy. For example, if we create class `Tortoise` that inherits from class `Animal` (which might respond to a `move` message by crawling one inch), we need to write only the `Tortoise` class and the part of the simulation that instantiates a `Tortoise` object. The portions of the simulation that process each `Animal` generically can remain the same.

We begin with a sequence of small, focused examples that lead up to an understanding of virtual functions and dynamic binding polymorphism's two underlying technologies. We then present a case study that revisits [Chapter 12](#)'s `Employee` hierarchy. In the case study, we define a common "interface" (i.e., set of functionality) for all the classes in the hierarchy. This common functionality among employees is defined in a so-called abstract base class, `Employee`, from which classes `SalariedEmployee`, `HourlyEmployee` and `CommissionEmployee` inherit directly and class `BaseCommissionEmployee` inherits indirectly. We will soon see what makes a class "abstract" or its opposite "concrete."

In this hierarchy, every employee has an `earnings` function to calculate the employee's weekly pay. These `earnings` functions vary by employee type for instance, `SalariedEmployees` are paid a fixed weekly salary regardless of the number of hours worked, while `HourlyEmployees` are paid by the hour and receive overtime pay. We show how to process each employee "in the general" that is, using base-class pointers to call the `earnings` function of several derived-class objects. This way, the programmer needs to be concerned with only one type of function call, which can be used to execute several different functions based on the objects referred to by the base-class pointers.

A key feature of this chapter is its (optional) detailed discussion of polymorphism, virtual functions and dynamic binding "under the hood," which uses a detailed diagram to explain how polymorphism can be implemented in C++.

Occasionally, when performing polymorphic processing, we need to program "in the specific," meaning that operations need to be performed on a specific type of object in a hierarchy the operation cannot be generally applied to several types of objects. We reuse our `Employee` hierarchy to demonstrate the powerful capabilities of **run-time type information (RTTI)** and **dynamic casting**, which enable a program to determine the type of an object at execution time and act on that object accordingly. We use these capabilities to determine whether a particular employee object is a `BasePlusCommissionEmployee`, then give that employee a 10 percent bonus on his or her base salary.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 689]

## 13.2. Polymorphism Examples

In this section, we discuss several polymorphism examples. With polymorphism, one function can cause different actions to occur, depending on the type of the object on which the function is invoked. This gives the programmer tremendous expressive capability. If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral` object. Therefore, any operation (such as calculating the perimeter or the area) that can be performed on an object of class `Quadrilateral` also can be performed on an object of class `Rectangle`. Such operations also can be performed on other kinds of `Quadrilaterals`, such as `Squares`, `Parallelograms` and `TRapezoids`. The polymorphism occurs when a program invokes a `virtual` function through a base-class (i.e., `Quadrilateral`) pointer or referenceC++ dynamically (i.e., at execution time) chooses the correct function for the class from which the object was instantiated. You will see a code example that illustrates this process in [Section 13.3](#).

As another example, suppose that we design a video game that manipulates objects of many different types, including objects of classes `Martian`, `Venutian`, `Plutonian`, `SpaceShip` and `LaserBeam`. Imagine that each of these classes inherits from the common base class `SpaceObject`, which contains member function `draw`. Each derived class implements this function in a manner appropriate for that class. A screen-manager program maintains a container (e.g., a `vector`) that holds `SpaceObject` pointers to objects of the various classes. To refresh the screen, the screen manager periodically sends each object the same message namely, `draw`. Each type of object responds in a unique way. For example, a `Martian` object might draw itself in red with the appropriate number of antennae. A `SpaceShip` object might draw itself as a silver flying saucer. A `LaserBeam` object might draw itself as a bright red beam across the screen. Again, the same message (in this case, `draw`) sent to a variety of objects has "many forms" of results.

A polymorphic screen manager facilitates adding new classes to a system with minimal modifications to its code. Suppose that we want to add objects of class `Mercurian` to our video game. To do so, we must build a class `Mercurian` that inherits from `SpaceObject`, but provides its own definition of member function `draw`. Then, when pointers to objects of class `Mercurian` appear in the container, the programmer does not need to modify the code for the screen manager. The screen manager invokes member function `draw` on every object in the container, regardless of the object's type, so the new `Mercurian` objects simply "plug right in." Thus, without modifying the system (other than to build and include the classes themselves), programmers can use polymorphism to accommodate additional classes, including ones that were not even envisioned when the system was created.

Software Engineering Observation 13.1



With virtual functions and polymorphism, you can deal in generalities and let the execution time environment concern itself with the specifics. You can direct a variety of objects to behave in manners appropriate to those objects without even knowing their types (as long as those objects belong to the same inheritance hierarchy and are being accessed off a common base-class pointer).

## Software Engineering Observation 13.2



Polymorphism promotes extensibility: Software written to invoke polymorphic behavior is written independently of the types of the objects to which messages are sent. Thus, new types of objects that can respond to existing messages can be incorporated into such a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 691]

**Figure 13.1. CommissionEmployee class header file.**

```
1 // Fig. 13.1: CommissionEmployee.h
2 // CommissionEmployee class definition represents a commission employee.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // C++ standard string class
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12 CommissionEmployee(const string &, const string &, const string &,
13 double = 0.0, double = 0.0);
14
15 void setFirstName(const string &); // set first name
16 string getFirstName() const; // return first name
17
18 void setLastName(const string &); // set last name
19 string getLastname() const; // return last name
20
21 void setSocialSecurityNumber(const string &); // set SSN
22 string getSocialSecurityNumber() const; // return SSN
23
24 void setGrossSales(double); // set gross sales amount
25 double getGrossSales() const; // return gross sales amount
26
27 void setCommissionRate(double); // set commission rate
28 double getCommissionRate() const; // return commission rate
29
30 double earnings() const; // calculate earnings
31 void print() const; // print CommissionEmployee object
32 private:
33 string firstName;
34 string lastName;
35 string socialSecurityNumber;
36 double grossSales; // gross weekly sales
37 double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

**Figure 13.2. CommissionEmployee class implementation file.**

(This item is displayed on pages 692 - 693 in the print version)

```
1 // Fig. 13.2: CommissionEmployee.cpp
2 // Class CommissionEmployee member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8 // constructor
9 CommissionEmployee::CommissionEmployee(
10 const string &first, const string &last, const string &ssn,
11 double sales, double rate)
12 : firstName(first), lastName(last), socialSecurityNumber(ssn)
13 {
14 setGrossSales(sales); // validate and store gross sales
15 setCommissionRate(rate); // validate and store commission rate
16 } // end CommissionEmployee constructor
17
18 // set first name
19 void CommissionEmployee::setFirstName(const string &first)
20 {
21 firstName = first; // should validate
22 } // end function setFirstName
23
24 // return first name
25 string CommissionEmployee::getFirstName() const
26 {
27 return firstName;
28 } // end function getFirstName
29
30 // set last name
31 void CommissionEmployee::setLastName(const string &last)
32 {
33 lastName = last; // should validate
34 } // end function setLastName
35
36 // return last name
37 string CommissionEmployee::getLastName() const
38 {
39 return lastName;
```

```

40 } // end function getLastname
41
42 // set social security number
43 void CommissionEmployee::setSocialSecurityNumber(const string &ssn)
44 {
45 socialSecurityNumber = ssn; // should validate
46 } // end function setSocialSecurityNumber
47
48 // return social security number
49 string CommissionEmployee::getSocialSecurityNumber() const
50 {
51 return socialSecurityNumber;
52 } // end function getSocialSecurityNumber
53
54 // set gross sales amount
55 void CommissionEmployee::setGrossSales(double sales)
56 {
57 grossSales = (sales < 0.0) ? 0.0 : sales;
58 } // end function setGrossSales
59
60 // return gross sales amount
61 double CommissionEmployee::getGrossSales() const
62 {
63 return grossSales;
64 } // end function getGrossSales
65
66 // set commission rate
67 void CommissionEmployee::setCommissionRate(double rate)
68 {
69 commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;
70 } // end function setCommissionRate
71
72 // return commission rate
73 double CommissionEmployee::getCommissionRate() const
74 {
75 return commissionRate;
76 } // end function getCommissionRate
77
78 // calculate earnings
79 double CommissionEmployee::earnings() const
80 {
81 return getCommissionRate() * getGrossSales();
82 } // end function earnings
83
84 // print CommissionEmployee object
85 void CommissionEmployee::print() const
86 {
87 cout << "commission employee: "
88 << getFirstName() << ' ' << getLastname()
89 << "\nsocial security number: " << getSocialSecurityNumber()

```

```

90 << "\ngross sales: " << getGrossSales()
91 << "\ncommission rate: " << getCommissionRate();
92 } // end function print

```

[Page 693]

**Figure 13.3. BasePlusCommissionEmployee class header file.**

(This item is displayed on pages 693 - 694 in the print version)

```

1 // Fig. 13.3: BasePlusCommissionEmployee.h
2 // BasePlusCommissionEmployee class derived from class
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // C++ standard string class
8 using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15 BasePlusCommissionEmployee(const string &, const string &,
16 const string &, double = 0.0, double = 0.0, double = 0.0);
17
18 void setBaseSalary(double); // set base salary
19 double getBaseSalary() const; // return base salary
20
21 double earnings() const; // calculate earnings
22 void print() const; // print BasePlusCommissionEmployee object
23 private:
24 double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif

```

[Page 694]

**Figure 13.4. BasePlusCommissionEmployee class implementation file.**

(This item is displayed on pages 694 - 695 in the print version)

```

1 // Fig. 13.4: BasePlusCommissionEmployee.cpp
2 // Class BasePlusCommissionEmployee member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 // BasePlusCommissionEmployee class definition
7 #include "BasePlusCommissionEmployee.h"
8
9 // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11 const string &first, const string &last, const string &ssn,
12 double sales, double rate, double salary)
13 // explicitly call base-class constructor
14 : CommissionEmployee(first, last, ssn, sales, rate)
15 {
16 setBaseSalary(salary); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary(double salary)
21 {
22 baseSalary = (salary < 0.0) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28 return baseSalary;
29 } // end function getBaseSalary
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const
33 {
34 return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee object
38 void BasePlusCommissionEmployee::print() const
39 {
40 cout << "base-salaried ";
41
42 // invoke CommissionEmployee's print function
43 CommissionEmployee::print();
44
45 cout << "\nbase salary: " << getBaseSalary();
46 } // end function print

```

---

[Page 695]

In Fig. 13.5, lines 1920 create a `CommissionEmployee` object and line 23 creates a pointer to a `CommissionEmployee` object; lines 2627 create a `BasePlusCommissionEmployee` object and line 30 creates a pointer to a `BasePlusCommissionEmployee` object. Lines 37 and 39 use each object's name (`CommissionEmployee` and `BasePlusCommissionEmployee`, respectively) to invoke each object's `print` member function. Line 42 assigns the address of base-class object `CommissionEmployee` to base-class pointer `CommissionEmployeePtr`, which line 45 uses to invoke member function `print` on that `CommissionEmployee` object. This invokes the version of `print` defined in base class `CommissionEmployee`. Similarly, line 48 assigns the address of derived-class object `BasePlusCommissionEmployee` to derived-class pointer `BasePlusCommissionEmployeePtr`, which line 52 uses to invoke member function `print` on that `BasePlusCommissionEmployee` object. This invokes the version of `print` defined in derived class `BasePlusCommissionEmployee`. Line 55 then assigns the address of derived-class object `BasePlusCommissionEmployee` to base-class pointer `CommissionEmployeePtr`, which line 59 uses to invoke member function `print`. The C++ compiler allows this "crossover" because an object of a derived class is an object of its base class. Note that despite the fact that the base class `CommissionEmployee` pointer points to a derived class `BasePlusCommissionEmployee` object, the base class `CommissionEmployee`'s `print` member function is invoked (rather than `BasePlusCommissionEmployee`'s `print` function). The output of each `print` member-function invocation in this program reveals that the invoked functionality depends on the type of the handle (i.e., the pointer or reference type) used to invoke the function, not the type of the object to which the handle points. In Section 13.3.4, when we introduce virtual functions, we demonstrate that it is possible to invoke the object type's functionality, rather than invoke the handle type's functionality. We will see that this is crucial to implementing polymorphic behaviorthe key topic of this chapter.

---

[Page 697]

**Figure 13.5. Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers.**

(This item is displayed on pages 695 - 697 in the print version)

```

1 // Fig. 13.5: fig13_05.cpp
2 // Aiming base-class and derived-class pointers at base-class
3 // and derived-class objects, respectively.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 // include class definitions
13 #include "CommissionEmployee.h"
14 #include "BasePlusCommissionEmployee.h"
15
16 int main()
17 {
18 // create base-class object
19 CommissionEmployee commissionEmployee(
20 "Sue", "Jones", "222-22-2222", 10000, .06);
21
22 // create base-class pointer
23 CommissionEmployee *commissionEmployeePtr = 0;
24
25 // create derived-class object
26 BasePlusCommissionEmployee basePlusCommissionEmployee(
27 "Bob", "Lewis", "333-33-3333", 5000, .04, 300);
28
29 // create derived-class pointer
30 BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
31
32 // set floating-point output formatting
33 cout << fixed << setprecision(2);
34
35 // output objects commissionEmployee and basePlusCommissionEmployee
36 cout << "Print base-class and derived-class objects:\n\n";
37 commissionEmployee.print(); // invokes base-class print
38 cout << "\n\n";
39 basePlusCommissionEmployee.print(); // invokes derived-class print
40
41 // aim base-class pointer at base-class object and print
42 commissionEmployeePtr = &commissionEmployee; // perfectly natural
43 cout << "\n\n\nCalling print with base-class pointer to "
44 << "\nbase-class object invokes base-class print function:\n\n";
45 commissionEmployeePtr->print(); // invokes base-class print
46
47 // aim derived-class pointer at derived-class object and print
48 basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
49 cout << "\n\n\nCalling print with derived-class pointer to "
50 << "\nderived-class object invokes derived-class "

```

```

51 << "print function:\n\n";
52 basePlusCommissionEmployeePtr->print(); // invokes derived-class print
53
54 // aim base-class pointer at derived-class object and print
55 commissionEmployeePtr = &basePlusCommissionEmployee;
56 cout << "\n\n\nCalling print with base-class pointer to "
57 << "derived-class object\ninvokes base-class print "
58 << "function on that derived-class object:\n\n";
59 commissionEmployeePtr->print(); // invokes base-class print
60 cout << endl;
61 return 0;
62 } // end main

```

Print base-class and derived-class objects:

```

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

```

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

```

Calling print with base-class pointer to  
base-class object invokes base-class print function:

```

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

```

Calling print with derived-class pointer to  
derived-class object invokes derived-class print function:

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

```

Calling print with base-class pointer to derived-class object  
invokes base-class print function on that derived-class object:

```

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00

```

```
commission rate: 0.04
```

[Page 698]

### 13.3.2. Aiming Derived-Class Pointers at Base-Class Objects

In [Section 13.3.1](#), we assigned the address of a derived-class object to a base-class pointer and explained that the C++ compiler allows this assignment, because a derived-class object is a base-class object. We take the opposite approach in [Fig. 13.6](#), as we aim a derived-class pointer at a base-class object. [Note: This program uses classes `CommissionEmployee` and `BasePlusCommissionEmployee` of [Figs. 13.113.4](#)] Lines 89 of [Fig. 13.6](#) create a `CommissionEmployee` object, and line 10 creates a `BasePlusCommissionEmployee` pointer. Line 14 attempts to assign the address of base-class object `commissionEmployee` to derived-class pointer `basePlusCommissionEmployeePtr`, but the C++ compiler generates an error. The compiler prevents this assignment, because a `CommissionEmployee` is not a `BasePlusCommissionEmployee`. Consider the consequences if the compiler were to allow this assignment. Through a `BasePlusCommissionEmployee` pointer, we can invoke every `BasePlusCommissionEmployee` member function, including `setBaseSalary`, for the object to which the pointer points (i.e., the base-class object `commissionEmployee`). However, the `CommissionEmployee` object does not provide a `setBaseSalary` member function, nor does it provide a `baseSalary` data member to set. This could lead to problems, because member function `setBaseSalary` would assume that there is a `baseSalary` data member to set at its "usual location" in a `BasePlusCommissionEmployee` object. This memory does not belong to the `CommissionEmployee` object so member function `setBaseSalary` might overwrite other important data in memory, possibly data that belongs to a different object.

**Figure 13.6. Aiming a derived-class pointer at a base-class object.**

(This item is displayed on pages 698 - 699 in the print version)

```

1 // Fig. 13.6: fig13_06.cpp
2 // Aiming a derived-class pointer at a base-class object.
3 #include "CommissionEmployee.h"
4 #include "BasePlusCommissionEmployee.h"
5
6 int main()
7 {
8 CommissionEmployee commissionEmployee(
9 "Sue", "Jones", "222-22-2222", 10000, .06);
10 BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
11
12 // aim derived-class pointer at base-class object
13 // Error: a CommissionEmployee is not a BasePlusCommissionEmployee

```

```

14 basePlusCommissionEmployeePtr = &commissionEmployee;
15
16 } // end main

```

Borland C++ command-line compiler error messages:

```
Error E2034 Fig13_06\fig13_06.cpp 14: Cannot convert 'CommissionEmployee *'
to 'BasePlusCommissionEmployee *' in function main()
```

GNU C++ compiler error messages:

```
fig13_06.cpp:14: error: invalid conversion from `CommissionEmployee*' to
`BasePlusCommissionEmployee*'
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphttp5_examples\ch13\Fig13_06\fig13_06.cpp(14) : error C2440:
'=' : cannot convert from 'CommissionEmployee *__w64' to
'BasePlusCommissionEmployee *'
 Cast from base to derived requires dynamic_cast or static_cast
```

---

[Page 699]

### 13.3.3. Derived-Class Member-Function Calls via Base-Class Pointers

Off a base-class pointer, the compiler allows us to invoke only bases-class member functions. Thus, if a base-class pointer is aimed at a derived-class object, and an attempt is made to access a derived-class-only member function, a compilation error will occur.

Figure 13.7 shows the consequences of attempting to invoke a derived-class member function off a base-class pointer. [Note: We are again using classes `CommissionEmployee` and `BasePlusCommissionEmployee` of Figs. 13.113.4] Line 9 creates `commissionEmployeePtra` pointer to a `CommissionEmployee` object and

lines 1011 create a `BasePlusCommissionEmployee` object. Line 14 aims `commissionEmployeePtr` at derived-class object `basePlusCommissionEmployee`. Recall from [Section 13.3.1](#) that the C++ compiler allows this, because a `BasePlusCommissionEmployee` is a `CommissionEmployee` (in the sense that a `BasePlusCommissionEmployee` object contains all the functionality of a `CommissionEmployee` object). Lines 1822 invoke base-class member functions `getFirstName`, `getLastName`, `getSocialSecurityNumber`, `getGrossSales` and `getCommissionRate` off the base-class pointer. All of these calls are legitimate, because `BasePlusCommissionEmployee` inherits these member functions from `CommissionEmployee`. We know that `commissionEmployeePtr` is aimed at a `BasePlusCommissionEmployee` object, so in lines 2627 we attempt to invoke `BasePlusCommissionEmployee` member functions `getBaseSalary` and `setBaseSalary`. The C++ compiler generates errors on both of these lines, because these are not member functions of base-class `CommissionEmployee`. The handle can invoke only those functions that are members of that handle's associated class type. (In this case, off a `CommissionEmployee *`, we can invoke only `CommissionEmployee` member functions `setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber`, `getSocialSecurityNumber`, `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` and `print`.)

---

[Page 700]

**Figure 13.7. Attempting to invoke derived-class-only functions via a base-class pointer.**

(This item is displayed on pages 699 - 700 in the print version)

```

1 // Fig. 13.7: fig13_07.cpp
2 // Attempting to invoke derived-class-only member functions
3 // through a base-class pointer.
4 #include "CommissionEmployee.h"
5 #include "BasePlusCommissionEmployee.h"
6
7 int main()
8 {
9 CommissionEmployee *commissionEmployeePtr = 0; // base class
10 BasePlusCommissionEmployee basePlusCommissionEmployee(
11 "Bob", "Lewis", "333-33-3333", 5000, .04, 300); // derived class
12
13 // aim base-class pointer at derived-class object
14 commissionEmployeePtr = &basePlusCommissionEmployee;
15
16 // invoke base-class member functions on derived-class
17 // object through base-class pointer
18 string firstName = commissionEmployeePtr->getFirstName();
19 string lastName = commissionEmployeePtr->getLastName();
20 string ssn = commissionEmployeePtr->getSocialSecurityNumber();
21 double grossSales = commissionEmployeePtr->getGrossSales();
22 double commissionRate = commissionEmployeePtr->getCommissionRate();
23
24 // attempt to invoke derived-class-only member functions

```

```

25 // on derived-class object through base-class pointer
26 double baseSalary = commissionEmployeePtr->getBaseSalary();
27 commissionEmployeePtr->setBaseSalary(500);
28 return 0;
29 } // end main

```

Borland C++ command-line compiler error messages:

```

Error E2316 Fig13_07\fig13_07.cpp 26: 'getBaseSalary' is not a member of
'CommissionEmployee' in function main()
Error E2316 Fig13_07\fig13_07.cpp 27: 'setBaseSalary' is not a member of
'CommissionEmployee' in function main()

```

Microsoft Visual C++.NET compiler error messages:

```

C:\cpphttp5_examples\ch13\Fig13_07\fig13_07.cpp(26) : error C2039:
 'getBaseSalary' : is not a member of 'CommissionEmployee'
 C:\cpphttp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
 see declaration of 'CommissionEmployee'
C:\cpphttp5_examples\ch13\Fig13_07\fig13_07.cpp(27) : error C2039:
 'setBaseSalary' : is not a member of 'CommissionEmployee'
 C:\cpphttp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
 see declaration of 'CommissionEmployee'

```

GNU C++ compiler error messages:

```

fig13_07.cpp:26: error: `getBaseSalary' undeclared (first use this function)
fig13_07.cpp:26: error: (Each undeclared identifier is reported only once for
each function it appears in.)
fig13_07.cpp:27: error: `setBaseSalary' undeclared (first use this function)

```

It turns out that the C++ compiler does allow access to derived-class-only members from a base-class pointer that is aimed at a derived-class object if we explicitly cast the base-class pointer to a derived-class pointer a technique known as **downcasting**. As you learned in Section 13.3.1, it is possible to aim a base-class pointer

at a derived-class object. However, as we demonstrated in Fig. 13.7, a base-class pointer can be used to invoke only the functions declared in the base class. Downcasting allows a program to perform a derived-class-specific operation on a derived-class object pointed to by a base-class pointer. After a downcast, the program can invoke derived-class functions that are not in the base class. We will show you a concrete example of downcasting in Section 13.8.

[Page 701]

## Software Engineering Observation 13.3



If the address of a derived-class object has been assigned to a pointer of one of its direct or indirect base classes, it is acceptable to cast that base-class pointer back to a pointer of the derived-class type. In fact, this must be done to send that derived-class object messages that do not appear in the base class.

### 13.3.4. Virtual Functions

In Section 13.3.1, we aimed a base-class `CommissionEmployee` pointer at a derived-class `BasePlusCommissionEmployee` object, then invoked member function `print` through that pointer. Recall that the type of the handle determines which class's functionality to invoke. In that case, the `CommissionEmployee` pointer invoked the `CommissionEmployee` member function `print` on the `BasePlusCommissionEmployee` object, even though the pointer was aimed at a `BasePlusCommissionEmployee` object that has its own customized `print` function. With virtual functions, the type of the object being pointed to, not the type of the handle, determines which version of a virtual function to invoke.

First, we consider why virtual functions are useful. Suppose that a set of shape classes such as `Circle`, `triangle`, `Rectangle` and `Square` are all derived from base class `Shape`. Each of these classes might be endowed with the ability to draw itself via a member function `draw`. Although each class has its own `draw` function, the function for each shape is quite different. In a program that draws a set of shapes, it would be useful to be able to treat all the shapes generically as objects of the base class `Shape`. Then, to draw any shape, we could simply use a base-class `Shape` pointer to invoke function `draw` and let the program determine **dynamically** (i.e., at runtime) which derived-class `draw` function to use, based on the type of the object to which the base-class `Shape` pointer points at any given time.

To enable this kind of behavior, we declare `draw` in the base class as a **virtual function**, and we **override** `draw` in each of the derived classes to draw the appropriate shape. From an implementation perspective, overriding a function is no different than redefining one (which is the approach we have been using until now). An overridden function in a derived class has the same signature and return type (i.e., prototype) as the function it overrides in its base class. If we do not declare the base-class function as `virtual`, we can redefine that function. By contrast, if we declare the base-class function as `virtual`, we can override that function to enable polymorphic behavior. We declare a virtual function by preceding the function's prototype with the keyword `virtual` in the base class. For example,

```
virtual void draw() const;
```

would appear in base class `Shape`. The preceding prototype declares that function `draw` is a `virtual` function that takes no arguments and returns nothing. The function is declared `const` because a `draw` function typically would not make changes to the `Shape` object on which it is invoked. Virtual functions do not necessarily have to be `const` functions.

#### Software Engineering Observation 13.4



Once a function is declared `virtual`, it remains `virtual` all the way down the inheritance hierarchy from that point, even if that function is not explicitly declared `virtual` when a class overrides it.

[Page 702]

#### Good Programming Practice 13.1



Even though certain functions are implicitly `virtual` because of a declaration made higher in the class hierarchy, explicitly declare these functions `virtual` at every level of the hierarchy to promote program clarity.

#### Error-Prevention Tip 13.1



When a programmer browses a class hierarchy to locate a class to reuse, it is possible that a function in that class will exhibit `virtual` function behavior even though it is not explicitly declared `virtual`. This happens when the class inherits a `virtual` function from its base class, and it can lead to subtle logic errors. Such errors can be avoided by explicitly declaring all `virtual` functions `virtual` throughout the inheritance hierarchy.

#### Software Engineering Observation 13.5



When a derived class chooses not to override a `virtual` function from its base class, the derived class simply inherits its base class's `virtual` function implementation.

If the program invokes a `virtual` function through a base-class pointer to a derived-class object (e.g., `shapePtr->draw()`), the program will choose the correct derived-class `draw` function dynamically (i.e., at

execution time) based on the object type not the pointer type. Choosing the appropriate function to call at execution time (rather than at compile time) is known as **dynamic binding** or **late binding**.

When a virtual function is called by referencing a specific object by name and using the dot member-selection operator (e.g., `squareObject.draw()`), the function invocation is resolved at compile time (this is called **static binding**) and the virtual function that is called is the one defined for (or inherited by) the class of that particular object this is not polymorphic behavior. Thus, dynamic binding with virtual functions occurs only off pointer (and, as we will soon see, reference) handles.

Now let's see how virtual functions can enable polymorphic behavior in our employee hierarchy. Figures 13.813.9 are the header files for classes `CommissionEmployee` and `BasePlusCommissionEmployee`, respectively. Note that the only difference between these files and those of Fig. 13.1 and Fig. 13.3 is that we specify each class's `earnings` and `print` member functions as `virtual` (lines 3031 of Fig. 13.8 and lines 2122 of Fig. 13.9). Because functions `earnings` and `print` are `virtual` in class `CommissionEmployee`, class `BasePlusCommissionEmployee`'s `earnings` and `print` functions override class `CommissionEmployee`'s. Now, if we aim a base-class `CommissionEmployee` pointer at a derived-class `BasePlusCommissionEmployee` object, and the program uses that pointer to call either function `earnings` or `print`, the `BasePlusCommissionEmployee` object's corresponding function will be invoked. There were no changes to the member-function implementations of classes `CommissionEmployee` and `BasePlusCommissionEmployee`, so we reuse the versions of Fig. 13.2 and Fig. 13.4.

**Figure 13.8. `CommissionEmployee` class header file declares `earnings` and `print` functions as `virtual`.**

(This item is displayed on page 703 in the print version)

```

1 // Fig. 13.8: CommissionEmployee.h
2 // CommissionEmployee class definition represents a commission employee.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // C++ standard string class
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12 CommissionEmployee(const string &, const string &, const string &,
13 double = 0.0, double = 0.0);
14
15 void setFirstName(const string &); // set first name
16 string getFirstName() const; // return first name
17
18 void setLastName(const string &); // set last name
19 string getLastname() const; // return last name
20
21 void setSocialSecurityNumber(const string &); // set SSN

```

```

22 string getSocialSecurityNumber() const; // return SSN
23
24 void setGrossSales(double); // set gross sales amount
25 double getGrossSales() const; // return gross sales amount
26
27 void setCommissionRate(double); // set commission rate
28 double getCommissionRate() const; // return commission rate
29
30 virtual double earnings() const; // calculate earnings
31 virtual void print() const; // print CommissionEmployee object
32 private:
33 string firstName;
34 string lastName;
35 string socialSecurityNumber;
36 double grossSales; // gross weekly sales
37 double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif

```

**Figure 13.9. BasePlusCommissionEmployee class header file declares earnings and print functions as virtual.**

(This item is displayed on pages 703 - 704 in the print version)

```

1 // Fig. 13.9: BasePlusCommissionEmployee.h
2 // BasePlusCommissionEmployee class derived from class
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // C++ standard string class
8 using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15 BasePlusCommissionEmployee(const string &, const string &,
16 const string &, double = 0.0, double = 0.0, double = 0.0);
17
18 void setBaseSalary(double); // set base salary
19 double getBaseSalary() const; // return base salary
20
21 virtual double earnings() const; // calculate earnings

```

```

22 virtual void print() const; // print BasePlusCommissionEmployee object
23 private:
24 double baseSalary; // base salary
25 };// end class BasePlusCommissionEmployee
26
27 #endif

```

We modified Fig. 13.5 to create the program of Fig. 13.10. Lines 4657 demonstrate again that a CommissionEmployee pointer aimed at a CommissionEmployee object can be used to invoke CommissionEmployee functionality, and a BasePlusCommissionEmployee pointer aimed at a BasePlusCommissionEmployee object can be used to invoke BasePlusCommissionEmployee functionality. Line 60 aims base-class pointer commissionEmployeePtr at derived-class object basePlusCommissionEmployee. Note that when line 67 invokes member function print off the base-class pointer, the derived-class BasePlusCommissionEmployee's print member function is invoked, so line 67 outputs different text than line 59 does in Fig. 13.5 (when member function print was not declared virtual). We see that declaring a member function virtual causes the program to dynamically determine which function to invoke based on the type of object to which the handle points, rather than on the type of the handle. The decision about which function to call is an example of polymorphism. Note again that when commissionEmployeePtr points to a CommissionEmployee object (line 46), class CommissionEmployee's print function is invoked, and when commissionEmployeePtr points to a BasePlusCommissionEmployee object, class BasePlusCommissionEmployee's print function is invoked. Thus, the same messageprint, in this case sent (off a base-class pointer) to a variety of objects related by inheritance to that base class, takes on many forms—this is polymorphic behavior.

---

[Page 704]

### Figure 13.10. Demonstrating polymorphism by invoking a derived-class virtual function via a base-class pointer to a derived-class object.

(This item is displayed on pages 704 - 706 in the print version)

```

1 // Fig. 13.10: fig13_10.cpp
2 // Introducing polymorphism, virtual functions and dynamic binding.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // include class definitions
12 #include "CommissionEmployee.h"
13 #include "BasePlusCommissionEmployee.h"

```

```

14
15 int main()
16 {
17 // create base-class object
18 CommissionEmployee commissionEmployee(
19 "Sue", "Jones", "222-22-2222", 10000, .06);
20
21 // create base-class pointer
22 CommissionEmployee *commissionEmployeePtr = 0;
23
24 // create derived-class object
25 BasePlusCommissionEmployee basePlusCommissionEmployee(
26 "Bob", "Lewis", "333-33-3333", 5000, .04, 300);
27
28 // create derived-class pointer
29 BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
30
31 // set floating-point output formatting
32 cout << fixed << setprecision(2);
33
34 // output objects using static binding
35 cout << "Invoking print function on base-class and derived-class "
36 << "\nobjects with static binding\n\n";
37 commissionEmployee.print(); // static binding
38 cout << "\n\n";
39 basePlusCommissionEmployee.print(); // static binding
40
41 // output objects using dynamic binding
42 cout << "\n\n\nInvoking print function on base-class and "
43 << "derived-class \nobjects with dynamic binding";
44
45 // aim base-class pointer at base-class object and print
46 commissionEmployeePtr = &commissionEmployee;
47 cout << "\n\nCalling virtual function print with base-class pointer"
48 << "\nto base-class object invokes base-class "
49 << "print function:\n\n";
50 commissionEmployeePtr->print(); // invokes base-class print
51
52 // aim derived-class pointer at derived-class object and print
53 basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
54 cout << "\n\nCalling virtual function print with derived-class "
55 << "pointer\nto derived-class object invokes derived-class "
56 << "print function:\n\n";
57 basePlusCommissionEmployeePtr->print(); // invokes derived-class print
58
59 // aim base-class pointer at derived-class object and print
60 commissionEmployeePtr = &basePlusCommissionEmployee;
61 cout << "\n\nCalling virtual function print with base-class pointer"
62 << "\nto derived-class object invokes derived-class "
63 << "print function:\n\n";

```

```

64 // polymorphism; invokes BasePlusCommissionEmployee's print;
65 // base-class pointer to derived-class object
66 commissionEmployeePtr->print();
67 cout << endl;
68 return 0;
70 } // end main

```

Invoking print function on base-class and derived-class objects with static binding

```

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

```

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

```

Invoking print function on base-class and derived-class objects with dynamic binding

Calling virtual function print with base-class pointer to base-class object invokes base-class print function:

```

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

```

Calling virtual function print with derived-class pointer to derived-class object invokes derived-class print function:

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

```

Calling virtual function print with base-class pointer to derived-class object invokes derived-class print function:

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00

```

```
commission rate: 0.04
base salary: 300.00
```

[Page 707]

### 13.3.5. Summary of the Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers

Now that you have seen a complete application that processes diverse objects polymorphically, we summarize what you can and cannot do with base-class and derived-class objects and pointers. Although a derived-class object also is a base-class object, the two objects are nevertheless different. As discussed previously, derived-class objects can be treated as if they are base-class objects. This is a logical relationship, because the derived class contains all the members of the base class. However, base-class objects cannot be treated as if they are derived-class objects—the derived class can have additional derived-class-only members. For this reason, aiming a derived-class pointer at a base-class object is not allowed without an explicit cast; such an assignment would leave the derived-class-only members undefined on the base-class object. The cast relieves the compiler of the responsibility of issuing an error message. In a sense, by using the cast you are saying, "I know that what I'm doing is dangerous and I take full responsibility for my actions."

In the current section and in [Chapter 12](#), we have discussed four ways to aim base-class pointers and derived-class pointers at base-class objects and derived-class objects:

**1.**

Aiming a base-class pointer at a base-class object is straightforward; calls made off the base-class pointer simply invoke base-class functionality.

**2.**

Aiming a derived-class pointer at a derived-class object is straightforward; calls made off the derived-class pointer simply invoke derived-class functionality.

**3.**

Aiming a base-class pointer at a derived-class object is safe, because the derived-class object is an object of its base class. However, this pointer can be used to invoke only base-class member functions. If the programmer attempts to refer to a derived-class-only member through the base-class pointer, the compiler reports an error. To avoid this error, the programmer must cast the base-class pointer to a derived-class pointer. The derived-class pointer can then be used to invoke the derived-class object's complete functionality. However, this technique called **downcasting** is a potentially dangerous operation. [Section 13.8](#) demonstrates how to safely use downcasting.

4.

Aiming a derived-class pointer at a base-class object generates a compilation error. The is-a relationship applies only from a derived class to its direct and indirect base classes, and not vice versa. A base-class object does not contain the derived-class-only members that can be invoked off a derived-class pointer.

Common Programming Error 13.1



After aiming a base-class pointer at a derived-class object, attempting to reference derived-class-only members with the base-class pointer is a compilation error.

Common Programming Error 13.2



Treating a base-class object as a derived-class object can cause errors.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 708]

However, using `switch` logic exposes programs to a variety of potential problems. For example, the programmer might forget to include a type test when one is warranted, or might forget to test all possible cases in a `switch` statement. When modifying a `switch`-based system by adding new types, the programmer might forget to insert the new cases in all relevant `switch` statements. Every addition or deletion of a class requires the modification of every `switch` statement in the system; tracking these statements down can be time consuming and error prone.

#### Software Engineering Observation 13.6



Polymorphic programming can eliminate the need for unnecessary `switch` logic. By using the C++ polymorphism mechanism to perform the equivalent logic, programmers can avoid the kinds of errors typically associated with `switch` logic.

#### Software Engineering Observation 13.7



An interesting consequence of using polymorphism is that programs take on a simplified appearance. They contain less branching logic and more simple, sequential code. This simplification facilitates testing, debugging and program maintenance.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 709]

A class is made abstract by declaring one or more of its `virtual` functions to be "pure." A **pure virtual function** is specified by placing "`= 0`" in its declaration, as in

```
virtual void draw() const = 0; // pure virtual function
```

The "`=0`" is known as a **pure specifier**. Pure virtual functions do not provide implementations. Every concrete derived class must override all base-class pure virtual functions with concrete implementations of those functions. The difference between a `virtual` function and a `pure virtual` function is that a `virtual` function has an implementation and gives the derived class the option of overriding the function; by contrast, a `pure virtual` function does not provide an implementation and requires the derived class to override the function (for that derived class to be concrete; otherwise the derived class remains abstract).

Pure virtual functions are used when it does not make sense for the base class to have an implementation of a function, but the programmer wants all concrete derived classes to implement the function. Returning to our earlier example of space objects, it does not make sense for the base class `SpaceObject` to have an implementation for function `draw` (as there is no way to draw a generic space object without having more information about what type of space object is being drawn). An example of a function that would be defined as `virtual` (and not `pure virtual`) would be one that returns a name for the object. We can name a generic `SpaceObject` (for instance, as "`space object`"), so a default implementation for this function can be provided, and the function does not need to be `pure virtual`. The function is still declared `virtual`, however, because it is expected that derived classes will override this function to provide more specific names for the derived-class objects.

## Software Engineering Observation 13.8



An abstract class defines a common public interface for the various classes in a class hierarchy. An abstract class contains one or more `pure virtual` functions that concrete derived classes must override.

## Common Programming Error 13.3



Attempting to instantiate an object of an abstract class causes a compilation error.

### Common Programming Error 13.4



Failure to override a pure `virtual` function in a derived class, then attempting to instantiate objects of that class, is a compilation error.

### Software Engineering Observation 13.9



An abstract class has at least one pure `virtual` function. An abstract class also can have data members and concrete functions (including constructors and destructors), which are subject to the normal rules of inheritance by derived classes.

Although we cannot instantiate objects of an abstract base class, we can use the abstract base class to declare pointers and references that can refer to objects of any concrete classes derived from the abstract class. Programs typically use such pointers and references to manipulate derived-class objects polymorphically.

Let us consider another application of polymorphism. A screen manager needs to display a variety of objects, including new types of objects that the programmer will add to the system after writing the screen manager. The system might need to display various shapes, such as `Circles`, `triangles` or `Rectangles`, which are derived from abstract base class `Shape`. The screen manager uses `Shape` pointers to manage the objects that are displayed. To draw any object (regardless of the level at which that object's class appears in the inheritance hierarchy), the screen manager uses a base-class pointer to the object to invoke the object's `draw` function, which is a pure `virtual` function in base class `Shape`; therefore, each concrete derived class must implement function `draw`. Each `Shape` object in the inheritance hierarchy knows how to draw itself. The screen manager does not have to worry about the type of each object or whether the screen manager has ever encountered objects of that type.

---

[Page 710]

Polymorphism is particularly effective for implementing layered software systems. In operating systems, for example, each type of physical device could operate quite differently from the others. Even so, commands to read or write data from and to devices may have a certain uniformity. The write message sent to a device-driver object needs to be interpreted specifically in the context of that device driver and how that device driver manipulates devices of a specific type. However, the write call itself really is no

different from the write to any other device in the system place some number of bytes from memory onto that device. An object-oriented operating system might use an abstract base class to provide an interface appropriate for all device drivers. Then, through inheritance from that abstract base class, derived classes are formed that all operate similarly. The capabilities (i.e., the public functions) offered by the device drivers are provided as pure virtual functions in the abstract base class. The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of device drivers. This architecture also allows new devices to be added to a system easily, even after the operating system has been defined. The user can just plug in the device and install its new device driver. The operating system "talks" to this new device through its device driver, which has the same public member functions as all other device drivers those defined in the abstract base device driver class.

It is common in object-oriented programming to define an **iterator class** that can traverse all the objects in a container (such as an array). For example, a program can print a list of objects in a vector by creating an iterator object, then using the iterator to obtain the next element of the list each time the iterator is called. Iterators often are used in polymorphic programming to traverse an array or a linked list of pointers to objects from various levels of a hierarchy. The pointers in such a list are all base-class pointers. (Chapter 23, Standard Template Library (STL), presents a thorough treatment of iterators.) A list of pointers to objects of base class TwoDimensionalShape could contain pointers to objects of classes Square, Circle, triangle and so on. Using polymorphism to send a draw message, off a TwoDimensionalShape \* pointer, to each object in the list would draw each object correctly on the screen.

 PREVNEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

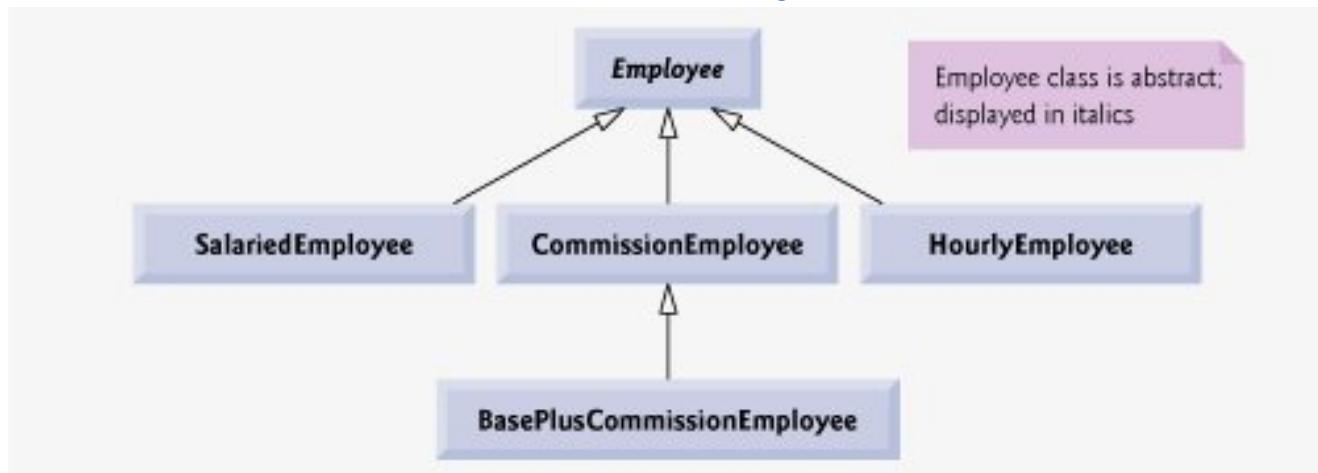
[Page 711]

A company pays its employees weekly. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales and base-salary-plus-commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward base-salary-plus-commission employees by adding 10 percent to their base salaries. The company wants to implement a C++ program that performs its payroll calculations polymorphically.

We use abstract class `Employee` to represent the general concept of an employee. The classes that derive directly from `Employee` are `SalariedEmployee`, `CommissionEmployee` and `HourlyEmployee`. Class `BasePlusCommissionEmployee` derived from `CommissionEmployee` represents the last employee type. The UML class diagram in Fig. 13.11 shows the inheritance hierarchy for our polymorphic employee payroll application. Note that abstract class name `Employee` is italicized, as per the convention of the UML.

**Figure 13.11. Employee hierarchy UML class diagram.**

[View full size image]



Abstract base class `Employee` declares the "interface" to the hierarchy that is, the set of member functions that a program can invoke on all `Employee` objects. Each employee, regardless of the way his or her earnings are calculated, has a first name, a last name and a social security number, so private data members `firstName`, `lastName` and `socialSecurityNumber` appear in abstract base class `Employee`.

Software Engineering Observation 13.10



A derived class can inherit interface or implementation from a base class. Hierarchies designed for **implementation inheritance** tend to have their functionality high in the hierarchy each new derived class inherits one or more member functions that were defined in a base class, and the derived class uses the base-class definitions. Hierarchies designed for **interface inheritance** tend to have their functionality lower in the hierarchy a base class specifies one or more functions that should be defined for each class in the hierarchy (i.e., they have the same prototype), but the individual derived classes provide their own implementations of the function(s).

The following sections implement the `Employee` class hierarchy. The first five each implement one of the abstract or concrete classes. The last section implements a test program that builds objects of all these classes and processes the objects polymorphically.

[Page 712]

### 13.6.1. Creating Abstract Base Class `Employee`

Class `Employee` (Figs. 13.1313.14, discussed in further detail shortly) provides functions `earnings` and `print`, in addition to various get and set functions that manipulate `Employee`'s data members. An `earnings` function certainly applies generically to all employees, but each `earnings` calculation depends on the employee's class. So we declare `earnings` as pure `virtual` in base class `Employee` because a default implementation does not make sense for that function there is not enough information to determine what amount `earnings` should return. Each derived class overrides `earnings` with an appropriate implementation. To calculate an employee's `earnings`, the program assigns the address of an employee's object to a base class `Employee` pointer, then invokes the `earnings` function on that object. We maintain a vector of `Employee` pointers, each of which points to an `Employee` object (of course, there cannot be `Employee` objects, because `Employee` is an abstract class because of inheritance, however, all objects of all derived classes of `Employee` may nevertheless be thought of as `Employee` objects). The program iterates through the vector and calls function `earnings` for each `Employee` object. C++ processes these function calls polymorphically. Including `earnings` as a pure `virtual` function in `Employee` forces every direct derived class of `Employee` that wishes to be a concrete class to override `earnings`. This enables the designer of the class hierarchy to demand that each derived class provide an appropriate pay calculation, if indeed that derived class is to be concrete.

Function `print` in class `Employee` displays the first name, last name and social security number of the employee. As we will see, each derived class of `Employee` overrides function `print` to output the employee's type (e.g., "salaried employee:") followed by the rest of the employee's information.

The diagram in Fig. 13.12 shows each of the five classes in the hierarchy down the left side and functions `earnings` and `print` across the top. For each class, the diagram shows the desired results of each function. Note that class `Employee` specifies "`= 0`" for function `earnings` to indicate that this is a pure `virtual` function. Each derived class overrides this function to provide an appropriate implementation. We do not list base class `Employee`'s get and set functions because they are not overridden in any of the

derived classes each of these functions is inherited and used "as is" by each of the derived classes.

**Figure 13.12. Polymorphic interface for the Employee hierarchy classes.**

(This item is displayed on page 713 in the print version)

[View full size image]

|                              | earnings                                                                                                                                            | print                                                                                                                                                                                          |
|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Employee                     | $= 0$                                                                                                                                               | <i>firstName lastName<br/>social security number: SSN</i>                                                                                                                                      |
| Salaried-Employee            | <i>weeklySalary</i>                                                                                                                                 | <i>salaried employee: firstName lastName<br/>social security number: SSN<br/>weekly salary: weeklySalary</i>                                                                                   |
| Hourly-Employee              | $\text{if } hours \leq 40$<br>$\quad wage * hours$<br>$\text{if } hours > 40$<br>$\quad ( 40 * wage ) +$<br>$\quad ( ( hours - 40 ) * wage * 1.5 )$ | <i>hourly employee: firstName lastName<br/>social security number: SSN<br/>hourly wage: wage; hours worked: hours</i>                                                                          |
| Commission-Employee          | <i>commissionRate * grossSales</i>                                                                                                                  | <i>commission employee: firstName lastName<br/>social security number: SSN<br/>gross sales: grossSales;<br/>commission rate: commissionRate</i>                                                |
| BasePlus-Commission-Employee | <i>baseSalary + ( commissionRate * grossSales )</i>                                                                                                 | <i>base salaried commission employee:<br/>firstName lastName<br/>social security number: SSN<br/>gross sales: grossSales;<br/>commission rate: commissionRate;<br/>base salary: baseSalary</i> |

Let us consider class Employee's header file (Fig. 13.13). The public member functions include a constructor that takes the first name, last name and social security number as arguments (line 12); set functions that set the first name, last name and social security number (lines 14, 17 and 20, respectively); get functions that return the first name, last name and social security number (lines 15, 18 and 21, respectively); pure virtual function earnings (line 24) and virtual function print (line 25).

**Figure 13.13. Employee class header file.**

(This item is displayed on pages 713 - 714 in the print version)

```

1 // Fig. 13.13: Employee.h
2 // Employee abstract base class.
3 #ifndef EMPLOYEE_H
4 #define EMPLOYEE_H
5
6 #include <string> // C++ standard string class
7 using std::string;
8
9 class Employee
10 {
11 public:
12 Employee(const string &, const string &, const string &);
13
14 void setFirstName(const string &); // set first name
15 string getFirstName() const; // return first name
16
17 void setLastName(const string &); // set last name
18 string getLastname() const; // return last name
19
20 void setSocialSecurityNumber(const string &); // set SSN
21 string getSocialSecurityNumber() const; // return SSN
22
23 // pure virtual function makes Employee abstract base class
24 virtual double earnings() const = 0; // pure virtual
25 virtual void print() const; // virtual
26 private:
27 string firstName;
28 string lastName;
29 string socialSecurityNumber;
30 }; // end class Employee
31
32 #endif // EMPLOYEE_H

```

Recall that we declared `earnings` as a pure virtual function because we first must know the specific `Employee` type to determine the appropriate `earnings` calculations. Declaring this function as pure virtual indicates that each concrete derived class must provide an appropriate `earnings` implementation and that a program can use base-class `Employee` pointers to invoke function `earnings` polymorphically for any type of `Employee`.

Figure 13.14 contains the member-function implementations for class `Employee`. No implementation is provided for virtual function `earnings`. Note that the `Employee` constructor (lines 1015) does not validate the social security number. Normally, such validation should be provided. An exercise in Chapter 12 asks you to validate a social security number to ensure that it is in the form `###-##-####`, where each `#` represents a digit.

**Figure 13.14. Employee class implementation file.**

(This item is displayed on pages 714 - 715 in the print version)

```
1 // Fig. 13.14: Employee.cpp
2 // Abstract-base-class Employee member-function definitions.
3 // Note: No definitions are given for pure virtual functions.
4 #include <iostream>
5 using std::cout;
6
7 #include "Employee.h" // Employee class definition
8
9 // constructor
10 Employee::Employee(const string &first, const string &last,
11 const string &ssn)
12 : firstName(first), lastName(last), socialSecurityNumber(ssn)
13 {
14 // empty body
15 } // end Employee constructor
16
17 // set first name
18 void Employee::setFirstName(const string &first)
19 {
20 firstName = first;
21 } // end function setFirstName
22
23 // return first name
24 string Employee::getFirstName() const
25 {
26 return firstName;
27 } // end function getFirstName
28
29 // set last name
30 void Employee::setLastName(const string &last)
31 {
32 lastName = last;
33 } // end function setLastName
34
35 // return last name
36 string Employee::getLastName() const
37 {
38 return lastName;
39 } // end function getLastName
40
41 // set social security number
```

```

42 void Employee::setSocialSecurityNumber(const string &ssn)
43 {
44 socialSecurityNumber = ssn; // should validate
45 } // end function setSocialSecurityNumber
46
47 // return social security number
48 string Employee::getSocialSecurityNumber() const
49 {
50 return socialSecurityNumber;
51 } // end function getSocialSecurityNumber
52
53 // print Employee's information (virtual, but not pure virtual)
54 void Employee::print() const
55 {
56 cout << getFirstName() << ' ' << getLastName()
57 << "\nsocial security number: " << getSocialSecurityNumber();
58 } // end function print

```

[Page 715]

Note that `virtual` function `print` (Fig. 13.14, lines 5458) provides an implementation that will be overridden in each of the derived classes. Each of these functions will, however, use the abstract class's version of `print` to print information common to all classes in the `Employee` hierarchy.

### 13.6.2. Creating Concrete Derived Class `SalariedEmployee`

Class `SalariedEmployee` (Figs. 13.1513.16) derives from class `Employee` (line 8 of Fig. 13.15). The public member functions include a constructor that takes a first name, a last name, a social security number and a weekly salary as arguments (lines 1112); a set function to assign a new nonnegative value to data member `weeklySalary` (lines 14); a get function to return `weeklySalary`'s value (line 15); a virtual function `earnings` that calculates a `SalariedEmployee`'s earnings (line 18) and a virtual function `print` that outputs the employee's type, namely, "salaried employee:" followed by employee-specific information produced by base class `Employee`'s `print` function and `SalariedEmployee`'s `getWeeklySalary` function (line 19).

**Figure 13.15. `SalariedEmployee` class header file.**

(This item is displayed on page 716 in the print version)

```

1 // Fig. 13.15: SalariedEmployee.h
2 // SalariedEmployee class derived from Employee.
3 #ifndef SALARIED_H
4 #define SALARIED_H
5
6 #include "Employee.h" // Employee class definition
7
8 class SalariedEmployee : public Employee
9 {
10 public:
11 SalariedEmployee(const string &, const string &,
12 const string &, double = 0.0);
13
14 void setWeeklySalary(double); // set weekly salary
15 double getWeeklySalary() const; // return weekly salary
16
17 // keyword virtual signals intent to override
18 virtual double earnings() const; // calculate earnings
19 virtual void print() const; // print SalariedEmployee object
20 private:
21 double weeklySalary; // salary per week
22 }; // end class SalariedEmployee
23
24 #endif // SALARIED_H

```

Figure 13.16 contains the member-function implementations for `SalariedEmployee`. The class's constructor passes the first name, last name and social security number to the `Employee` constructor (line 11) to initialize the private data members that are inherited from the base class, but not accessible in the derived class. Function `earnings` (line 3033) overrides pure virtual function `earnings` in `Employee` to provide a concrete implementation that returns the `SalariedEmployee`'s weekly salary. If we do not implement `earnings`, class `SalariedEmployee` would be an abstract class, and any attempt to instantiate an object of the class would result in a compilation error (and, of course, we want `SalariedEmployee` here to be a concrete class). Note that in class `SalariedEmployee`'s header file, we declared member functions `earnings` and `print` as `virtual` (lines 1819 of Fig. 13.15) actually, placing the `virtual` keyword before these member functions is redundant. We defined them as `virtual` in base class `Employee`, so they remain `virtual` functions throughout the class hierarchy. Recall from Good Programming Practice 13.1 that explicitly declaring such functions `virtual` at every level of the hierarchy can promote program clarity.

**Figure 13.16. `SalariedEmployee` class implementation file.**

(This item is displayed on pages 716 - 717 in the print version)

```

1 // Fig. 13.16: SalariedEmployee.cpp
2 // SalariedEmployee class member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 #include "SalariedEmployee.h" // SalariedEmployee class definition
7
8 // constructor
9 SalariedEmployee::SalariedEmployee(const string &first,
10 const string &last, const string &ssn, double salary)
11 : Employee(first, last, ssn)
12 {
13 setWeeklySalary(salary);
14 } // end SalariedEmployee constructor
15
16 // set salary
17 void SalariedEmployee::setWeeklySalary(double salary)
18 {
19 weeklySalary = (salary < 0.0) ? 0.0 : salary;
20 } // end function setWeeklySalary
21
22 // return salary
23 double SalariedEmployee::getWeeklySalary() const
24 {
25 return weeklySalary;
26 } // end function getWeeklySalary
27
28 // calculate earnings;
29 // override pure virtual function earnings in Employee
30 double SalariedEmployee::earnings() const
31 {
32 return getWeeklySalary();
33 } // end function earnings
34
35 // print SalariedEmployee's information
36 void SalariedEmployee::print() const
37 {
38 cout << "salaried employee: ";
39 Employee::print(); // reuse abstract base-class print function
40 cout << "\nweekly salary: " << getWeeklySalary();
41 } // end function print

```

Function `print` of class `SalariedEmployee` (lines 3641 of Fig. 13.16) overrides `Employee` function

`print`. If class `SalariedEmployee` did not override `print`, `SalariedEmployee` would inherit the `Employee` version of `print`. In that case, `SalariedEmployee`'s `print` function would simply return the employee's full name and social security number, which does not adequately represent a `SalariedEmployee`. To print a `SalariedEmployee`'s complete information, the derived class's `print` function outputs "salaried employee:" followed by the base-class `Employee`-specific information (i.e., first name, last name and social security number) printed by invoking the base class's `print` using the scope resolution operator (line 39)this is a nice example of code reuse. The output produced by `SalariedEmployee`'s `print` function contains the employee's weekly salary obtained by invoking the class's `getWeeklySalary` function.

### 13.6.3. Creating Concrete Derived Class `HourlyEmployee`

Class `HourlyEmployee` (Figs. 13.1713.18) also derives from class `Employee` (line 8 of Fig. 13.17). The public member functions include a constructor (lines 1112) that takes as arguments a first name, a last name, a social security number, an hourly wage and the number of hours worked; set functions that assign new values to data members `wage` and `hours`, respectively (lines 14 and 17); get functions to return the values of `wage` and `hours`, respectively (lines 15 and 18); a virtual function `earnings` that calculates an `HourlyEmployee`'s earnings (line 21) and a virtual function `print` that outputs the employee's type, namely, "hourly employee:" and employee-specific information (line 22).

---

[Page 718]

**Figure 13.17. `HourlyEmployee` class header file.**

```

1 // Fig. 13.17: HourlyEmployee.h
2 // HourlyEmployee class definition.
3 #ifndef HOURLY_H
4 #define HOURLY_H
5
6 #include "Employee.h" // Employee class definition
7
8 class HourlyEmployee : public Employee
9 {
10 public:
11 HourlyEmployee(const string &, const string &,
12 const string &, double = 0.0, double = 0.0);
13
14 void setWage(double); // set hourly wage
15 double getWage() const; // return hourly wage
16
17 void setHours(double); // set hours worked
18 double getHours() const; // return hours worked
19
20 // keyword virtual signals intent to override
21 virtual double earnings() const; // calculate earnings

```

```
22 virtual void print() const; // print HourlyEmployee object
23 private:
24 double wage; // wage per hour
25 double hours; // hours worked for week
26 }; // end class HourlyEmployee
27
28 #endif // HOURLY_H
```

---

[Page 719]

Figure 13.18 contains the member-function implementations for class HourlyEmployee. Lines 1821 and 3034 define set functions that assign new values to data members `wage` and `hours`, respectively. Function `setWage` (lines 1821) ensures that `wage` is nonnegative, and function `setHours` (lines 3034) ensures that data member `hours` is between 0 and 168 (the total number of hours in a week). Class HourlyEmployee's get functions are implemented in lines 2427 and 3740. We do not declare these functions `virtual`, so classes derived from class HourlyEmployee cannot override them (although derived classes certainly can redefine them). Note that the HourlyEmployee constructor, like the SalariedEmployee constructor, passes the first name, last name and social security number to the base class Employee constructor (line 11) to initialize the inherited `private` data members declared in the base class. In addition, HourlyEmployee's `print` function calls base-class function `print` (line 56) to output the Employee-specific information (i.e., first name, last name and social security number) this is another nice example of code reuse.

---

[Page 720]

**Figure 13.18. HourlyEmployee class implementation file.**

(This item is displayed on pages 718 - 719 in the print version)

```

1 // Fig. 13.18: HourlyEmployee.cpp
2 // HourlyEmployee class member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 #include "HourlyEmployee.h" // HourlyEmployee class definition
7
8 // constructor
9 HourlyEmployee::HourlyEmployee(const string &first, const string &last,
10 const string &ssn, double hourlyWage, double hoursWorked)
11 : Employee(first, last, ssn)
12 {
13 setWage(hourlyWage); // validate hourly wage
14 setHours(hoursWorked); // validate hours worked
15 } // end HourlyEmployee constructor
16
17 // set wage
18 void HourlyEmployee::setWage(double hourlyWage)
19 {
20 wage = (hourlyWage < 0.0 ? 0.0 : hourlyWage);
21 } // end function setWage
22
23 // return wage
24 double HourlyEmployee::getWage() const
25 {
26 return wage;
27 } // end function getWage
28
29 // set hours worked
30 void HourlyEmployee::setHours(double hoursWorked)
31 {
32 hours = (((hoursWorked >= 0.0) && (hoursWorked <= 168.0)) ?
33 hoursWorked : 0.0);
34 } // end function setHours
35
36 // return hours worked
37 double HourlyEmployee::getHours() const
38 {
39 return hours;
40 } // end function getHours
41
42 // calculate earnings;
43 // override pure virtual function earnings in Employee
44 double HourlyEmployee::earnings() const
45 {
46 if (getHours() <= 40) // no overtime
47 return getWage() * getHours();
48 else
49 return 40 * getWage() + ((getHours() - 40) * getWage() * 1.5);

```

```
50 } // end function earnings
51
52 // print HourlyEmployee's information
53 void HourlyEmployee::print() const
54 {
55 cout << "hourly employee: ";
56 Employee::print(); // code reuse
57 cout << "\nhourly wage: " << getWage() <<
58 "; hours worked: " << getHours();
59 } // end function print
```

### 13.6.4. Creating Concrete Derived Class CommissionEmployee

Class `CommissionEmployee` (Figs. 13.1913.20) derives from class `Employee` (line 8 of Fig. 13.19). The member-function implementations (Fig. 13.20) include a constructor (lines 915) that takes a first name, a last name, a social security number, a sales amount and a commission rate; set functions (lines 1821 and 3033) to assign new values to data members `commissionRate` and `grossSales`, respectively; get functions (lines 2427 and 3639) that retrieve the values of these data members; function `earnings` (lines 4346) to calculate a `CommissionEmployee`'s earnings; and function `print` (lines 4955), which outputs the employee's type, namely, "commission employee:" and employee-specific information. The `CommissionEmployee`'s constructor also passes the first name, last name and social security number to the `Employee` constructor (line 11) to initialize `Employee`'s private data members. Function `print` calls base-class function `print` (line 52) to display the `Employee`-specific information (i.e., first name, last name and social security number).

---

[Page 722]

**Figure 13.19. CommissionEmployee class header file.**

(This item is displayed on page 720 in the print version)

```
1 // Fig. 13.19: CommissionEmployee.h
2 // CommissionEmployee class derived from Employee.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include "Employee.h" // Employee class definition
7
8 class CommissionEmployee : public Employee
9 {
10 public:
11 CommissionEmployee(const string &, const string &,
12 const string &, double = 0.0, double = 0.0);
13
14 void setCommissionRate(double); // set commission rate
15 double getCommissionRate() const; // return commission rate
16
17 void setGrossSales(double); // set gross sales amount
18 double getGrossSales() const; // return gross sales amount
19
20 // keyword virtual signals intent to override
21 virtual double earnings() const; // calculate earnings
22 virtual void print() const; // print CommissionEmployee object
23 private:
24 double grossSales; // gross weekly sales
25 double commissionRate; // commission percentage
26 }; // end class CommissionEmployee
27
28 #endif // COMMISSION_H
```

**Figure 13.20. CommissionEmployee class implementation file.**

(This item is displayed on pages 720 - 721 in the print version)

```

1 // Fig. 13.20: CommissionEmployee.cpp
2 // CommissionEmployee class member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8 // constructor
9 CommissionEmployee::CommissionEmployee(const string &first,
10 const string &last, const string &ssn, double sales, double rate)
11 : Employee(first, last, ssn)
12 {
13 setGrossSales(sales);
14 setCommissionRate(rate);
15 } // end CommissionEmployee constructor
16
17 // set commission rate
18 void CommissionEmployee::setCommissionRate(double rate)
19 {
20 commissionRate = ((rate > 0.0 && rate < 1.0) ? rate : 0.0);
21 } // end function setCommissionRate
22
23 // return commission rate
24 double CommissionEmployee::getCommissionRate() const
25 {
26 return commissionRate;
27 } // end function getCommissionRate
28
29 // set gross sales amount
30 void CommissionEmployee::setGrossSales(double sales)
31 {
32 grossSales = ((sales < 0.0) ? 0.0 : sales);
33 } // end function setGrossSales
34
35 // return gross sales amount
36 double CommissionEmployee::getGrossSales() const
37 {
38 return grossSales;
39 } // end function getGrossSales
40
41 // calculate earnings;
42 // override pure virtual function earnings in Employee
43 double CommissionEmployee::earnings() const
44 {
45 return getCommissionRate() * getGrossSales();
46 } // end function earnings
47
48 // print CommissionEmployee's information
49 void CommissionEmployee::print() const

```

```

50 {
51 cout << "commission employee: ";
52 Employee::print(); // code reuse
53 cout << "\ngross sales: " << getGrossSales()
54 << "; commission rate: " << getCommissionRate();
55 } // end function print

```

### 13.6.5. Creating Indirect Concrete Derived Class BasePlusCommissionEmployee

Class `BasePlusCommissionEmployee` (Figs. 13.2113.22) directly inherits from class `CommissionEmployee` (line 8 of Fig. 13.21) and therefore is an indirect derived class of class `Employee`. Class `BasePlusCommissionEmployee`'s member-function implementations include a constructor (lines 1016 of Fig. 13.22) that takes as arguments a first name, a last name, a social security number, a sales amount, a commission rate and a base salary. It then passes the first name, last name, social security number, sales amount and commission rate to the `CommissionEmployee` constructor (line 13) to initialize the inherited members. `BasePlusCommissionEmployee` also contains a set function (lines 1922) to assign a new value to data member `baseSalary` and a get function (lines 2528) to return `baseSalary`'s value. Function `earnings` (lines 3235) calculates a `BasePlusCommissionEmployee`'s earnings. Note that line 34 in function `earnings` calls base-class `CommissionEmployee`'s `earnings` function to calculate the commission-based portion of the employee's earnings. This is a nice example of code reuse. `BasePlusCommissionEmployee`'s `print` function (lines 3843) outputs "base-salaried", followed by the output of base-class `CommissionEmployee`'s `print` function (another example of code reuse), then the base salary. The resulting output begins with "base-salaried commission employee:" followed by the rest of the `BasePlusCommissionEmployee`'s information. Recall that `CommissionEmployee`'s `print` displays the employee's first name, last name and social security number by invoking the `print` function of its base class (i.e., `Employee`) yet another example of code reuse. Note that `BasePlusCommissionEmployee`'s `print` initiates a chain of functions calls that spans all three levels of the `Employee` hierarchy.

---

[Page 723]

**Figure 13.21. BasePlusCommissionEmployee class header file.**

(This item is displayed on page 722 in the print version)

```

1 // Fig. 13.21: BasePlusCommissionEmployee.h
2 // BasePlusCommissionEmployee class derived from Employee.
3 #ifndef BASEPLUS_H
4 #define BASEPLUS_H
5
6 #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8 class BasePlusCommissionEmployee : public CommissionEmployee
9 {
10 public:
11 BasePlusCommissionEmployee(const string &, const string &,
12 const string &, double = 0.0, double = 0.0, double = 0.0);
13
14 void setBaseSalary(double); // set base salary
15 double getBaseSalary() const; // return base salary
16
17 // keyword virtual signals intent to override
18 virtual double earnings() const; // calculate earnings
19 virtual void print() const; // print BasePlusCommissionEmployee object
20 private:
21 double baseSalary; // base salary per week
22 }; // end class BasePlusCommissionEmployee
23
24 #endif // BASEPLUS_H

```

**Figure 13.22. BasePlusCommissionEmployee class implementation file.**

```

1 // Fig. 13.22: BasePlusCommissionEmployee.cpp
2 // BasePlusCommissionEmployee member-function definitions.
3 #include <iostream>
4 using std::cout;
5
6 // BasePlusCommissionEmployee class definition
7 #include "BasePlusCommissionEmployee.h"
8
9 // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11 const string &first, const string &last, const string &ssn,
12 double sales, double rate, double salary)
13 : CommissionEmployee(first, last, ssn, sales, rate)
14 {
15 setBaseSalary(salary); // validate and store base salary
16 } // end BasePlusCommissionEmployee constructor
17
18 // set base salary

```

```

19 void BasePlusCommissionEmployee::setBaseSalary(double salary)
20 {
21 baseSalary = ((salary < 0.0) ? 0.0 : salary);
22 } // end function setBaseSalary
23
24 // return base salary
25 double BasePlusCommissionEmployee::getBaseSalary() const
26 {
27 return baseSalary;
28 } // end function getBaseSalary
29
30 // calculate earnings;
31 // override pure virtual function earnings in Employee
32 double BasePlusCommissionEmployee::earnings() const
33 {
34 return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee's information
38 void BasePlusCommissionEmployee::print() const
39 {
40 cout << "base-salaried ";
41 CommissionEmployee::print(); // code reuse
42 cout << "; base salary: " << getBaseSalary();
43 } // end function print

```

[Page 724]

### 13.6.6. Demonstrating Polymorphic Processing

To test our `Employee` hierarchy, the program in Fig. 13.23 creates an object of each of the four concrete classes `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`. The program manipulates these objects, first with static binding, then polymorphically, using a vector of `Employee` pointers. Lines 3138 create objects of each of the four concrete `Employee` derived classes. Lines 4351 output each `Employee`'s information and earnings. Each member-function invocation in lines 4351 is an example of static binding at compile time, because we are using name handles (not pointers or references that could be set at execution time), the compiler can identify each object's type to determine which `print` and `earnings` functions are called.

[Page 727]

**Figure 13.23. Employee class hierarchy driver program.**

(This item is displayed on pages 724 - 727 in the print version)

```

1 // Fig. 13.23: fig13_23.cpp
2 // Processing Employee derived-class objects individually
3 // and polymorphically using dynamic binding.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 #include <vector>
13 using std::vector;
14
15 // include definitions of classes in Employee hierarchy
16 #include "Employee.h"
17 #include "SalariedEmployee.h"
18 #include "HourlyEmployee.h"
19 #include "CommissionEmployee.h"
20 #include "BasePlusCommissionEmployee.h"
21
22 void virtualViaPointer(const Employee * const); // prototype
23 void virtualViaReference(const Employee &); // prototype
24
25 int main()
26 {
27 // set floating-point output formatting
28 cout << fixed << setprecision(2);
29
30 // create derived-class objects
31 SalariedEmployee salariedEmployee(
32 "John", "Smith", "111-11-1111", 800);
33 HourlyEmployee hourlyEmployee(
34 "Karen", "Price", "222-22-2222", 16.75, 40);
35 CommissionEmployee commissionEmployee(
36 "Sue", "Jones", "333-33-3333", 10000, .06);
37 BasePlusCommissionEmployee basePlusCommissionEmployee(
38 "Bob", "Lewis", "444-44-4444", 5000, .04, 300);
39
40 cout << "Employees processed individually using static binding:\n\n";
41
42 // output each Employee's information and earnings using static binding
43 salariedEmployee.print();
44 cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
45 hourlyEmployee.print();
46 cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";

```

```

47 commissionEmployee.print();
48 cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
49 basePlusCommissionEmployee.print();
50 cout << "\nearned $" << basePlusCommissionEmployee.earnings()
51 << "\n\n";
52
53 // create vector of four base-class pointers
54 vector < Employee * > employees(4);
55
56 // initialize vector with Employees
57 employees[0] = &salariedEmployee;
58 employees[1] = &hourlyEmployee;
59 employees[2] = &commissionEmployee;
60 employees[3] = &basePlusCommissionEmployee;
61
62 cout << "Employees processed polymorphically via dynamic binding:\n\n";
63
64 // call virtualViaPointer to print each Employee's information
65 // and earnings using dynamic binding
66 cout << "Virtual function calls made off base-class pointers:\n\n";
67
68 for (size_t i = 0; i < employees.size(); i++)
69 virtualViaPointer(employees[i]);
70
71 // call virtualViaReference to print each Employee's information
72 // and earnings using dynamic binding
73 cout << "Virtual function calls made off base-class references:\n\n";
74
75 for (size_t i = 0; i < employees.size(); i++)
76 virtualViaReference(*employees[i]); // note dereferencing
77
78 return 0;
79 } // end main
80
81 // call Employee virtual functions print and earnings off a
82 // base-class pointer using dynamic binding
83 void virtualViaPointer(const Employee * const baseClassPtr)
84 {
85 baseClassPtr->print();
86 cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
87 } // end function virtualViaPointer
88
89 // call Employee virtual functions print and earnings off a
90 // base-class reference using dynamic binding
91 void virtualViaReference(const Employee &baseClassRef)
92 {
93 baseClassRef.print();
94 cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
95 } // end function virtualViaReference

```

Employees processed individually using static binding:

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00
```

```
hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00
```

```
commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00
```

```
base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

Employees processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00
```

```
hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00
```

```
commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00
```

```
base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

Virtual function calls made off base-class references:

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00
```

```
hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00
```

```
commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00
```

```
base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```

Line 54 allocates vector `employees`, which contains four `Employee` pointers. Line 57 aims `employees[0]` at object `salariedEmployee`. Line 58 aims `employees[1]` at object `hourlyEmployee`. Line 59 aims `employees[2]` at object `commissionEmployee`. Line 60 aims `employee[3]` at object `basePlusCommissionEmployee`. The compiler allows these assignments, because a `SalariedEmployee` is an `Employee`, an `HourlyEmployee` is an `Employee`, a `CommissionEmployee` is an `Employee` and a `BasePlusCommissionEmployee` is an `Employee`. Therefore, we can assign the addresses of `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee` objects to base-class `Employee` pointers (even though `Employee` is an abstract class).

The `for` statement at lines 6869 traverses vector `employees` and invokes function `virtualViaPointer` (lines 8387) for each element in `employees`. Function `virtualViaPointer` receives in parameter `baseClassPtr` (of type `const Employee * const`) the address stored in an `employees` element. Each call to `virtualViaPointer` uses `baseClassPtr` to invoke virtual functions `print` (line 85) and `earnings` (line 86). Note that function `virtualViaPointer` does not contain any `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` or `BasePlusCommissionEmployee` type information. The function knows only about base-class type `Employee`. Therefore, at compile time, the compiler cannot know which concrete class's functions to call through `baseClassPtr`. Yet at execution time, each virtual-function invocation calls the function on the object to which `baseClassPtr` points at that time. The output illustrates that the appropriate functions for each class are indeed invoked and that each object's proper information is displayed. For instance, the

weekly salary is displayed for the `SalariedEmployee`, and the gross sales are displayed for the `CommissionEmployee` and `BasePlusCommissionEmployee`. Also note that obtaining the earnings of each `Employee` polymorphically in line 86 produces the same results as obtaining these employees' earnings via static binding in lines 44, 46, 48 and 50. All virtual function calls to `print` and `earnings` are resolved at runtime with dynamic binding.

Finally, another `for` statement (lines 7576) traverses `employees` and invokes function `virtualViaReference` (lines 9195) for each element in the `vector`. Function `virtualViaReference` receives in its parameter `baseClassRef` (of type `const Employee &`) a reference formed by dereferencing the pointer stored in each `employees` element (line 76). Each call to `virtualViaReference` invokes virtual functions `print` (line 93) and `earnings` (line 94) via reference `baseClassRef` to demonstrate that polymorphic processing occurs with base-class references as well. Each virtual-function invocation calls the function on the object to which `baseClassRef` refers at runtime. This is another example of dynamic binding. The output produced using base-class references is identical to the output produced using base-class pointers.

---

[Page 728]

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 730]

The `earnings` function pointer in the vtable for class `CommissionEmployee` points to `CommissionEmployee`'s `earnings` function that returns the employee's gross sales multiplied by commission rate. The `print` function pointer points to the `CommissionEmployee` version of the function, which prints the employee's type, name, social security number, commission rate and gross sales. As in class `HourlyEmployee`, both functions override the functions in class `Employee`.

The `earnings` function pointer in the vtable for class `BasePlusCommissionEmployee` points to `BasePlusCommissionEmployee`'s `earnings` function that returns the employee's base salary plus gross sales multiplied by commission rate. The `print` function pointer points to the `BasePlusCommissionEmployee` version of the function, which prints the employee's base salary plus the type, name, social security number, commission rate and gross sales. Both functions override the functions in class `CommissionEmployee`.

Notice that in our `Employee` case study, each concrete class provides its own implementation for virtual functions `earnings` and `print`. You have already learned that each class which inherits directly from abstract base class `Employee` must implement `earnings` in order to be a concrete class, because `earnings` is a pure virtual function. These classes do not need to implement function `print`, however, to be considered concrete `print` is not a pure virtual function and derived classes can inherit class `Employee`'s implementation of `print`. Furthermore, class `BasePlusCommissionEmployee` does not have to implement either function `print` or `earnings` both function implementations can be inherited from class `CommissionEmployee`. If a class in our hierarchy were to inherit function implementations in this manner, the vtable pointers for these functions would simply point to the function implementation that was being inherited. For example, if `BasePlusCommissionEmployee` did not override `earnings`, the `earnings` function pointer in the vtable for class `BasePlusCommissionEmployee` would point to the same `earnings` function as the vtable for class `CommissionEmployee` points to.

Polymorphism is accomplished through an elegant data structure involving three levels of pointers. We have discussed one level the function pointers in the vtable. These point to the actual functions that execute when a virtual function is invoked.

Now we consider the second level of pointers. Whenever an object of a class with one or more virtual functions is instantiated, the compiler attaches to the object a pointer to the vtable for that class. This pointer is normally at the front of the object, but it is not required to be implemented that way. In Fig. 13.24, these pointers are associated with the objects created in Fig. 13.23 (one object for each of the types `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`). Notice that the diagram displays each of the object's data member values. For example, the `salariedEmployee` object contains a pointer to the `SalariedEmployee` vtable; the object also contains the values John Smith, 111-11-1111 and \$800.00.

The third level of pointers simply contains the handles to the objects that receive the virtual function calls. The handles in this level may also be references. Note that Fig. 13.24 depicts the vector employees that contains Employee pointers.

Now let us see how a typical virtual function call executes. Consider the call `baseClassPtr->print()` in function `virtualViaPointer` (line 85 of Fig. 13.23). Assume that `baseClassPtr` contains `employees[ 1 ]` (i.e., the address of object `hourlyEmployee` in `employees`). When the compiler compiles this statement, it determines that the call is indeed being made via a base-class pointer and that `print` is a virtual function.

[Page 731]

The compiler determines that `print` is the second entry in each of the vtables. To locate this entry, the compiler notes that it will need to skip the first entry. Thus, the compiler compiles an **offset** or **displacement** of four bytes (four bytes for each pointer on today's popular 32-bit machines, and only one pointer needs to be skipped) into the table of machine-language object-code pointers to find the code that will execute the virtual function call.

The compiler generates code that performs the following operations [Note: The numbers in the list correspond to the circled numbers in Fig. 13.24]:

**1.**

Select the  $i^{\text{th}}$  entry of `employees` (in this case, the address of object `hourlyEmployee`), and pass it as an argument to function `virtualViaPointer`. This sets parameter `baseClassPtr` to point to `hourlyEmployee`.

**2.**

Dereference that pointer to get to the `hourlyEmployee` object which, as you recall, begins with a pointer to the `HourlyEmployee` vtable.

**3.**

Dereference `hourlyEmployee`'s vtable pointer to get to the `HourlyEmployee` vtable.

**4.**

Skip the offset of four bytes to select the `print` function pointer.

**5.**

Dereference the `print` function pointer to form the "name" of the actual function to execute, and use the function call operator `( )` to execute the appropriate `print` function, which in this case prints the employee's type, name, social security number, hourly wage and hours worked.

The data structures of Fig. 13.24 may appear to be complex, but this complexity is managed by the compiler and hidden from you, making polymorphic programming straightforward. The pointer dereferencing operations and memory accesses that occur on every `virtual` function call require some additional execution time. The vtables and the vtable pointers added to the objects require some additional memory. You now have enough information to determine whether `virtual` functions are appropriate for your programs.

### Performance Tip 13.1



Polymorphism, as typically implemented with `virtual` functions and dynamic binding in C++, is efficient. Programmers may use these capabilities with nominal impact on performance.

### Performance Tip 13.2



`Virtual` functions and dynamic binding enable polymorphic programming as an alternative to `switch` logic programming. Optimizing compilers normally generate polymorphic code that runs as efficiently as hand-coded `switch`-based logic. The overhead of polymorphism is acceptable for most applications. But in some situations real-time applications with stringent performance requirements, for example the overhead of polymorphism may be too high.

### Software Engineering Observation 13.11



Dynamic binding enables independent software vendors (ISVs) to distribute software without revealing proprietary secrets. Software distributions can consist of only header files and object files no source code needs to be revealed. Software developers can then use inheritance to derive new classes from those provided by the ISVs. Other software that worked with the classes the ISVs provided will still work with the derived classes and will use the overridden `virtual` functions provided in these classes (via dynamic binding).

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 734]

**Figure 13.25. Demonstrating downcasting and run-time type information.**

(This item is displayed on pages 732 - 734 in the print version)

```
1 // Fig. 13.25: fig13_25.cpp
2 // Demonstrating downcasting and run-time type information.
3 // NOTE: For this example to run in Visual C++ .NET,
4 // you need to enable RTTI (Run-Time Type Info) for the project.
5 #include <iostream>
6 using std::cout;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12
13 #include <vector>
14 using std::vector;
15
16 #include <typeinfo>
17
18 // include definitions of classes in Employee hierarchy
19 #include "Employee.h"
20 #include "SalariedEmployee.h"
21 #include "HourlyEmployee.h"
22 #include "CommissionEmployee.h"
23 #include "BasePlusCommissionEmployee.h"
24
25 int main()
26 {
27 // set floating-point output formatting
28 cout << fixed << setprecision(2);
29
30 // create vector of four base-class pointers
31 vector < Employee * > employees(4);
32
33 // initialize vector with various kinds of Employees
34 employees[0] = new SalariedEmployee(
35 "John", "Smith", "111-11-1111", 800);
36 employees[1] = new HourlyEmployee(
```

```

37 "Karen", "Price", "222-22-2222", 16.75, 40);
38 employees[2] = new CommissionEmployee(
39 "Sue", "Jones", "333-33-3333", 10000, .06);
40 employees[3] = new BasePlusCommissionEmployee(
41 "Bob", "Lewis", "444-44-4444", 5000, .04, 300);
42
43 // polymorphically process each element in vector employees
44 for (size_t i = 0; i < employees.size(); i++)
45 {
46 employees[i]->print(); // output employee information
47 cout << endl;
48
49 // downcast pointer
50 BasePlusCommissionEmployee *derivedPtr =
51 dynamic_cast < BasePlusCommissionEmployee * >
52 (employees[i]);
53
54 // determine whether element points to base-salaried
55 // commission employee
56 if (derivedPtr != 0) // 0 if not a BasePlusCommissionEmployee
57 {
58 double oldBaseSalary = derivedPtr->getBaseSalary();
59 cout << "old base salary: $" << oldBaseSalary << endl;
60 derivedPtr->setBaseSalary(1.10 * oldBaseSalary);
61 cout << "new base salary with 10% increase is: $"
62 << derivedPtr->getBaseSalary() << endl;
63 } // end if
64
65 cout << "earned $" << employees[i]->earnings() << "\n\n";
66 } // end for
67
68 // release objects pointed to by vector's elements
69 for (size_t j = 0; j < employees.size(); j++)
70 {
71 // output class name
72 cout << "deleting object of "
73 << typeid(*employees[j]).name() << endl;
74
75 delete employees[j];
76 } // end for
77
78 return 0;
79 } // end main

```

```

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
deleting object of class HourlyEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee

```

The for statement at lines 4466 iterates through the employees vector and displays each Employee's information by invoking member function print (line 46). Recall that because print is declared virtual in base class Employee, the system invokes the appropriate derived-class object's print function.

In this example, as we encounter BasePlusCommissionEmployee objects, we wish to increase their base salary by 10 percent. Since we process the employees generically (i.e., polymorphically), we cannot (with the techniques we've learned) be certain as to which type of Employee is being manipulated at any given time. This creates a problem, because BasePlusCommissionEmployee employees must be identified when we encounter them so they can receive the 10 percent salary increase. To accomplish this, we use operator `dynamic_cast` (line 51) to determine whether the type of each object is BasePlusCommissionEmployee. This is the downcast operation we referred to in [Section 13.3.3](#). Lines 5052 dynamically downcast `employees[i]` from type `Employee *` to type `BasePlusCommissionEmployee *`. If the vector element points to an object that is a

BasePlusCommissionEmployee object, then that object's address is assigned to commissionPtr; otherwise, 0 is assigned to derived-class pointer derivedPtr.

If the value returned by the `dynamic_cast` operator in lines 5052 is not 0, the object is the correct type and the `if` statement (lines 5663) performs the special processing required for the BasePlusCommissionEmployee object. Lines 58, 60 and 62 invoke BasePlusCommissionEmployee functions `getBaseSalary` and `setBaseSalary` to retrieve and update the employee's salary.

[Page 735]

Line 65 invokes member function `earnings` on the object to which `employees[i]` points. Recall that `earnings` is declared `virtual` in the base class, so the program invokes the derived-class object's `earnings` function another example of dynamic binding.

The `for` loop at lines 6976 displays each employee's object type and uses the `delete` operator to deallocate the dynamic memory to which each `vector` element points. Operator `typeid` (line 73) returns a reference to an object of class `type_info` that contains the information about the type of its operand, including the name of that type. When invoked, `type_info` member function `name` (line 73) returns a pointer-based string that contains the type name (e.g., "class BasePlusCommissionEmployee") of the argument passed to `typeid`. [Note: The exact contents of the string returned by `type_info` member function `name` may vary by compiler.] To use `typeid`, the program must include header file `<typeinfo>` (line 16).

Note that we avoid several compilation errors in this example by downcasting an `Employee` pointer to a `BasePlusCommissionEmployee` pointer (lines 5052). If we remove the `dynamic_cast` from line 51 and attempt to assign the current `Employee` pointer directly to `BasePlusCommissionEmployee` pointer `commissionPtr`, we will receive a compilation error. C++ does not allow a program to assign a base-class pointer to a derived-class pointer because the is-a relationship does not apply a `CommissionEmployee` is not a `BasePlusCommissionEmployee`. The is-a relationship applies only between the derived class and its base classes, not vice versa.

Similarly, if lines 58, 60 and 62 used the current base-class pointer from `employees`, rather than derived-class pointer `commissionPtr`, to invoke derived-class-only functions `getBaseSalary` and `setBaseSalary`, we would receive a compilation error at each of these lines. As you learned in Section 13.3.3, attempting to invoke derived-class-only functions through a base-class pointer is not allowed. Although lines 58, 60 and 62 execute only if `commissionPtr` is not 0 (i.e., if the cast can be performed), we cannot attempt to invoke derived class `BasePlusCommissionEmployee` functions `getBaseSalary` and `setBaseSalary` on the base class `Employee` pointer. Recall that, using a base class `Employee` pointer, we can invoke only functions found in base class `Employee`'s `earnings`, `print` and `Employee`'s get and set functions.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 736]

## Good Programming Practice 13.2



If a class has `virtual` functions, provide a `virtual` destructor, even if one is not required for the class. Classes derived from this class may contain destructors that must be called properly.

## Common Programming Error 13.5



Constructors cannot be `virtual`. Declaring a constructor `virtual` is a compilation error.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 737]

The UML specifies a relationship called a **generalization** to model inheritance. Figure 13.27 is the class diagram that models the inheritance relationship between base class `transaction` and its three derived classes. The arrows with triangular hollow arrowheads indicate that classes `BalanceInquiry`, `Withdrawal` and `Deposit` are derived from class `transaction`. Class `transaction` is said to be a generalization of its derived classes. The derived classes are said to be **specializations** of class `TTransaction`.

Classes `BalanceInquiry`, `Withdrawal` and `Deposit` share integer attribute `accountNumber`, so we factor out this common attribute and place it in base class `TTransaction`. We no longer list `accountNumber` in the second compartment of each derived class, because the three derived classes inherit this attribute from `transaction`. Recall, however, that derived classes cannot access **private** attributes of a base class. We therefore include **public** member function `getAccountNumber` in class `TTransaction`. Each derived class inherits this member function, enabling the derived class to access its `accountNumber` as needed to execute a transaction.

According to Fig. 13.26, classes `BalanceInquiry`, `Withdrawal` and `Deposit` also share operation `execute`, so base class `transaction` should contain **public** member function `execute`. However, it does not make sense to implement `execute` in class `transaction`, because the functionality that this member function provides depends on the specific type of the actual transaction. We therefore declare member function `execute` as a **pure virtual** function in base class `transaction`. This makes `transaction` an abstract class and forces any class derived from `transaction` that must be a concrete class (i.e., `BalanceInquiry`, `Withdrawal` and `Deposit`) to implement pure virtual member function `execute` to make the derived class concrete. The UML requires that we place abstract class names (and pure virtual functions **abstract operations** in the UML) in italics, so `transaction` and its member function `execute` appear in italics in Fig. 13.27. Note that operation `execute` is not italicized in derived classes `BalanceInquiry`, `Withdrawal` and `Deposit`. Each derived class overrides base class `transaction`'s `execute` member function with an appropriate implementation. Note that Fig. 13.27 includes operation `execute` in the third compartment of classes `BalanceInquiry`, `Withdrawal` and `Deposit`, because each class has a different concrete implementation of the overridden member function.

[Page 738]

As you learned in this chapter, a derived class can inherit interface or implementation from a base class. Compared to a hierarchy designed for implementation inheritance, one designed for interface inheritance tends to have its functionality lower in the hierarchy: a base class signifies one or more functions that should be defined by each class in the hierarchy, but the individual derived classes provide their own implementations of the function(s). The inheritance hierarchy designed for the ATM system takes advantage of this type of inheritance, which provides the ATM with an elegant way to execute all transactions "in the general." Each class derived from `transaction` inherits some implementation details (e.g., data member `accountNumber`), but the primary benefit of incorporating inheritance into our system is that the derived classes share a common interface (e.g., pure virtual member function `execute`). The ATM can aim a `transaction` pointer at any `transaction`, and when the ATM invokes `execute` through this pointer, the version of `execute` appropriate to that `transaction` (i.e., implemented in that derived class's .cpp file) runs automatically. For example, suppose a user chooses to perform a balance inquiry. The ATM aims a `transaction` pointer at a new object of class `BalanceInquiry`, which the C++ compiler allows because a `BalanceInquiry` is a `transaction`. When the ATM uses this pointer to invoke `execute`, `BalanceInquiry`'s version of `execute` is called.

This polymorphic approach also makes the system easily extensible. Should we wish to create a new transaction type (e.

g., funds transfer or bill payment), we would just create an additional `TTransaction` derived class that overrides the `execute` member function with a version appropriate for the new transaction type. We would need to make only minimal changes to the system code to allow users to choose the new transaction type from the main menu and for the ATM to instantiate and execute objects of the new derived class. The ATM could execute transactions of the new type using the current code, because it executes all transactions identically.

As you learned earlier in the chapter, an abstract class like `TTransaction` is one for which the programmer never intends to instantiate objects. An abstract class simply declares common attributes and behaviors for its derived classes in an inheritance hierarchy. Class `TTransaction` defines the concept of what it means to be a transaction that has an account number and executes. You may wonder why we bother to include pure virtual member function `execute` in class `TTransaction` if `execute` lacks a concrete implementation. Conceptually, we include this member function because it is the defining behavior of all transaction executing. Technically, we must include member function `execute` in base class `TTransaction` so that the ATM (or any other class) can polymorphically invoke each derived class's overridden version of this function through a `TTransaction` pointer or reference.

Derived classes `BalanceInquiry`, `Withdrawal` and `Deposit` inherit attribute `accountNumber` from base class `transaction`, but classes `Withdrawal` and `Deposit` contain the additional attribute `amount` that distinguishes them from class `BalanceInquiry`. Classes `Withdrawal` and `Deposit` require this additional attribute to store the amount of money that the user wishes to withdraw or deposit. Class `BalanceInquiry` has no need for such an attribute and requires only an account number to execute. Even though two of the three transaction derived classes share this attribute, we do not place it in base class `transaction` we place only features common to all the derived classes in the base class, so derived classes do not inherit unnecessary attributes (and operations).

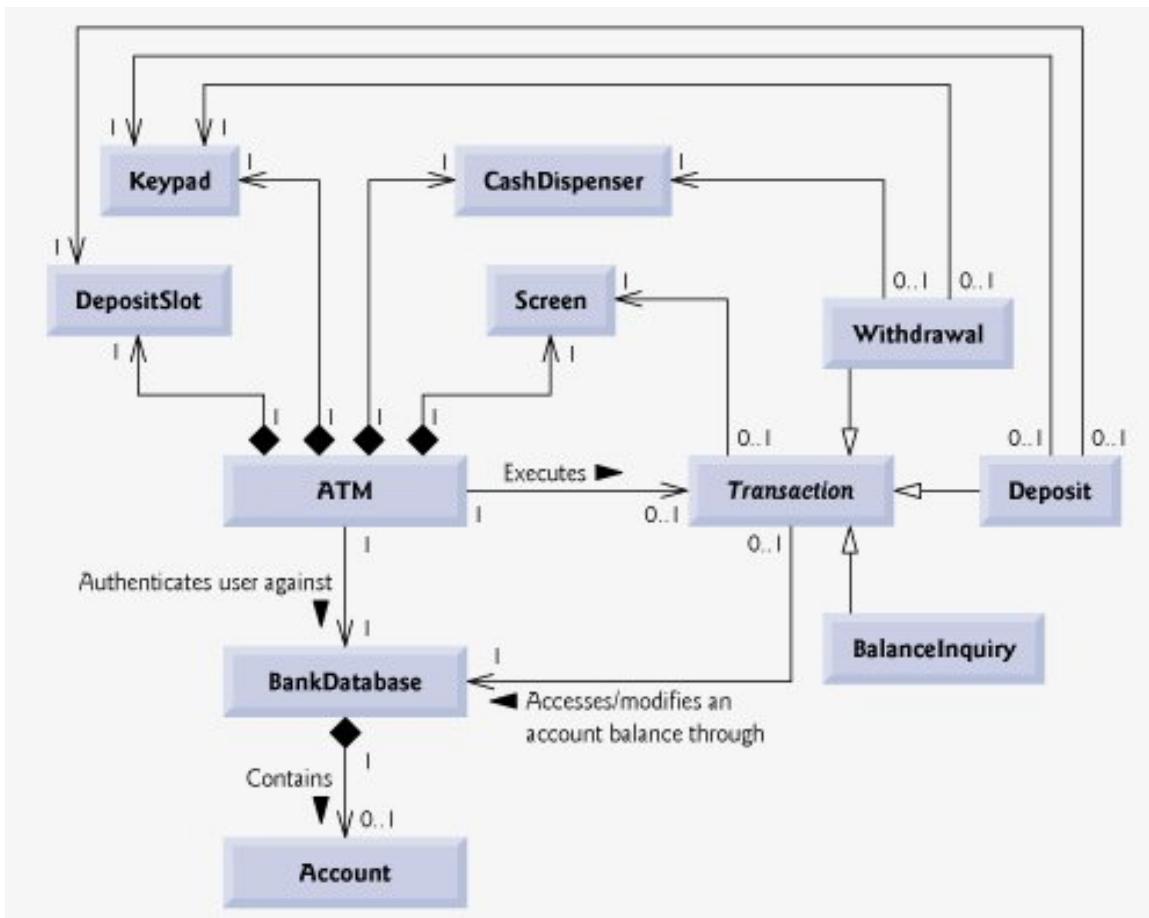
[Figure 13.28](#) presents an updated class diagram of our model that incorporates inheritance and introduces class `TTransaction`. We model an association between class `ATM` and class `transaction` to show that the ATM, at any given moment, either is executing a transaction or is not (i.e., zero or one objects of type `TTransaction` exist in the system at a time). Because a `Withdrawal` is a type of `TTransaction`, we no longer draw an association line directly between class `ATM` and class `Withdrawal` derived class `Withdrawal` inherits base class `transaction`'s association with class `ATM`. Derived classes `BalanceInquiry` and `Deposit` also inherit this association, which replaces the previously omitted associations between classes `BalanceInquiry` and `Deposit` and class `ATM`. Note again the use of triangular hollow arrowheads to indicate the specializations of class `TTransaction`, as indicated in [Fig. 13.27](#).

---

[Page 739]

**Figure 13.28. Class diagram of the ATM system (incorporating inheritance). Note that abstract class name `transaction` appears in italics.**

[\[View full size image\]](#)



We also add an association between class **TRansaction** and the **BankDatabase** (Fig. 13.28). All **TRansactions** require a reference to the **BankDatabase** so they can access and modify account information. Each **TRansaction** derived class inherits this reference, so we no longer model the association between class **Withdrawal** and the **BankDatabase**. Note that the association between class **transaction** and the **BankDatabase** replaces the previously omitted associations between classes **BalanceInquiry** and **Deposit** and the **BankDatabase**.

We include an association between class **transaction** and the **Screen** because all **transactions** display output to the user via the **Screen**. Each derived class inherits this association. Therefore, we no longer include the association previously modeled between **Withdrawal** and the **Screen**. Class **withdrawal** still participates in associations with the **CashDispenser** and the **Keypad**, however these associations apply to derived class **Withdrawal** but not to derived classes **BalanceInquiry** and **Deposit**, so we do not move these associations to base class **transaction**.

---

[Page 740]

Our class diagram incorporating inheritance (Fig. 13.28) also models **Deposit** and **BalanceInquiry**. We show associations between **Deposit** and both the **DepositSlot** and the **Keypad**. Note that class **BalanceInquiry** takes part in no associations other than those inherited from class **transaction**. **BalanceInquiry** interacts only with the **BankDatabase** and the **Screen**.

The class diagram of Fig. 9.20 showed attributes and operations with visibility markers. Now we present a modified class diagram in Fig. 13.29 that includes abstract base class **transaction**. This abbreviated diagram does not show inheritance relationships (these appear in Fig. 13.28), but instead shows the attributes and operations after we have employed inheritance in our system. Note that abstract class name **transaction** and abstract operation name **execute** in class **transaction** appear in italics. To save space, as we did in Fig. 4.24, we do not include those attributes shown

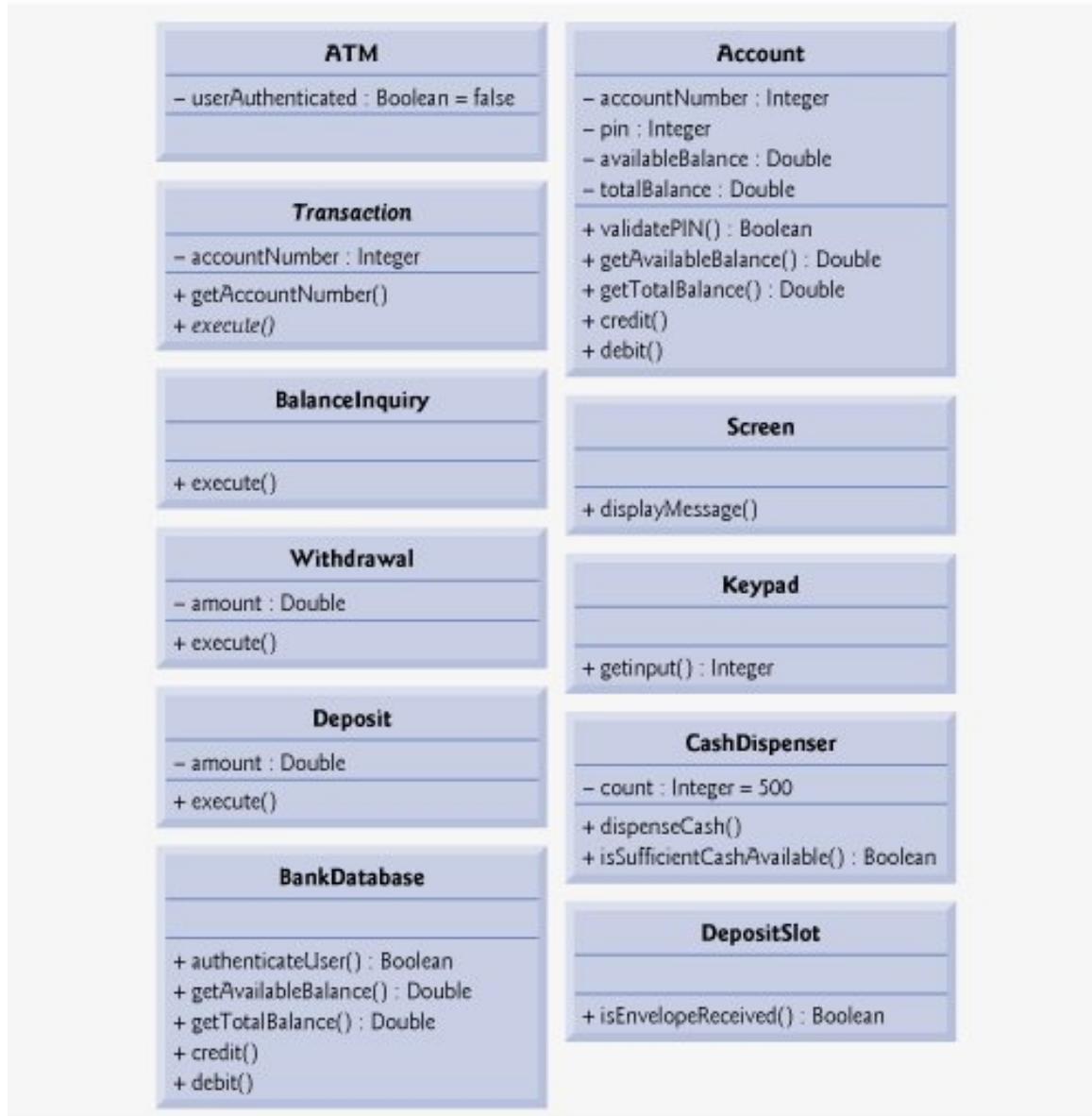
by associations in Fig. 13.28 we do, however, include them in the C++ implementation in Appendix G. We also omit all operation parameters, as we did in Fig. 9.20 incorporating inheritance does not affect the parameters already modeled in Figs. 6.226.25.

[Page 741]

**Figure 13.29. Class diagram after incorporating inheritance into the system.**

(This item is displayed on page 740 in the print version)

[View full size image]



Software Engineering Observation 13.12



A complete class diagram shows all the associations among classes and all the attributes and operations for each class. When the number of class attributes, operations and associations is substantial (as in Fig. 13.28 and Fig. 13.29), a good practice that promotes readability is to divide this information between two class diagrams—one focusing on associations and the other on attributes and operations. However, when examining classes modeled in this fashion, it is crucial to consider both class diagrams to get a complete view of the classes. For example, one must refer to Fig. 13.28 to observe the inheritance relationship between `transaction` and its derived classes that is omitted from Fig. 13.29.

## Implementing the ATM System Design Incorporating Inheritance

In Section 9.12, we began implementing the ATM system design in C++ code. We now modify our implementation to incorporate inheritance, using class `Withdrawal` as an example.

1.

If a class A is a generalization of class B, then class B is derived from (and is a specialization of) class A. For example, abstract base class `TTransaction` is a generalization of class `Withdrawal`. Thus, class `Withdrawal` is derived from (and is a specialization of) class `transaction`. Figure 13.30 contains a portion of class `Withdrawal`'s header file, in which the class definition indicates the inheritance relationship between `Withdrawal` and `transaction` (line 9).

**Figure 13.30. `Withdrawal` class definition that derives from `TTransaction`.**

(This item is displayed on page 742 in the print version)

```

1 // Fig. 13.30: Withdrawal.h
2 // Definition of class Withdrawal that represents a withdrawal transaction
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 #include "Transaction.h" // Transaction class definition
7
8 // class Withdrawal derives from base class Transaction
9 class Withdrawal : public Transaction
10 {
11 }; // end class Withdrawal
12
13 #endif // WITHDRAWAL_H

```

2.

If class A is an abstract class and class B is derived from class A, then class B must implement the pure virtual functions of class A if class B is to be a concrete class. For example, class `transaction` contains pure virtual function `execute`, so class `Withdrawal` must implement this member function if we want to instantiate a `Withdrawal` object. Figure 13.31 contains the C++ header file for class `Withdrawal` from Fig. 13.28 and Fig. 13.29. Class `Withdrawal` inherits data member `accountNumber` from base class `transaction`, so `Withdrawal` does not declare this data member. Class `Withdrawal` also inherits references to the `Screen` and

the `BankDatabase` from its base class `transaction`, so we do not include these references in our code. [Figure 13.29](#) specifies attribute `amount` and operation `execute` for class `Withdrawal`. Line 19 of [Fig. 13.31](#) declares a data member for attribute `amount`. Line 16 contains the function prototype for operation `execute`. Recall that, to be a concrete class, derived class `Withdrawal` must provide a concrete implementation of the pure virtual function `execute` in base class `transaction`. The prototype in line 16 signals your intent to override the base class pure virtual function. You must provide this prototype if you will provide an implementation in the `.cpp` file. We present this implementation in [Appendix G](#). The `keypad` and `cashDispenser` references (lines 2021) are data members derived from `Withdrawal`'s associations in [Fig. 13.28](#). In the implementation of this class in [Appendix G](#), a constructor initializes these references to actual objects. Once again, to be able to compile the declarations of the references in lines 2021, we include the forward declarations in lines 89.

---

[Page 742]

**Figure 13.31. `Withdrawal` class header file based on [Fig. 13.28](#) and [Fig. 13.29](#).**

```

1 // Fig. 13.31: Withdrawal.h
2 // Definition of class Withdrawal that represents a withdrawal transaction
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 #include "Transaction.h" // Transaction class definition
7
8 class Keypad; // forward declaration of class Keypad
9 class CashDispenser; // forward declaration of class CashDispenser
10
11 // class Withdrawal derives from base class Transaction
12 class Withdrawal : public Transaction
13 {
14 public:
15 // member function overriding execute in base class Transaction
16 virtual void execute(); // perform the transaction
17 private:
18 // attributes
19 double amount; // amount to withdraw
20 Keypad &keypad; // reference to ATM's keypad
21 CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22 }; // end class Withdrawal
23
24 #endif // WITHDRAWAL_H

```

## ATM Case Study Wrap-Up

This concludes our object-oriented design of the ATM system. A complete C++ implementation of the ATM system in 877 lines of code appears in [Appendix G](#). This working implementation uses key programming notions, including classes, objects, encapsulation, visibility, composition, inheritance and polymorphism. The code is abundantly commented and conforms to the coding practices you've learned. Mastering this code is a wonderful capstone experience for you after studying [Chapters 1213](#).

## Software Engineering Case Study Self-Review Exercises

**13.1** The UML uses an arrow with a \_\_\_\_\_ to indicate a generalization relationship.

a.

solid filled arrowhead

b.

triangular hollow arrowhead

c.

diamond-shaped hollow arrowhead

d.

stick arrowhead

**13.2** State whether the following statement is true or false, and if false, explain why: The UML requires that we underline abstract class names and operation names.

**13.3** Write a C++ header file to begin implementing the design for class `transaction` specified in Fig. 13.28 and Fig. 13.29. Be sure to include `private` references based on class `TRansaction`'s associations. Also be sure to include `public` get functions for any of the `private` data members that the derived classes must access to perform their tasks.

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

**13.1** b.

**13.2** False. The UML requires that we italicize abstract class names and operation names.

- 13.3** The design for class transaction yields the header file in Fig. 13.32. In the implementation in Appendix G, a constructor initializes private reference attributes screen and bankDatabase to actual objects, and member functions getScreen and getBankDatabase access these attributes. These member functions allow classes derived from TTransaction to access the ATM's screen and interact with the bank's database.

**Figure 13.32. transaction class header file based on Fig. 13.28 and Fig. 13.29.**

```

1 // Fig. 13.32: Transaction.h
2 // Transaction abstract base class definition.
3 #ifndef TRANSACTION_H
4 #define TRANSACTION_H
5
6 class Screen; // forward declaration of class Screen
7 class BankDatabase; // forward declaration of class BankDatabase
8
9 class Transaction
10 {
11 public:
12 int getAccountNumber(); // return account number
13 Screen &getScreen(); // return reference to screen
14 BankDatabase &getBankDatabase(); // return reference to bank database
15
16 // pure virtual function to perform the transaction
17 virtual void execute() = 0; // overridden in derived classes
18 private:
19 int accountNumber; // indicates account involved
20 Screen &screen; // reference to the screen of the ATM
21 BankDatabase &bankDatabase; // reference to the account info database
22 }; // end class Transaction
23
24 #endif // TRANSACTION_H

```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 744]

## 13.11. Wrap-Up

In this chapter we discussed polymorphism, which enables us to "program in the general" rather than "program in the specific," and we showed how this makes programs more extensible. We began with an example of how polymorphism would allow a screen manager to display several "space" objects. We then demonstrated how base-class and derived-class pointers can be aimed at base-class and derived-class objects. We said that aiming base-class pointers at base-class objects is natural, as is aiming derived-class pointers at derived-class objects. Aiming base-class pointers at derived-class objects is also natural because a derived-class object is an object of its base class. You learned why aiming derived-class pointers at base-class objects is dangerous and why the compiler disallows such assignments. We introduced `virtual` functions, which enable the proper functions to be called when objects at various levels of an inheritance hierarchy are referenced (at execution time) via base-class pointers. This is known as dynamic or late binding. We then discussed pure `virtual` functions (`virtual` functions that do not provide an implementation) and abstract classes (classes with one or more pure `virtual` functions). You learned that abstract classes cannot be used to instantiate objects, while concrete classes can. We then demonstrated using abstract classes in an inheritance hierarchy. You learned how polymorphism works "under the hood" with vtables that are created by the compiler. We discussed downcasting base-class pointers to derived-class pointers to enable a program to call derived-class-only member functions. The chapter concluded with a discussion of `virtual` destructors, and how they ensure that all appropriate destructors in an inheritance hierarchy run on a derived-class object when that object is deleted via a base-class pointer.

In the next chapter, we discuss templates, a sophisticated feature of C++ that enables programmers to define a family of related classes or functions with a single code segment.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 745]

- If a class is derived from a class with a pure `virtual` function and that derived class does not supply a definition for that pure `virtual` function, then that `virtual` function remains pure in the derived class. Consequently, the derived class is also an abstract class.
- C++ enables polymorphism—the ability for objects of different classes related by inheritance to respond differently to the same member-function call.
- Polymorphism is implemented via `virtual` functions and dynamic binding.
- When a request is made through a base-class pointer or reference to use a `virtual` function, C++ chooses the correct overridden function in the appropriate derived class associated with the object.
- Through the use of `virtual` functions and polymorphism, a member-function call can cause different actions, depending on the type of the object receiving the call.
- Although we cannot instantiate objects of abstract base classes, we can declare pointers and references to objects of abstract base classes. Such pointers and references can be used to enable polymorphic manipulations of derived-class objects instantiated from concrete derived classes.
- Dynamic binding requires that at runtime, the call to a `virtual` member function be routed to the `virtual` function version appropriate for the class. A `virtual` function table called the `vtable` is implemented as an array containing function pointers. Each class with `virtual` functions has a `vtable`. For each `virtual` function in the class, the `vtable` has an entry containing a function pointer to the version of the `virtual` function to use for an object of that class. The `virtual` function to use for a particular class could be the function defined in that class, or it could be a function inherited either directly or indirectly from a base class higher in the hierarchy.
- When a base class provides a `virtual` member function, derived classes can override the `virtual` function, but they do not have to override it. Thus, a derived class can use a base class's version of a `virtual` function.
- Each object of a class with `virtual` functions contains a pointer to the `vtable` for that class. When a function call is made from a base-class pointer to a derived-class object, the appropriate function pointer in the `vtable` is obtained and dereferenced to complete the call at execution time. This `vtable` lookup and pointer dereferencing require nominal runtime overhead.
- Any class that has one or more 0 pointers in its `vtable` is an abstract class. Classes without any 0 `vtable` pointers are concrete classes.
- New kinds of classes are regularly added to systems. New classes are accommodated by dynamic binding (also called late binding). The type of an object need not be known at compile time for a `virtual`-function call to be compiled. At runtime, the appropriate member function will be called for the object to which the pointer points.
- Operator `dynamic_cast` checks the type of the object to which the pointer points, then determines whether this type has an `is-a` relationship with the type to which the pointer is being converted. If there is an `is-a` relationship, `dynamic_cast` returns the object's address. If not, `dynamic_cast` returns 0.
- Operator `typeid` returns a reference to an object of class `type_info` that contains information about the type of its operand, including the name of the type. To use `typeid`, the program must include header file `<typeinfo>`.
- When invoked, `type_info` member function `name` returns a pointer-based string that contains the name of the type that the `type_info` object represents.
- Operators `dynamic_cast` and `typeid` are part of C++'s run-time type information (RTTI) feature,

which allows a program to determine an object's type at runtime.

- Declare the base-class destructor `virtual` if the class contains `virtual` functions. This makes all derived-class destructors `virtual`, even though they do not have the same name as the base-class destructor. If an object in the hierarchy is destroyed explicitly by applying the `delete` operator to a base-class pointer to a derived-class object, the destructor for the appropriate class is called. After a derived-class destructor runs, the destructors for all of that class's base classes run all the way up the hierarchy the root class's destructor runs last.

---

[Page 746]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 746 (continued)]

## Terminology

abstract base class

abstract class

base-class pointer to a base-class object

base-class pointer to a derived-class object

concrete class

dangerous pointer manipulation

derived-class pointer to a base-class object

derived-class pointer to a derived-class object

displacement

downcasting

dynamic binding

dynamic casting

dynamic\_cast

dynamically determine function to execute

flow of control of a virtual function call

implementation inheritance

interface inheritance

iterator class

late binding

name function of class `type_info`

nonvirtual destructor

object's vtable pointer

offset into a vtable

override a function

polymorphic programming

polymorphism

polymorphism as an alternative to `switch` logic

programming in the general

programming in the specific

pure specifier

pure virtual function

RTTI (run-time type information)

static binding

`switch` logic

`type_info` class

`typeid` operator

<typeinfo> header file

virtual destructor

virtual function

virtual function table (vtable)

virtual keyword

vtable

vtable pointer

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 746 (continued)]

## Self-Review Exercises

**13.1** Fill in the blanks in each of the following statements:

a.

Treating a base-class object as a(n) \_\_\_\_\_ can cause errors.

b.

Polymorphism helps eliminate \_\_\_\_\_ logic.

c.

If a class contains at least one pure virtual function, it is a(n) \_\_\_\_\_ class.

d.

Classes from which objects can be instantiated are called \_\_\_\_\_ classes.

e.

Operator \_\_\_\_\_ can be used to downcast base-class pointers safely.

f.

Operator `typeid` returns a reference to a(n) \_\_\_\_\_ object.

g.

\_\_\_\_\_ involves using a base-class pointer or reference to invoke virtual functions on base-class and derived-class objects.

Overridable functions are declared using keyword \_\_\_\_\_.

i.

Casting a base-class pointer to a derived-class pointer is called \_\_\_\_\_.

**13.2** State whether each of the following is true or false. If false, explain why.

a.

All virtual functions in an abstract base class must be declared as pure virtual functions.

b.

Referring to a derived-class object with a base-class handle is dangerous.

c.

A class is made abstract by declaring that class `virtual`.

d.

If a base class declares a pure virtual function, a derived class must implement that function to become a concrete class.

e.

Polymorphic programming can eliminate the need for `switch` logic.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 747]

## Answers to Self-Review Exercises

**13.1** a) derived-class object. b) switch. c) abstract. d) concrete. e) `dynamic_cast`. f) `type_info`. g) Polymorphism. h) `virtual`. i) downcasting.

**13.2** a) False. An abstract base class can include virtual functions with implementations. b) False. Referring to a base-class object with a derived-class handle is dangerous. c) False. Classes are never declared `virtual`. Rather, a class is made abstract by including at least one pure virtual function in the class. d) True. e) True.

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 748]

### 13.15

(Package Inheritance Hierarchy) Use the Package inheritance hierarchy created in [Exercise 12.9](#) to create a program that displays the address information and calculates the shipping costs for several Packages. The program should contain a vector of Package pointers to objects of classes TwoDayPackage and OvernightPackage. Loop through the vector to process the Packages polymorphically. For each Package, invoke get functions to obtain the address information of the sender and the recipient, then print the two addresses as they would appear on mailing labels. Also, call each Package's calculateCost member function and print the result. Keep track of the total shipping cost for all Packages in the vector, and display this total when the loop terminates.

### 13.16

(Polymorphic Banking Program Using Account Hierarchy) Develop a polymorphic banking program using the Account hierarchy created in [Exercise 12.10](#). Create a vector of Account pointers to SavingsAccount and CheckingAccount objects. For each Account in the vector, allow the user to specify an amount of money to withdraw from the Account using member function debit and an amount of money to deposit into the Account using member function credit. As you process each Account, determine its type. If an Account is a SavingsAccount, calculate the amount of interest owed to the Account using member function calculateInterest, then add the interest to the account balance using member function credit. After processing an Account, print the updated account balance obtained by invoking base class member function getBalance.

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 750]

## Outline

[14.1 Introduction](#)

[14.2 Function Templates](#)

[14.3 Overloading Function Templates](#)

[14.4 Class Templates](#)

[14.5 Nontype Parameters and Default Types for Class Templates](#)

[14.6 Notes on Templates and Inheritance](#)

[14.7 Notes on Templates and Friends](#)

[14.8 Notes on Templates and static Members](#)

[14.9 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

[Exercises](#)

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 750 (continued)]

## 14.1. Introduction

In this chapter, we discuss one of C++'s more powerful software reuse features, namely [templates](#). **Function templates** and **class templates** enable programmers 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 function template for an array-sort function, then have C++ generate separate function-template specializations that will sort `int` arrays, `float` arrays, `string` arrays and so on. We introduced function templates in [Chapter 6](#). We present an additional discussion and example in this chapter.

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.

Note the distinction between templates and template specializations: Function templates and class templates are like stencils out of which we trace shapes; function-template specializations and class-template specializations are like the separate tracings that all have the same shape, but could, for example, be drawn in different colors.

In this chapter, we present a function template and a class template. We also consider the relationships between templates and other C++ features, such as overloading, inheritance, friends and `static` members. The design and details of the template mechanisms discussed here are based on the work of Bjarne Stroustrup as presented in his paper, Parameterized Types for C++, and as published in the Proceedings of the USENIX C++ Conference held in Denver, Colorado, in October 1988.

This chapter is only an introduction to templates. [Chapter 23](#), Standard Template Library (STL), presents an in-depth treatment of the template container classes, iterators and algorithms of the STL. [Chapter 23](#) contains dozens of live-code template-based examples illustrating more sophisticated template-programming techniques than those used here.

Software Engineering Observation 14.1



Most C++ compilers require the complete definition of a template to appear in the client source-code file that uses the template. For this reason and for reusability, templates are often defined in header files, which are then #included into the appropriate client source-code files. For class templates, this means that the member functions are also defined in the header file.

[PREV](#)[NEXT](#)

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 752]

```
void printArray(const int *array, int count)
{
 for (int i = 0; i < count; i++)
 cout << array[i] << " ";
 cout << endl;
} // end function printArray
```

The name of a template parameter can be declared only once in the template parameter list of a template header but can be used repeatedly in the function's header and body. Template parameter names among function templates need not be unique.

[Page 753]

**Figure 14.1** demonstrates function template `printArray` (lines 815). The program begins by declaring five-element `int` array `a`, seven-element `double` array `b` and six-element `char` array `c` (lines 2325, respectively). Then, the program outputs each array by calling `printArray` once with a first argument `a` of type `int *` (line 30), once with a first argument `b` of type `double *` (line 35) and once with a first argument `c` of type `char *` (line 40). The call in line 30, for example, causes the compiler to infer that `T` is `int` and to instantiate a `printArray` function-template specialization, for which type parameter `T` is `int`. The call in line 35 causes the compiler to infer that `T` is `double` and to instantiate a second `printArray` function-template specialization, for which type parameter `T` is `double`. The call in line 40 causes the compiler to infer that `T` is `char` and to instantiate a third `printArray` function-template specialization, for which type parameter `T` is `char`. It is important to note that if `T` (line 8) represents a user-defined type (which it does not in Fig. 14.1), there must be an overloaded stream insertion operator for that type; otherwise, the first stream insertion operator in line 12 will not compile.

## Common Programming Error 14.2



If a template is invoked with a user-defined type, and if that template uses functions or operators (e.g., `==`, `+`, `<=`) with objects of that class type, then those functions and operators must be overloaded for the user-defined type. Forgetting to overload such operators causes compilation errors.

In this example, the template mechanism saves the programmer from having to write three separate overloaded functions with prototypes

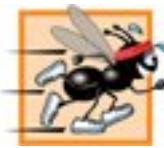
```
void printArray(const int *, int);
void printArray(const double *, int);
void printArray(const char *, int);
```

that all use the same code, except for type T (as used in line 9).

---

[Page 754]

### Performance Tip 14.1



Although templates offer software-reusability benefits, remember that multiple function-template specializations and class-template specializations are instantiated in a program (at compile time), despite the fact that the template is written only once. These copies can consume considerable memory. This is not normally an issue, though, because the code generated by the template is the same size as the code the programmer would have written to produce the separate overloaded functions.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 754 (continued)]

## 14.3. Overloading Function Templates

Function templates and overloading are intimately related. The function-template specializations generated from a function template all have the same name, so the compiler uses overloading resolution to invoke the proper function.

A function template may be overloaded in several ways. We can provide other function templates that specify the same function name but different function parameters. For example, function template `printArray` of Fig. 14.1 could be overloaded with another `printArray` function template with additional parameters `lowSubscript` and `highSubscript` to specify the portion of the array to output (see Exercise 14.4).

A function template also can be overloaded by providing nontemplate functions with the same function name but different function arguments. For example, function template `printArray` of Fig. 14.1 could be overloaded with a nontemplate version that specifically prints an array of character strings in neat, tabular format (see Exercise 14.5).

The compiler performs a matching process to determine what function to call when a function is invoked. First, the compiler finds all function templates that match the function named in the function call and creates specializations based on the arguments in the function call. Then, the compiler finds all the ordinary functions that match the function named in the function call. If one of the ordinary functions or function-template specializations is the best match for the function call, that ordinary function or specialization is used. If an ordinary function and a specialization are equally good matches for the function call, then the ordinary function is used. Otherwise, if there are multiple matches for the function call, the compiler considers the call to be ambiguous and the compiler generates an error message.

### Common Programming Error 14.3



If no matching function definition can be found for a particular function call, or if there are multiple matches, the compiler generates an error.

[◀ PREV](#)

[NEXT ▶](#)



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 755]

## Software Engineering Observation 14.2



Class templates encourage software reusability by enabling type-specific versions of generic classes to be instantiated.

Class templates are called **parameterized types**, because they require one or more type parameters to specify how to customize a "generic class" template to form a class-template specialization.

The programmer who wishes to produce a variety of class-template specializations writes only one class-template definition. Each time an additional class-template specialization is needed, the programmer uses a concise, simple notation, and the compiler writes the source code for the specialization the programmer requires. One Stack class template, for example, could thus become the basis for creating many Stack classes (such as "Stack of double," "Stack of int," "Stack of char," "Stack of Employee," etc.) used in a program.

### **Creating Class Template Stack< T >**

Note the Stack class-template definition in Fig. 14.2. It looks like a conventional class definition, except that it is preceded by the header (line 6)

```
template< typename T >
```

to specify a class-template definition with type parameter `T` which acts as a placeholder for the type of the `Stack` class to be created. The programmer need not specifically use identifier `T` any valid identifier can be used. The type of element to be stored on this `Stack` is mentioned generically as `T` throughout the `Stack` class header and member-function definitions. In a moment, we show how `T` becomes associated with a specific type, such as `double` or `int`. Due to the way this class template is designed, there are two constraints for nonfundamental data types used with this `Stack` they must have a default constructor (for use in line 44 to create the array that stores the stack elements), and they must support the assignment operator (lines 55 and 69).

### **Figure 14.2. Class template Stack.**

(This item is displayed on pages 756 - 757 in the print version)

```

1 // Fig. 14.2: Stack.h
2 // Stack class template.
3 #ifndef STACK_H
4 #define STACK_H
5
6 template< typename T >
7 class Stack
8 {
9 public:
10 Stack(int = 10); // default constructor (Stack size 10)
11
12 // destructor
13 ~Stack()
14 {
15 delete [] stackPtr; // deallocate internal space for Stack
16 } // end ~Stack destructor
17
18 bool push(const T&); // push an element onto the Stack
19 bool pop(T&); // pop an element off the Stack
20
21 // determine whether Stack is empty
22 bool isEmpty() const
23 {
24 return top == -1;
25 } // end function isEmpty
26
27 // determine whether Stack is full
28 bool isFull() const
29 {
30 return top == size - 1;
31 } // end function isFull
32
33 private:
34 int size; // # of elements in the Stack
35 int top; // location of the top element (-1 means empty)
36 T *stackPtr; // pointer to internal representation of the Stack
37 }; // end class template Stack
38
39 // constructor template
40 template< typename T >
41 Stack< T >::Stack(int s)
42 : size(s > 0 ? s : 10), // validate size
43 top(-1), // Stack initially empty
44 stackPtr(new T[size]) // allocate memory for elements
45 {
46 // empty body
47 } // end Stack constructor template

```

```

48
49 // push element onto Stack;
50 // if successful, return true; otherwise, return false
51 template< typename T >
52 bool Stack< T >::push(const T &pushValue)
53 {
54 if (!isFull())
55 {
56 stackPtr[++top] = pushValue; // place item on Stack
57 return true; // push successful
58 } // end if
59
60 return false; // push unsuccessful
61 } // end function template push
62
63 // pop element off Stack;
64 // if successful, return true; otherwise, return false
65 template< typename T >
66 bool Stack< T >::pop(T &popValue)
67 {
68 if (!isEmpty())
69 {
70 popValue = stackPtr[top--]; // remove item from Stack
71 return true; // pop successful
72 } // end if
73
74 return false; // pop unsuccessful
75 } // end function template pop
76
77 #endif

```

The member-function definitions of a class template are function templates. The member-function definitions that appear outside the class template definition each begin with the header

```
template< typename T >
```

(lines 40, 51 and 65). Thus, each definition resembles a conventional function definition, except that the `Stack` element type always is listed generically as type parameter `T`. The binary scope resolution operator is used with the class-template name `Stack< T >` (lines 41, 52 and 66) to tie each member-function definition to the class template's scope. In this case, the generic class name is `Stack< T >`. When `doubleStack` is instantiated as type `Stack< double >`, the `Stack` constructor function-template specialization uses `new` to create an array of elements of type `double` to represent the stack (line 44). The statement

```
stackPtr = new T[size];
```

in the Stack class-template definition is generated by the compiler in the class-template specialization Stack< double > as

```
stackPtr = new double[size];
```

[Page 757]

## Creating a Driver to Test Class Template Stack< T >

Now, let us consider the driver (Fig. 14.3) that exercises the Stack class template. The driver begins by instantiating object doubleStack of size 5 (line 11). This object is declared to be of class Stack< double > (pronounced "Stack of double"). The compiler associates type double with type parameter T in the class template to produce the source code for a Stack class of type double. Although templates offer software-reusability benefits, remember that multiple class-template specializations are instantiated in a program (at compile time), even though the template is written only once.

**Figure 14.3. Class template Stack test program.**

(This item is displayed on pages 758 - 759 in the print version)

```

1 // Fig. 14.3: fig14_03.cpp
2 // Stack class template test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Stack.h" // Stack class template definition
8
9 int main()
10 {
11 Stack< double > doubleStack(5); // size 5
12 double doubleValue = 1.1;
13
14 cout << "Pushing elements onto doubleStack\n";
15
16 // push 5 doubles onto doubleStack
17 while (doubleStack.push(doubleValue))
18 {
19 cout << doubleValue << ' ';
20 doubleValue += 1.1;

```

```

21 } // end while
22
23 cout << "\nStack is full. Cannot push " << doubleValue
24 << "\n\nPopping elements from doubleStack\n";
25
26 // pop elements from doubleStack
27 while (doubleStack.pop(doubleValue))
28 cout << doubleValue << ' ';
29
30 cout << "\nStack is empty. Cannot pop\n";
31
32 Stack< int > intStack; // default size 10
33 int intValue = 1;
34 cout << "\nPushing elements onto intStack\n";
35
36 // push 10 integers onto intStack
37 while (intStack.push(intValue))
38 {
39 cout << intValue << ' ';
40 intValue++;
41 } // end while
42
43 cout << "\nStack is full. Cannot push " << intValue
44 << "\n\nPopping elements from intStack\n";
45
46 // pop elements from intStack
47 while (intStack.pop(intValue))
48 cout << intValue << ' ';
49
50 cout << "\nStack is empty. Cannot pop" << endl;
51 return 0;
52 } // end main

```

Pushing elements onto doubleStack  
 1.1 2.2 3.3 4.4 5.5  
 Stack is full. Cannot push 6.6

Popping elements from doubleStack  
 5.5 4.4 3.3 2.2 1.1  
 Stack is empty. Cannot pop

Pushing elements onto intStack  
 1 2 3 4 5 6 7 8 9 10  
 Stack is full. Cannot push 11

Popping elements from intStack  
 10 9 8 7 6 5 4 3 2 1

Stack is empty. Cannot pop

Lines 1721 invoke `push` to place the double values 1.1, 2.2, 3.3, 4.4 and 5.5 onto `doubleStack`. The while loop terminates when the driver attempts to push a sixth value onto `doubleStack` (which is full, because it holds a maximum of five elements). Note that function `push` returns `false` when it is unable to push a value onto the stack. [1]

[1] Class `Stack` (Fig. 14.2) provides the function `isFull`, which the programmer can use to determine whether the stack is full before attempting a push operation. This would avoid the potential error of pushing onto a full stack. In Chapter 16, Exception Handling, if the operation cannot be completed, function `push` would "throw an exception." The programmer can write code to "catch" that exception, then decide how to handle it appropriately for the application. The same technique can be used with function `pop` when an attempt is made to pop an element from an empty stack.

Lines 2728 invoke `pop` in a while loop to remove the five values from the stack (note, in Fig. 14.3, that the values do pop off in last-in, first-out order). When the driver attempts to pop a sixth value, the `doubleStack` is empty, so the `pop` loop terminates.

[Page 759]

Line 32 instantiates integer stack `intStack` with the declaration

```
Stack< int > intStack;
```

(pronounced "intStack is a Stack of int"). Because no size is specified, the size defaults to 10 as specified in the default constructor (Fig. 14.2, line 10). Lines 3741 loop and invoke `push` to place values onto `intStack` until it is full, then lines 4748 loop and invoke `pop` to remove values from `intStack` until it is empty. Once again, notice in the output that the values `pop` off in last-in, first-out order.

## Creating Function Templates to Test Class Template `Stack< T >`

Notice that the code in function `main` of Fig. 14.3 is almost identical for both the `doubleStack` manipulations in lines 1130 and the `intStack` manipulations in lines 3250. This presents another opportunity to use a function template. Figure 14.4 defines function template `testStack` (lines 1438) to perform the same tasks as `main` in Fig. 14.3 push a series of values onto a `Stack< T >` and `pop` the values off a `Stack< T >`. Function template `testStack` uses template parameter `T` (specified at line

14) to represent the data type stored in the `Stack< T >`. The function template takes four arguments (lines 1619) a reference to an object of type `Stack< T >`, a value of type `T` that will be the first value pushed onto the `Stack< T >`, a value of type `T` used to increment the values pushed onto the `Stack< T >` and a string that represents the name of the `Stack< T >` object for output purposes. Function `main` (lines 4049) instantiates an object of type `Stack< double >` called `doubleStack` (line 42) and an object of type `Stack< int >` called `intStack` (line 43) and uses these objects in lines 45 and 46. The `testStack` function calls each result in a `testStack` function-template specialization. The compiler infers the type of `T` for `testStack` from the type used to instantiate the function's first argument (i.e., the type used to instantiate `doubleStack` or `intStack`). The output of Fig. 14.4 precisely matches the output of Fig. 14.3.

[Page 761]

**Figure 14.4. Passing a stack template object to a function template.**

(This item is displayed on pages 759 - 760 in the print version)

```

1 // Fig. 14.4: fig14_04.cpp
2 // Stack class template test program. Function main uses a
3 // function template to manipulate objects of type Stack< T >.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9 using std::string;
10
11 #include "Stack.h" // Stack class template definition
12
13 // function template to manipulate Stack< T >
14 template< typename T >
15 void testStack(
16 Stack< T > &theStack, // reference to Stack< T >
17 T value, // initial value to push
18 T increment, // increment for subsequent values
19 const string stackName) // name of the Stack< T > object
20 {
21 cout << "\nPushing elements onto " << stackName >> '\n';
22
23 // push element onto Stack
24 while (theStack.push(value))
25 {
26 cout << value >> ' ';
27 value += increment;
28 } // end while

```

```

29
30 cout << "\nStack is full. Cannot push " << value
31 << "\n\nPopping elements from " << stackName << '\n';
32
33 // pop elements from Stack
34 while (theStack.pop(value))
35 cout << value << ' ';
36
37 cout << "\nStack is empty. Cannot pop" << endl;
38 } // end function template testStack
39
40 int main()
41 {
42 Stack< double > doubleStack(5); // size 5
43 Stack< int > intStack; // default size 10
44
45 testStack(doubleStack, 1.1, 1.1, "doubleStack");
46 testStack(intStack, 1, 1, "intStack");
47
48 return 0;
49 } // end main

```

Pushing elements onto doubleStack  
 1.1 2.2 3.3 4.4 5.5  
 Stack is full. Cannot push 6.6

Popping elements from doubleStack  
 5.5 4.4 3.3 2.2 1.1  
 Stack is empty. Cannot pop

Pushing elements onto intStack  
 1 2 3 4 5 6 7 8 9 10  
 Stack is full. Cannot push 11

Popping elements from intStack  
 10 9 8 7 6 5 4 3 2 1  
 Stack is empty. Cannot pop



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 762]

## Software Engineering Observation 14.3



Specifying the size of a container at compile time avoids the potentially fatal execution-time error if `new` is unable to obtain the needed memory.

In the exercises, you will be asked to use a nontype parameter to create a template for our class `Array` developed in [Chapter 11](#). This template will enable `Array` objects to be instantiated with a specified number of elements of a specified type at compile time, rather than creating space for the `Array` objects at execution time.

In some cases, it may not be possible to use a particular type with a class template. For example, the `Stack` template of [Fig. 14.2](#) requires that user-defined types that will be stored in a `Stack` must provide a default constructor and an assignment operator. If a particular user-defined type will not work with our `Stack` template or requires customized processing, you can define an **explicit specialization** of the class template for a particular type. Let's assume we want to create an explicit specialization `Stack` for `Employee` objects. To do this, form a new class with the name `Stack< Employee >` as follows:

```
template<>
class Stack< Employee >
{
 // body of class definition
};
```

Note that the `Stack< Employee >` explicit specialization is a complete replacement for the `Stack` class template that is specific to type `Employee`; it does not use anything from the original class template and can even have different members.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 762 (continued)]

## 14.6. Notes on Templates and Inheritance

Templates and inheritance relate in several ways:

- A class template can be derived from a class-template specialization.
- A class template can be derived from a nontemplate class.
- A class-template specialization can be derived from a class-template specialization.
- A nontemplate class can be derived from a class-template specialization.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 763]

```
friend void f1();
```

For example, function `f1` is a friend of `x< double >`, `x< string >` and `x< Employee >`, etc.

It is also possible to make a function `f2` a friend of only a class-template specialization with the same type argument. To do so, use a friendship declaration of the form

```
friend void f2(x< T > &);
```

For example, if `T` is a `float`, function `f2( x< float > & )` is a friend of class-template specialization `x< float >` but not a friend of class-template specification `x< string >`.

You can declare that a member function of another class is a friend of any class-template specialization generated from the class template. To do so, the `friend` declaration must qualify the name of the other class's member function using the class name and the binary scope resolution operator, as in:

```
friend void A::f3();
```

The declaration makes member function `f3` of class `A` a friend of every class-template specialization instantiated from the preceding class template. For example, function `f3` of class `A` is a friend of `x< double >`, `x< string >` and `x< Employee >`, etc.

As with a global function, another class's member function can be a friend of only a class-template specialization with the same type argument. A friendship declaration of the form

```
friend void C< T >::f4(x< T > &);
```

for a particular type `T` such as `float` makes member function

```
C< float >::f4(x< float > &)
```

a friend function of only class-template specialization `x< float >`.

In some cases, it is desirable to make an entire class's set of member functions friends of a class template. In this case, a friend declaration of the form

```
friend class Y;
```

makes every member function of class `Y` a friend of every class-template specialization produced from the class template `X`.

Finally, it is possible to make all member functions of one class-template specialization friends of another class-template specialization with the same type argument. For example, a friend declaration of the form:

```
friend class Z< T >;
```

indicates that when a class-template specialization is instantiated with a particular type for `T` (such as `float`), all members of class `Z< float >` become friends of class-template specialization `X< float >`. We use this particular relationship in several examples of [Chapter 21](#), Data Structures.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 764]

Each class-template specialization instantiated from a class template has its own copy of each `static` data member of the class template; all objects of that specialization share that one `static` data member. In addition, as with `static` data members of nontemplate classes, `static` data members of class-template specializations must be defined and, if necessary, initialized at file scope. Each class-template specialization gets its own copy of the class template's `static` member functions.

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 764 (continued)]

## 14.9. Wrap-Up

This chapter introduced one of C++'s most powerful features—templates. You learned how to use function templates to enable the compiler to produce a set of function-template specializations that represent a group of related overloaded functions. We also discussed how to overload a function template to create a specialized version of a function that handles a particular data type's processing in a manner that differs from the other function-template specializations. Next, you learned about class templates and class-template specializations. You saw examples of how to use a class template to create a group of related types that each perform identical processing on different data types. Finally, you learned about some of the relationships among templates, friends, inheritance and static members.

In the next chapter, we discuss many of C++'s I/O capabilities and demonstrate several stream manipulators that perform various formatting tasks.

[◀ PREV](#)[NEXT ▶](#)

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 765]

- The programmer who wishes to use class-template specializations writes one class template. When the programmer needs a new type-specific class, the programmer uses a concise notation, and the compiler writes the source code for the class-template specialization.
- A class-template definition looks like a conventional class definition, except that it is preceded by `template< typename T >` (or `template< class T >`) to indicate this is a class-template definition with type parameter `T` which acts as a placeholder for the type of the class to create. The type `T` is mentioned throughout the class header and member-function definitions as a generic type name.
- Member-function definitions outside a class template each begin with `template< typename T >` (or `template< class T >`). Then, each function definition resembles a conventional function definition, except that the generic data in the class always is listed generically as type parameter `T`. The binary scope-resolution operator is used with the class-template name to tie each member function definition to the class template's scope.
- It is possible to use nontype parameters in the header of a class or function template.
- An explicit specialization of a class template can be provided to override a class template for a specific type.
- A class template can be derived from a class-template specialization. A class template can be derived from a nontemplate class. A class-template specialization can be derived from a class-template specialization. A nontemplate class can be derived from a class-template specialization.
- Functions and entire classes can be declared as friends of nontemplate classes. With class templates, the obvious kinds of friendship arrangements can be declared. Friendship can be established between a class template and a global function, a member function of another class (possibly a class-template specialization) or even an entire class (possibly a class-template specialization).
- Each class-template specialization instantiated from a class template has its own copy of each `static` data member of the class template; all objects of that specialization share that `static` data member. And as with `static` data members of nontemplate classes, `static` data members of class-template specializations must be defined and, if necessary, initialized at file scope.
- Each class-template specialization gets a copy of the class template's `static` member functions.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 765 (continued)]

## Terminology

angle brackets (< and >)

class template

class-template definition

class-template specialization

explicit specialization

friend of a template

function template

function-template definition

function-template specialization

generic programming

keyword `class` in a template type parameter

keyword `template`

keyword `typename`

macro

member function of a class-template specialization

nontype parameter

nontype template parameter

overloading a function template

parameterized type

class-template specialization

`static` data member of a class template

`static` data member of a class-template specialization

`static` member function of a class template

`static` member function of a class-template specialization

template parameter

`template< class T >`

`template< typename T >`

`typename`

type parameter

type template parameter

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 766]

## Self-Review Exercises

- 14.1** State which of the following statements are true and which are false. If a statement is false, explain why.

a.

The template parameters of a function-template definition are used to specify the types of the arguments to the function, to specify the return type of the function and to declare variables within the function.

b.

Keywords `typename` and `class` as used with a template type parameter specifically mean "any user-defined class type."

c.

A function template can be overloaded by another function template with the same function name.

d.

Template parameter names among template definitions must be unique.

e.

Each member-function definition outside a class template must begin with a template header.

f.

A `friend` function of a class template must be a function-template specialization.

If several class-template specializations are generated from a single class template with a single `static` data member, each of the class-template specializations shares a single copy of the class template's `static` data member.

**14.2** Fill in the blanks in each of the following:

a.

Templates enable us to specify, with a single code segment, an entire range of related functions called \_\_\_\_\_, or an entire range of related classes called \_\_\_\_\_.

b.

All function-template definitions begin with the keyword \_\_\_\_\_, followed by a list of template parameters to the function template enclosed in \_\_\_\_\_.

c.

The related functions generated from a function template all have the same name, so the compiler uses \_\_\_\_\_ resolution to invoke the proper function.

d.

Class templates also are called \_\_\_\_\_ types.

e.

The \_\_\_\_\_ operator is used with a class-template name to tie each member-function definition to the class template's scope.

f.

As with `static` data members of nontemplate classes, `static` data members of class-template specializations must also be defined and, if necessary, initialized at \_\_\_\_\_ scope.

 PREV

NEXT 

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 766 (continued)]

## Answers to Self-Review Exercises

- 14.1** a) True. b) False. Keywords `typename` and `class` in this context also allow for a type parameter of a built-in type. c) True. d) False. Template parameter names among function templates need not be unique. e) True. f) False. It could be a nontemplate function. g) False. Each class-template specialization will have its own copy of the `static` data member.
- 14.2** a) function-template specializations, class-template specializations. b) `template`, angle brackets (`<` and `>`). c) overloading. d) parameterized. e) binary scope resolution. f) file.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 767]

## 14.5

Overload function template `printArray` of Fig. 14.1 with a nontemplate version that specifically prints an array of character strings in neat, tabular, column format.

## 14.6

Write a simple function template for predicate function `isEqualTo` that compares its two arguments of the same type with the equality operator (`==`) and returns `true` if they are equal and `false` if they are not equal. Use this function template in a program that calls `isEqualTo` only with a variety of built-in types. Now write a separate version of the program that calls `isEqualTo` with a user-defined class type, but does not overload the equality operator. What happens when you attempt to run this program? Now overload the equality operator (with the operator function) `operator==`. Now what happens when you attempt to run this program?

## 14.7

Use an `int` template nontype parameter `numberOfElements` and a type parameter `elementType` to help create a template for the `Array` class (Figs. 11.611.7) we developed in Chapter 11. This template will enable `Array` objects to be instantiated with a specified number of elements of a specified element type at compile time.

## 14.8

Write a program with class template `Array`. The template can instantiate an `Array` of any element type. Override the template with a specific definition for an `Array` of `float` elements (class `Array< float >`). The driver should demonstrate the instantiation of an `Array` of `int` through the template and should show that an attempt to instantiate an `Array` of `float` uses the definition provided in class `Array< float >`.

## 14.9

Distinguish between the terms "function template" and "function-template specialization."

## 14.10

Which is more like a stencil class template or a class-template specialization? Explain your answer.

## 14.11

What is the relationship between function templates and overloading?

**14.12**

Why might you choose to use a function template instead of a macro?

**14.13**

What performance problem can result from using function templates and class templates?

**14.14**

The compiler performs a matching process to determine which function-template specialization to call when a function is invoked. Under what circumstances does an attempt to make a match result in a compile error?

**14.15**

Why is it appropriate to refer to a class template as a parameterized type?

**14.16**

Explain why a C++ program would use the statement

```
Array< Employee > workerList(100);
```

**14.17**

Review your answer to [Exercise 14.16](#). Why might a C++ program use the statement

```
Array< Employee > workerList;
```

**14.18**

Explain the use of the following notation in a C++ program:

```
template< typename T > Array< T >::Array(int s)
```

**14.19**

Why might you use a nontype parameter with a class template for a container such as an array or stack?

---

[Page 768]

### 14.20

Describe how to provide an explicit specialization of a class template.

### 14.21

Describe the relationship between class templates and inheritance.

### 14.22

Suppose that a class template has the header

```
template< typename T > class Ct1
```

Describe the friendship relationships established by placing each of the following `friend` declarations inside this class template. Identifiers beginning with "f" are functions, identifiers beginning with "C" are classes, identifiers beginning with "Ct" are class templates and `T` is a template type parameter (i.e., `T` can represent any fundamental or class type).

a.

```
friend void f1();
```

b.

```
friend void f2(Ct1< T > &);
```

c.

```
friend void C2::f3();
```

d.

```
friend void Ct3< T >::f4(Ct1< T > &);
```

e.

```
friend class C4;
```

f.

```
friend class Ct5< T >;
```

## 14.23

Suppose that class template Employee has a static data member count. Suppose that three class-template specializations are instantiated from the class template. How many copies of the static data member will exist? How will the use of each be constrained (if at all)?

 PREV

page footer

NEXT 

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 770]

## Outline

### 15.1 Introduction

### 15.2 Streams

#### 15.2.1 Classic Streams vs. Standard Streams

#### 15.2.2 `iostream` Library Header Files

#### 15.2.3 Stream Input/Output Classes and Objects

### 15.3 Stream Output

#### 15.3.1 Output of `char * Variables`

#### 15.3.2 Character Output using Member Function `put`

### 15.4 Stream Input

#### 15.4.1 `get` and `getline` Member Functions

#### 15.4.2 `istream` Member Functions `peek`, `putback` and `ignore`

#### 15.4.3 Type-Safe I/O

### 15.5 Unformatted I/O using `read`, `write` and `gcount`

### 15.6 Introduction to Stream Manipulators

#### 15.6.1 Integral Stream Base: `dec`, `oct`, `hex` and `setbase`

#### 15.6.2 Floating-Point Precision (`precision`, `setprecision`)

#### 15.6.3 Field Width (`width`, `setw`)

#### 15.6.4 User-Defined Output Stream Manipulators

### 15.7 Stream Format States and Stream Manipulators

#### 15.7.1 Trailing Zeros and Decimal Points (`showpoint`)

#### 15.7.2 Justification (`left`, `right` and `internal`)

#### 15.7.3 Padding (`fill`, `setfill`)

#### 15.7.4 Integral Stream Base (`dec`, `oct`, `hex`, `showbase`)

#### 15.7.5 Floating-Point Numbers; Scientific and Fixed Notation (`scientific`, `fixed`)

#### 15.7.6 Uppercase/Lowercase Control (`uppercase`)

#### 15.7.7 Specifying Boolean Format (`boolalpha`)

#### 15.7.8 Setting and Resetting the Format State via Member-Function `flags`

### 15.8 Stream Error States

### 15.9 Tying an Output Stream to an Input Stream

### 15.10 Wrap-Up

### Summary

### Terminology

### Self-Review Exercises

### Answers to Self-Review Exercises

### Exercises

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 771]

C++ uses **type-safe I/O**. Each I/O operation is executed in a manner sensitive to the data type. If an I/O member function has been defined to handle a particular data type, then that member function is called to handle that data type. If there is no match between the type of the actual data and a function for handling that data type, the compiler generates an error. Thus, improper data cannot "sneak" through the system (as can occur in C, allowing for some subtle and bizarre errors).

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 (>> ). This **extensibility** is one of C++'s most valuable features.

### Software Engineering Observation 15.1



Use the C++-style I/O exclusively in C++ programs, even though C-style I/O is available to C++ programmers.

### Error-Prevention Tip 15.1



C++ I/O is type safe.

### Software Engineering Observation 15.2



C++ enables a common treatment of I/O for *predefined* types and user-defined types. This commonality facilitates software development and reuse.



page footer



The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 772]

## Performance Tip 15.1



Use unformatted I/O for the best performance in high-volume file processing.

## Portability Tip 15.1



Using unformatted I/O can lead to portability problems, because unformatted data is not portable across all platforms.

### 15.2.1. Classic Streams vs. Standard Streams

In the past, the C++ [classic stream libraries](#) enabled input and output of `char`s. Because a `char` occupies one byte, it can represent only a limited set of characters (such as those in the ASCII character set). However, many languages use alphabets that contain more characters than a single-byte `char` can represent. The ASCII character set does not provide these characters; the [Unicode character set](#) does. Unicode is an extensive international character set that represents the majority of the world's commercially viable languages, mathematical symbols and much more. For more information on Unicode, visit [www.unicode.org](http://www.unicode.org).

C++ includes the [standard stream libraries](#), which enable developers to build systems capable of performing I/O operations with Unicode characters. For this purpose, C++ includes an additional character type called `wchar_t`, which can store Unicode characters. The C++ standard also redesigned the classic C++ stream classes, which processed only `char`s, as class templates with separate specializations for processing characters of types `char` and `wchar_t`, respectively. We use the `char` type of class templates with separate specializations throughout this book.

### 15.2.2. `<iostream>` Library Header Files

The C++ `iostream` library provides hundreds of I/O capabilities. Several header files contain portions of the library interface.

Most C++ programs include the `<iostream>` header file, which declares basic services required for all

stream-I/O operations. The `<iostream>` header file defines the `cin`, `cout`, `cerr` and `clog` objects, which correspond to the standard input stream, the standard output stream, the unbuffered standard error stream and the buffered standard error stream, respectively. (`cerr` and `clog` are discussed in [Section 15.2.3.](#)) Both unformatted-and formatted-I/O services are provided.

The `<iomanip>` header declares services useful for performing formatted I/O with so-called **parameterized stream manipulators**, such as `setw` and `setprecision`.

The `<fstream>` header declares services for user-controlled file processing. We use this header in the file-processing programs of [Chapter 17](#).

C++ implementations generally contain other I/O-related libraries that provide system-specific capabilities, such as the controlling of special-purpose devices for audio and video I/O.

### 15.2.3. Stream Input/Output Classes and Objects

The `iostream` library provides many templates for handling common I/O operations. For example, class template `basic_istream` supports stream-input operations, class template `basic_ostream` supports stream-output operations, and class template `basic_iostream` supports both stream-input and stream-output operations. Each template has a predefined template specialization that enables `char` I/O. In addition, the `iostream` library provides a set of `typedefs` that provide aliases for these template specializations. The `typedef` specifier declares synonyms (aliases) for previously defined data types. Programmers sometimes use `typedef` to create shorter or more readable type names. For example, the statement

[Page 773]

```
typedef Card *CardPtr;
```

defines an additional type name, `CardPtr`, as a synonym for type `Card *`. Note that creating a name using `typedef` does not create a data type; `typedef` creates only a type name that may be used in the program. [Section 22.5](#) `typedef` discusses `typedef` in detail. The `typedef istream` represents a specialization of `basic_istream` that enables `char` input. Similarly, the `typedef ostream` represents a specialization of `basic_ostream` that enables `char` output. Also, the `typedef iostream` represents a specialization of `basic_iostream` that enables both `char` input and output. We use these `typedefs` throughout this chapter.

### Stream-I/O Template Hierarchy and Operator Overloading

Templates `basic_istream` and `basic_ostream` both derive through single inheritance from base template `basic_ios`. <sup>[1]</sup> Template `basic_iostream` derives through multiple inheritance <sup>[2]</sup> from

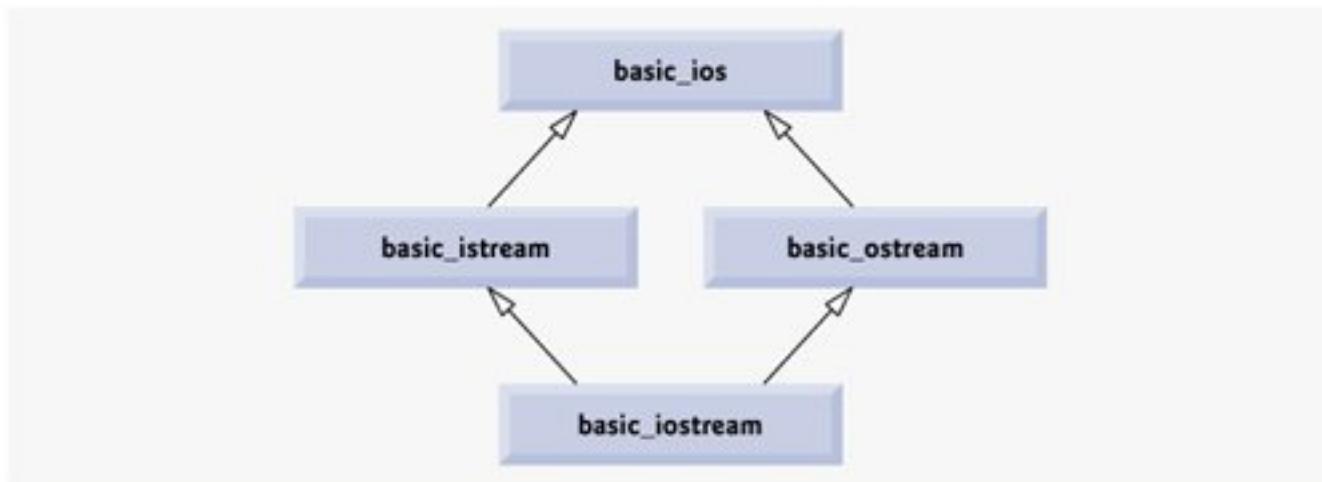
templates `basic_istream` and `basic_ostream`. The UML class diagram of Fig. 15.1 summarizes these inheritance relationships.

[1] Technically, templates do not inherit from other templates. However, in this chapter, we discuss templates only in the context of the template specializations that enable `char` I/O. These specializations are classes and thus can inherit from each other.

[2] Multiple inheritance is discussed in [Chapter 24](#), Other Topics.

**Figure 15.1. Stream-I/O template hierarchy portion.**

[\[View full size image\]](#)



Operator overloading provides a convenient notation for performing input/output. The left-shift operator (`<<`) is overloaded to designate stream output and is referred to as the stream insertion operator. The right-shift operator (`>>`) is overloaded to designate stream input and is referred to as the stream extraction operator. These operators are used with the standard stream objects `cin`, `cout`, `cerr` and `clog` and, commonly, with user-defined stream objects.

---

[Page 774]

### Standard Stream Objects `cin`, `cout`, `cerr` and `clog`

The predefined object `cin` is an `istream` instance and is said to be "connected to" (or attached to) the standard input device, which usually is the keyboard. The stream extraction operator (`>>`) as used in the following statement causes a value for integer variable `grade` (assuming that `grade` has been declared as an `int` variable) to be input from `cin` to memory:

```
cin >> grade; // data "flows" in the direction of the arrows
```

Note that the compiler determines the data type of grade and selects the appropriate overloaded stream extraction operator. Assuming that grade has been declared properly, the stream extraction operator does not require additional type information (as is the case, for example, in C-style I/O). The `>>` operator is overloaded to input data items of built-in types, strings and pointer values.

The predefined object `cout` is an `ostream` instance and is said to be "connected to" the standard output device, which usually is the display screen. The stream insertion operator (`<<`), as used in the following statement, causes the value of variable grade to be output from memory to the standard output device:

```
cout << grade; // data "flows" in the direction of the arrows
```

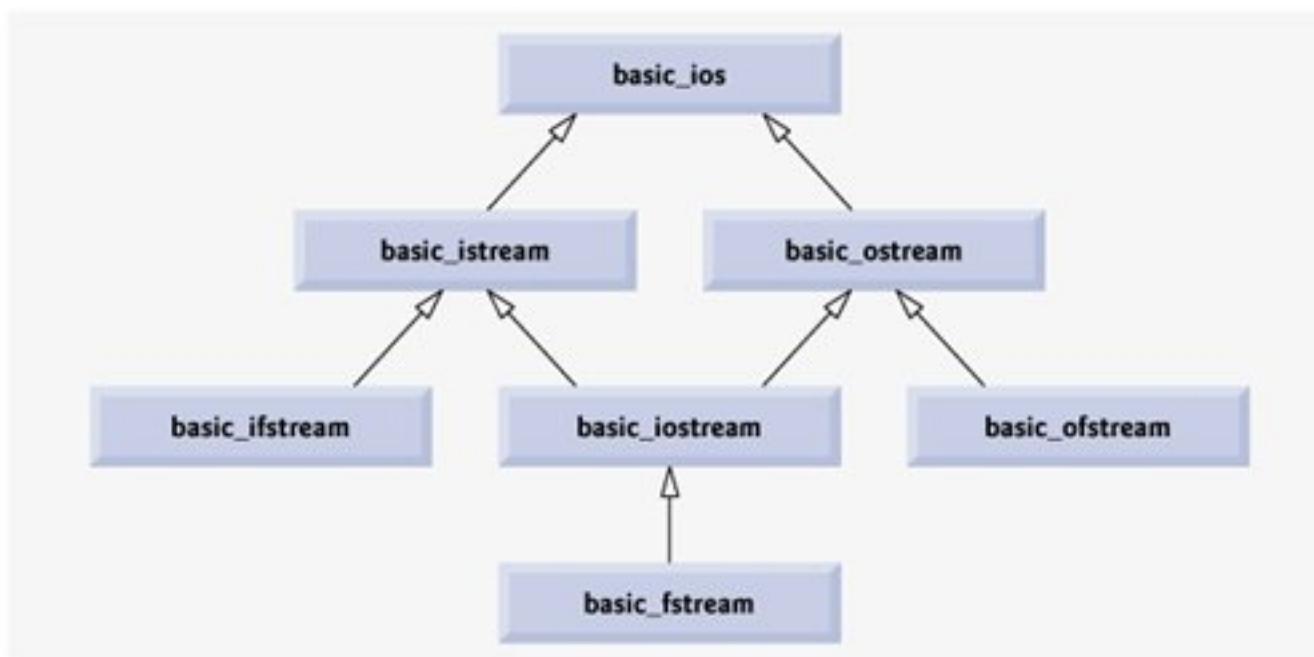
Note that the compiler also determines the data type of grade (assuming grade has been declared properly) and selects the appropriate stream insertion operator, so the stream insertion operator does not require additional type information. The `<<` operator is overloaded to output data items of built-in types, strings and pointer values.

The predefined object `cerr` is an `ostream` instance and is said to be "connected to" the standard error device. Outputs to object `cerr` are **unbuffered**, implying that each stream insertion to `cerr` causes its output to appear immediatelythis is appropriate for notifying a user promptly about errors.

The predefined object `clog` is an instance of the `ostream` class and is said to be "connected to" the standard error device. Outputs to `clog` are **buffered**. This means that each insertion to `clog` could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed. Buffering is an I/O performance-enhancement technique discussed in operating-systems courses.

## File-Processing Templates

C++ file processing uses class templates `basic_ifstream` (for file input), `basic_ofstream` (for file output) and `basic_fstream` (for file input and output). Each class template has a predefined template specialization that enables `char` I/O. C++ provides a set of `typedefs` that provide aliases for these template specializations. For example, the `typedef ifstream` represents a specialization of `basic_ifstream` that enables `char` input from a file. Similarly, `typedef ofstream` represents a specialization of `basic_ofstream` that enables `char` output to a file. Also, `typedef fstream` represents a specialization of `basic_fstream` that enables `char` input from, and output to, a file. Template `basic_ifstream` inherits from `basic_istream`, `basic_ofstream` inherits from `basic_ostream` and `basic_fstream` inherits from `basic_iostream`. The UML class diagram of Fig. 15.2 summarizes the various inheritance relationships of the I/O-related classes. The full stream-I/O class hierarchy provides most of the capabilities that programmers need. Consult the class-library reference for your C++ system for additional file-processing information.

**Figure 15.2. Stream-I/O template hierarchy portion showing the main file-processing templates.**[\[View full size image\]](#)[◀ PREV](#)[NEXT ▶](#)

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 776]

displays a single character A. Calls to `put` may be cascaded, as in the statement

```
cout.put('A').put('\n');
```

which outputs the letter A followed by a newline character. As with `<<`, the preceding statement executes in this manner, because the dot operator (.) evaluates from left to right, and the `put` member function returns a reference to the `ostream` object (`cout`) that received the `put` call. The `put` function also may be called with a numeric expression that represents an ASCII value, as in the following statement

```
cout.put(65);
```

which also outputs A.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 777]

Each stream object contains a set of **state bits** used to control the state of the stream (i.e., formatting, setting error states, etc.). These bits are used by the stream's overloaded `void *` cast operator to determine whether to return a non-null pointer or the null pointer. Stream extraction causes the stream's **failbit** to be set if data of the wrong type is input and causes the stream's **badbit** to be set if the operation fails. [Section 15.7](#) and [Section 15.8](#) discuss stream state bits in detail, then show how to test these bits after an I/O operation.

#### 15.4.1. **get** and **getline** Member Functions

The **get** member function with no arguments inputs one character from the designated stream (including white-space characters and other non-graphic characters, such as the key sequence that represents end-of-file) and returns it as the value of the function call. This version of **get** returns **EOF** when end-of-file is encountered on the stream.

#### Using Member Functions **eof**, **get** and **put**

[Figure 15.4](#) demonstrates the use of member functions **eof** and **get** on input stream **cin** and member function **put** on output stream **cout**. The program first prints the value of `cin.eof()` i.e., `false` (0 on the output) to show that end-of-file has not occurred on **cin**. The user enters a line of text and presses Enter followed by end-of-file (`<ctrl>-z` on Microsoft Windows systems, `<ctrl>-d` on UNIX and Macintosh systems). Line 17 reads each character, which line 18 outputs to **cout** using member function **put**. When end-of-file is encountered, the `while` statement ends, and line 22 displays the value of `cin.eof()`, which is now `True` (1 on the output), to show that end-of-file has been set on **cin**. Note that this program uses the version of `istream` member function **get** that takes no arguments and returns the character being input (line 17). Function **eof** returns `true` only after the program attempts to read past the last character in the stream.

[Page 778]

#### Figure 15.4. **get**, **put** and **eof** member functions.

(This item is displayed on pages 777 - 778 in the print version)

```

1 // Fig. 15.4: Fig15_04.cpp
2 // Using member functions get, put and eof.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10 int character; // use int, because char cannot represent EOF
11
12 // prompt user to enter line of text
13 cout << "Before input, cin.eof() is " << cin.eof() << endl
14 << "Enter a sentence followed by end-of-file:" << endl;
15
16 // use get to read each character; use put to display it
17 while ((character = cin.get()) != EOF)
18 cout.put(character);
19
20 // display end-of-file character
21 cout << "\nEOF in this system is: " << character << endl;
22 cout << "After input of EOF, cin.eof() is " << cin.eof() << endl;
23 return 0;
24 } // end main

```

Before input, cin.eof() is 0  
 Enter a sentence followed by end-of-file:  
 Testing the get and put member functions  
 Testing the get and put member functions  
 ^Z

EOF in this system is: -1  
 After input of EOF, cin.eof() is 1

The `get` member function with a character-reference argument inputs the next character from the input stream (even if this is a white-space character) and stores it in the character argument. This version of `get` returns a reference to the `istream` object for which the `get` member function is being invoked.

A third version of `get` takes three argumentsa character array, a size limit and a delimiter (with default value '`\n`'). This version reads characters from the input stream. It either reads one fewer than the

specified maximum number of characters and terminates or terminates as soon as the delimiter is read. A null character is inserted to terminate the input string in the character array used as a buffer by the program. The delimiter is not placed in the character array but does remain in the input stream (the delimiter will be the next character read). Thus, the result of a second consecutive get is an empty line, unless the delimiter character is removed from the input stream (possibly with `cin.ignore()`).

## Comparing `cin` and `cin.get`

**Figure 15.5** compares input using stream extraction with `cin` (which reads characters until a white-space character is encountered) and input using `cin.get`. Note that the call to `cin.get` (line 24) does not specify a delimiter, so the default '\n' character is used.

**Figure 15.5. Input of a string using `cin` with stream extraction contrasted with input using `cin.get`.**

(This item is displayed on pages 778 - 779 in the print version)

```

1 // Fig. 15.5: Fig15_05.cpp
2 // Contrasting input of a string via cin and cin.get.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10 // create two char arrays, each with 80 elements
11 const int SIZE = 80;
12 char buffer1[SIZE];
13 char buffer2[SIZE];
14
15 // use cin to input characters into buffer1
16 cout << "Enter a sentence:" << endl;
17 cin >> buffer1;
18
19 // display buffer1 contents
20 cout << "\nThe string read with cin was:" << endl
21 << buffer1 << endl << endl;
22
23 // use cin.get to input characters into buffer2
24 cin.get(buffer2, SIZE);
25
26 // display buffer2 contents
27 cout << "The string read with cin.get was:" << endl
28 << buffer2 << endl;
29 return 0;

```

```
30 } // end main
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The string read with cin was:
Contrasting

The string read with cin.get was:
string input with cin and cin.get
```

[Page 779]

## Using Member Function `getline`

Member function `getline` operates similarly to the third version of the `get` member function and inserts a null character after the line in the character array. The `getline` function removes the delimiter from the stream (i.e., reads the character and discards it), but does not store it in the character array. The program of Fig. 15.6 demonstrates the use of the `getline` member function to input a line of text (line 15).

**Figure 15.6. Inputting character data with `cin` member function `getline`.**

(This item is displayed on pages 779 - 780 in the print version)

```
1 // Fig. 15.6: Fig15_06.cpp
2 // Inputting characters using cin member function getline.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10 const int SIZE = 80;
11 char buffer[SIZE]; // create array of 80 characters
12
13 // input characters in buffer via cin function getline
```

```

14 cout << "Enter a sentence:" << endl;
15 cin.getline(buffer, SIZE);
16
17 // display buffer contents
18 cout << "\nThe sentence entered is:" << endl << buffer << endl;
19 return 0;
20 } // end main

```

Enter a sentence:  
Using the getline member function  
  
The sentence entered is:  
Using the getline member function

[Page 780]

### 15.4.2. `istream` Member Functions `peek`, `putback` and `ignore`

The `ignore` member function of `istream` either reads and discards a designated number of characters (the default is one character) or terminates upon encountering a designated delimiter (the default delimiter is `EOF`, which causes `ignore` to skip to the end of the file when reading from a file).

The `putback` member function places the previous character obtained by a `get` from an input stream back into that stream. This function is useful for applications that scan an input stream looking for a field beginning with a specific character. When that character is input, the application returns the character to the stream, so the character can be included in the input data.

The `peek` member function returns the next character from an input stream but does not remove the character from the stream.

### 15.4.3. Type-Safe I/O

C++ offers type-safe I/O. The `<<` and `>>` operators are overloaded to accept data items of specific types. If unexpected data is processed, various error bits are set, which the user may test to determine whether an I/O operation succeeded or failed. If operator `<<` has not been overloaded for a user-defined type and you attempt to input into or output the contents of an object of that user-defined type, the compiler reports an error. This enables the program to "stay in control." We discuss these error states in Section 15.8.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 781]

outputs the first 10 bytes of `buffer` (including null characters, if any, that would cause output with `cout` and `<<` to terminate). The call

```
cout.write("ABCDEFGHIJKLMNOPQRSTUVWXYZ" , 10);
```

displays the first 10 characters of the alphabet.

The `read` member function inputs a designated number of characters into a character array. If fewer than the designated number of characters are read, `failbit` is set. [Section 15.8](#) shows how to determine whether `failbit` has been set. Member function `gcount` reports the number of characters read by the last input operation.

[Figure 15.7](#) demonstrates `istream` member functions `read` and `gcount` and `ostream` member function `write`. The program inputs 20 characters (from a longer input sequence) into character array `buffer` with `read` (line 15), determines the number of characters input with `gcount` (line 19) and outputs the characters in `buffer` with `write` (line 19).

### Figure 15.7. Unformatted I/O using the `read`, `gcount` and `write` member functions.

```

1 // Fig. 15.7: Fig15_07.cpp
2 // Unformatted I/O using read, gcount and write.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10 const int SIZE = 80;
11 char buffer[SIZE]; // create array of 80 characters
12
13 // use function read to input characters into buffer
14 cout << "Enter a sentence:" << endl;
15 cin.read(buffer, 20);
16
17 // use functions write and gcount to display buffer characters
18 cout << endl << "The sentence entered was:" << endl;
19 cout.write(buffer, cin.gcount());

```

```
20 cout << endl;
21 return 0;
22 } // end main
```

Enter a sentence:

Using the read, write, and gcount member functions

The sentence entered was:

Using the read, writ

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 782]

### 15.6.1. Integral Stream Base: `dec`, `oct`, `hex` and `setbase`

Integers are interpreted normally as decimal (base-10) values. To change the base in which integers are interpreted on a stream, insert the `hex` manipulator to set the base to hexadecimal (base 16) or insert the `oct` manipulator to set the base to octal (base 8). Insert the `dec` manipulator to reset the stream base to decimal.

The base of a stream also may be changed by the `setbase` stream manipulator, which takes one integer argument of 10, 8, or 16 to set the base to decimal, octal or hexadecimal, respectively. Because `setbase` takes an argument, it is called a parameterized stream manipulator. Using `setbase` (or any other parameterized manipulator) requires the inclusion of the `<iomanip>` header file. The stream base value remains the same until changed explicitly; `setbase` settings are "sticky." [Figure 15.8](#) demonstrates stream manipulators `hex`, `oct`, `dec` and `setbase`.

**Figure 15.8. Stream manipulators `hex`, `oct`, `dec` and `setbase`.**

(This item is displayed on pages 782 - 783 in the print version)

```
1 // Fig. 15.8: Fig15_08.cpp
2 // Using stream manipulators hex, oct, dec and setbase.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::dec;
7 using std::endl;
8 using std::hex;
9 using std::oct;
10
11 #include <iomanip>
12 using std::setbase;
13
14 int main()
15 {
16 int number;
17
18 cout << "Enter a decimal number: ";
19 cin >> number; // input number
20
21 // use hex stream manipulator to show hexadecimal number
```

```

22 cout << number << " in hexadecimal is: " << hex
23 << number << endl;
24
25 // use oct stream manipulator to show octal number
26 cout << dec << number << " in octal is: "
27 << oct << number << endl;
28
29 // use setbase stream manipulator to show decimal number
30 cout << setbase(10) << number << " in decimal is: "
31 << number << endl;
32 return 0;
33 } // end main

```

```

Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20

```

[Page 783]

### 15.6.2. Floating-Point Precision (`precision`, `setprecision`)

We can control the `precision` of floating-point numbers (i.e., the number of digits to the right of the decimal point) by using either the `setprecision` stream manipulator or the `precision` member function of `ios_base`. A call to either of these sets the precision for all subsequent output operations until the next precision-setting call. A call to member function `precision` with no argument returns the current precision setting (this is what you need to use so that you can restore the original precision eventually after a "sticky" setting is no longer needed). The program of Fig. 15.9 uses both member function `precision` (line 28) and the `setprecision` manipulator (line 37) to print a table that shows the square root of 2, with precision varying from 09.

**Figure 15.9. Precision of floating-point values.**

(This item is displayed on pages 783 - 784 in the print version)

```
1 // Fig. 15.9: Fig15_09.cpp
2 // Controlling precision of floating-point values.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include <cmath>
12 using std::sqrt; // sqrt prototype
13
14 int main()
15 {
16 double root2 = sqrt(2.0); // calculate square root of 2
17 int places; // precision, vary from 0-9
18
19 cout << "Square root of 2 with precisions 0-9." << endl
20 << "Precision set by ios_base member function "
21 << "precision:" << endl;
22
23 cout << fixed; // use fixed-point notation
24
25 // display square root using ios_base function precision
26 for (places = 0; places <= 9; places++)
27 {
28 cout.precision(places);
29 cout << root2 << endl;
30 } // end for
31
32 cout << "\nPrecision set by stream manipulator "
33 << "setprecision:" << endl;
34
35 // set precision for each digit, then display square root
36 for (places = 0; places <= 9; places++)
37 cout << setprecision(places) << root2 << endl;
38
39 return 0;
40 } // end main
```

```
Square root of 2 with precisions 0-9.
Precision set by ios_base member function precision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Precision set by stream manipulator setprecision:

```
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

[Page 784]

### 15.6.3. Field Width (`width`, `setw`)

The `width` member function (of base class `ios_base`) sets the field width (i.e., the number of character positions in which a value should be output or the maximum number of characters that should be input) and returns the previous width. If values output are narrower than the field width, **fill characters** are inserted as **padding**. A value wider than the designated width will not be truncatedthe full number will be printed. The `width` function with no argument returns the current setting.

Common Programming Error 15.1



The width setting applies only for the next insertion or extraction (i.e., the width setting is not "sticky"); afterward, the width is set implicitly to 0 (i.e., input and output will be performed with default settings). Assuming that the width setting applies to all subsequent outputs is a logic error.

---

[Page 785]

## Common Programming Error 15.2



When a field is not sufficiently wide to handle outputs, the outputs print as wide as necessary, which can yield confusing outputs.

**Figure 15.10** demonstrates the use of the `width` member function on both input and output. Note that, on input into a `char` array, a maximum of one fewer characters than the width will be read, because provision is made for the null character to be placed in the input string. Remember that stream extraction terminates when nonleading white space is encountered. The `setw` stream manipulator also may be used to set the field width.

**Figure 15.10. `width` member function of class `ios_base`.**

```

1 // Fig. 15.10: Fig15_10.cpp
2 // Demonstrating member function width.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10 int widthValue = 4;
11 char sentence[10];
12
13 cout << "Enter a sentence:" << endl;
14 cin.width(5); // input only 5 characters from sentence
15
16 // set field width, then display characters based on that width
17 while (cin >> sentence)
18 {
19 cout.width(widthValue++);
20 cout << sentence << endl;
21 cin.width(5); // input 5 more characters from sentence

```

```

22 } // end while
23
24 return 0;
25 } // end main

```

Enter a sentence:

```

This is a test of the width member function
This
 is
 a
 test
 of
 the
 width
 h
 memb
 er
 func
 tion

```

[Page 786]

[Note: When prompted for input in Fig. 15.10, the user should enter a line of text and press Enter followed by end-of-file ( <ctrl>-z on Microsoft Windows systems, <ctrl>-d on UNIX and Macintosh systems).]

#### 15.6.4. User-Defined Output Stream Manipulators

Programmers can create their own stream manipulators.<sup>[3]</sup> Figure 15.11 shows the creation and use of new nonparameterized stream manipulators `bell` (lines 1013), `carriageReturn` (lines 1619), `tab` (lines 2225) and `endLine` (lines 2932). For output stream manipulators, the return type and parameter must be of type `ostream &`. When line 37 inserts the `endLine` manipulator in the output stream, function `endLine` is called and line 31 outputs the escape sequence `\n` and the `flush` manipulator to the standard output stream `cout`. Similarly, when lines 3746 insert the manipulators `tab`, `bell` and `carriageReturn` in the output stream, their corresponding functions `tab` (line 22), `bell` (line 10) and `carriageReturn` (line 16) are called, which in turn output various escape sequences.

<sup>[3]</sup> Programmers also may create their own parameterized stream manipulators consult your

C++ compiler's documentation for instructions on how to do this.

[Page 787]

**Figure 15.11. User-defined, nonparameterized stream manipulators.**

(This item is displayed on pages 786 - 787 in the print version)

```
1 // Fig. 15.11: Fig15_11.cpp
2 // Creating and testing user-defined, nonparameterized
3 // stream manipulators.
4 #include <iostream>
5 using std::cout;
6 using std::flush;
7 using std::ostream;
8
9 // bell manipulator (using escape sequence \a)
10 ostream& bell(ostream& output)
11 {
12 return output << '\a'; // issue system beep
13 } // end bell manipulator
14
15 // carriageReturn manipulator (using escape sequence \r)
16 ostream& carriageReturn(ostream& output)
17 {
18 return output << '\r'; // issue carriage return
19 } // end carriageReturn manipulator
20
21 // tab manipulator (using escape sequence \t)
22 ostream& tab(ostream& output)
23 {
24 return output << '\t'; // issue tab
25 } // end tab manipulator
26
27 // endLine manipulator (using escape sequence \n and member
28 // function flush)
29 ostream& endLine(ostream& output)
30 {
31 return output << '\n' << flush; // issue endl-like end of line
32 } // end endLine manipulator
33
34 int main()
35 {
36 // use tab and endLine manipulators
37 cout << "Testing the tab manipulator:" << endLine
38 << 'a' << tab << 'b' << tab << 'c' << endLine;
```

```
39 cout << "Testing the carriageReturn and bell manipulators:"
40 << endl << ".....";
41
42 cout << bell; // use bell manipulator
43
44 // use carriageReturn and endl manipulators
45 cout << carriageReturn << "----" << endl;
46 return 0;
47 } // end main
```

Testing the tab manipulator:

a        b        c

Testing the carriageReturn and bell manipulators:

----.....

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 788]

## Stream Manipulator

### Description

`skipws`

Skip white-space characters on an input stream. This setting is reset with stream manipulator `noskipws`.

`left`

Left justify output in a field. Padding characters appear to the right if necessary.

`right`

Right justify output in a field. Padding characters appear to the left if necessary.

`internal`

Indicate that a number's sign should be left justified in a field and a number's magnitude should be right justified in that same field (i.e., padding characters appear between the sign and the number).

`dec`

Specify that integers should be treated as decimal (base 10) values.

`oct`

Specify that integers should be treated as octal (base 8) values.

`hex`

Specify that integers should be treated as hexadecimal (base 16) values.

`showbase`

Specify that the base of a number is to be output ahead of the number (a leading `0` for octals; a leading `0x` or `0X` for hexadecimals). This setting is reset with stream manipulator `noshowbase`.

**showpoint**

Specify that floating-point numbers should be output with a decimal point. This is used normally with `fixed` to guarantee a certain number of digits to the right of the decimal point, even if they are zeros. This setting is reset with stream manipulator `noshowpoint`.

**uppercase**

Specify that uppercase letters (i.e., `x` and `A` through `F`) should be used in a hexadecimal integer and that uppercase `E` should be used when representing a floating-point value in scientific notation. This setting is reset with stream manipulator `nouppercase`.

**showpos**

Specify that positive numbers should be preceded by a plus sign (+). This setting is reset with stream manipulator `noshowpos`.

**scientific**

Specify output of a floating-point value in scientific notation.

**fixed**

Specify output of a floating-point value in fixed-point notation with a specific number of digits to the right of the decimal point.

[Page 788]

### 15.7.1. Trailing Zeros and Decimal Points (`showpoint`)

Stream manipulator `showpoint` forces a floating-point number to be output with its decimal point and trailing zeros. For example, the floating-point value `79.0` prints as `79` without using `showpoint` and prints as `79.000000` (or as many trailing zeros as are specified by the current precision) using `showpoint`. To reset the `showpoint` setting, output the stream manipulator `noshowpoint`. The program in Fig. 15.13 shows how to use stream manipulator `showpoint` to control the printing of trailing zeros and decimal points for floating-point values. Recall that the default precision of a floating-point number is 6. When neither the `fixed` nor the `scientific` stream manipulator is used, the precision represents the number of significant digits to display (i.e., the total number of digits to display), not the number of digits to display after decimal point.

#### Figure 15.13. Controlling the printing of trailing zeros and decimal points in floating-point

**values.**

(This item is displayed on pages 788 - 789 in the print version)

```

1 // Fig. 15.13: Fig15_13.cpp
2 // Using showpoint to control the printing of
3 // trailing zeros and decimal points for doubles.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::showpoint;
8
9 int main()
10 {
11 // display double values with default stream format
12 cout << "Before using showpoint" << endl
13 << "9.9900 prints as: " << 9.9900 << endl
14 << "9.9000 prints as: " << 9.9000 << endl
15 << "9.0000 prints as: " << 9.0000 << endl << endl;
16
17 // display double value after showpoint
18 cout << showpoint
19 << "After using showpoint" << endl
20 << "9.9900 prints as: " << 9.9900 << endl
21 << "9.9000 prints as: " << 9.9000 << endl
22 << "9.0000 prints as: " << 9.0000 << endl;
23 return 0;
24 } // end main

```

Before using showpoint  
 9.9900 prints as: 9.99  
 9.9000 prints as: 9.9  
 9.0000 prints as: 9

After using showpoint  
 9.9900 prints as: 9.99000  
 9.9000 prints as: 9.90000  
 9.0000 prints as: 9.00000

### 15.7.2. Justification (`left`, `right` and `internal`)

Stream manipulators `left` and `right` enable fields to be left justified with padding characters to the right or right justified with padding characters to the left, respectively. The padding character is specified by the `fill` member function or the `setfill` parameterized stream manipulator (which we discuss in Section 15.7.3). Figure 15.14 uses the `setw`, `left` and `right` manipulators to left justify and right justify integer data in a field.

**Figure 15.14. Left justification and right justification with stream manipulators `left` and `right`.**

(This item is displayed on pages 789 - 790 in the print version)

```
1 // Fig. 15.14: Fig15_14.cpp
2 // Demonstrating left justification and right justification.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::left;
7 using std::right;
8
9 #include <iomanip>
10 using std::setw;
11
12 int main()
13 {
14 int x = 12345;
15
16 // display x right justified (default)
17 cout << "Default is right justified:" << endl
18 << setw(10) << x;
19
20 // use left manipulator to display x left justified
21 cout << "\n\nUse std::left to left justify x:\n"
22 << left << setw(10) << x;
23
24 // use right manipulator to display x right justified
25 cout << "\n\nUse std::right to right justify x:\n"
26 << right << setw(10) << x << endl;
27 return 0;
28 } // end main
```

Default is right justified:

```
12345
```

Use std::left to left justify x:

```
12345
```

Use std::right to right justify x:

```
12345
```

[Page 790]

Stream manipulator `internal` indicates that a number's sign (or base when using stream manipulator `showbase`) should be left justified within a field, that the number's magnitude should be right justified and that intervening spaces should be padded with the fill character. Figure 15.15 shows the internal stream manipulator specifying internal spacing (line 15). Note that `showpos` forces the plus sign to print (line 15). To reset the `showpos` setting, output the stream manipulator `noshowpos`.

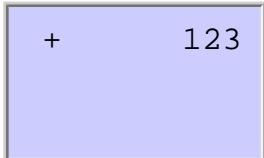
**Figure 15.15. Printing an integer with internal spacing and plus sign.**

(This item is displayed on pages 790 - 791 in the print version)

```

1 // Fig. 15.15: Fig15_15.cpp
2 // Printing an integer with internal spacing and plus sign.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::internal;
7 using std::showpos;
8
9 #include <iomanip>
10 using std::setw;
11
12 int main()
13 {
14 // display value with internal spacing and plus sign
15 cout << internal << showpos << setw(10) << 123 << endl;
16 return 0;
17 } // end main

```



[Page 791]

### 15.7.3. Padding (`fill`, `setfill`)

The `fill member function` specifies the fill character to be used with justified fields; if no value is specified, spaces are used for padding. The `fill` function returns the prior padding character. The `setfill manipulator` also sets the padding character. [Figure 15.16](#) demonstrates using member function `fill` (line 40) and stream manipulator `setfill` (lines 44 and 47) to set the fill character.

**Figure 15.16. Using member function `fill` and stream manipulator `setfill` to change the padding character for fields larger than the values being printed.**

(This item is displayed on pages 791 - 792 in the print version)

```

1 // Fig. 15.16: Fig15_16.cpp
2 // Using member function fill and stream manipulator setfill to change
3 // the padding character for fields larger than the printed value.
4 #include <iostream>
5 using std::cout;
6 using std::dec;
7 using std::endl;
8 using std::hex;
9 using std::internal;
10 using std::left;
11 using std::right;
12 using std::showbase;
13
14 #include <iomanip>
15 using std::setfill;
16 using std::setw;
17
18 int main()
19 {
20 int x = 10000;
21
22 // display x

```

```

23 cout << x << " printed as int right and left justified\n"
24 << "and as hex with internal justification.\n"
25 << "Using the default pad character (space):" << endl;
26
27 // display x with base
28 cout << showbase << setw(10) << x << endl;
29
30 // display x with left justification
31 cout << left << setw(10) << x << endl;
32
33 // display x as hex with internal justification
34 cout << internal << setw(10) << hex << x << endl << endl;
35
36 cout << "Using various padding characters:" << endl;
37
38 // display x using padded characters (right justification)
39 cout << right;
40 cout.fill('*');
41 cout << setw(10) << dec << x << endl;
42
43 // display x using padded characters (left justification)
44 cout << left << setw(10) << setfill('%') << x << endl;
45
46 // display x using padded characters (internal justification)
47 cout << internal << setw(10) << setfill('^') << hex
48 << x << endl;
49
50 } // end main

```

10000 printed as int right and left justified  
and as hex with internal justification.

Using the default pad character (space):

10000

10000

0x 2710

Using various padding characters:

\*\*\*\*\*10000

10000%%%%%

0x^^^^2710

#### 15.7.4. Integral Stream Base (dec, oct, hex, showbase)

C++ provides stream manipulators `dec`, `hex` and `oct` to specify that integers are to be displayed as decimal, hexadecimal and octal values, respectively. Stream insertions default to decimal if none of these manipulators is used. With stream extraction, integers prefixed with `0` (zero) are treated as octal values, integers prefixed with `0x` or `0X` are treated as hexadecimal values, and all other integers are treated as decimal values. Once a particular base is specified for a stream, all integers on that stream are processed using that base until a different base is specified or until the program terminates.

Stream manipulator `showbase` forces the base of an integral value to be output. Decimal numbers are output by default, octal numbers are output with a leading `0`, and hexadecimal numbers are output with either a leading `0x` or a leading `0X` (as we discuss in [Section 15.7.6](#), stream manipulator `uppercase` determines which option is chosen). [Figure 15.17](#) demonstrates the use of stream manipulator `showbase` to force an integer to print in decimal, octal and hexadecimal formats. To reset the `showbase` setting, output the stream manipulator `noshowbase`.

**Figure 15.17. Stream manipulator `showbase`.**

```
1 // Fig. 15.17: Fig15_17.cpp
2 // Using stream manipulator showbase.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::hex;
7 using std::oct;
8 using std::showbase;
9
10 int main()
11 {
12 int x = 100;
13
14 // use showbase to show number base
15 cout << "Printing integers preceded by their base:" << endl
16 << showbase;
17
18 cout << x << endl; // print decimal value
19 cout << oct << x << endl; // print octal value
20 cout << hex << x << endl; // print hexadecimal value
21 return 0;
22 } // end main
```

```
Printing integers preceded by their base:
100
0144
0x64
```

### 15.7.5. Floating-Point Numbers; Scientific and Fixed Notation (`scientific`, `fixed`)

Stream manipulators `scientific` and `fixed` control the output format of floating-point numbers. Stream manipulator `scientific` forces the output of a floating-point number to display in scientific format. Stream manipulator `fixed` forces a floating-point number to display a specific number of digits (as specified by member function `precision` or stream manipulator `setprecision`) to the right of the decimal point. Without using another manipulator, the floating-point-number value determines the output format.

Figure 15.18 demonstrates displaying floating-point numbers in fixed and scientific formats using stream manipulators `scientific` (line 21) and `fixed` (line 25). The exponent format in scientific notation might differ across different compilers.

**Figure 15.18. Floating-point values displayed in default, scientific and fixed formats.**

(This item is displayed on page 794 in the print version)

```
1 // Fig. 15.18: Fig15_18.cpp
2 // Displaying floating-point values in system default,
3 // scientific and fixed formats.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8 using std::scientific;
9
10 int main()
11 {
12 double x = 0.001234567;
13 double y = 1.946e9;
14
15 // display x and y in default format
16 cout << "Displayed in default format:" << endl
```

```
17 << x << '\t' << y << endl;
18
19 // display x and y in scientific format
20 cout << "\nDisplayed in scientific format:" << endl
21 << scientific << x << '\t' << y << endl;
22
23 // display x and y in fixed format
24 cout << "\nDisplayed in fixed format:" << endl
25 << fixed << x << '\t' << y << endl;
26 return 0;
27 } // end main
```

```
Displayed in default format:
0.00123457 1.946e+009

Displayed in scientific format:
1.234567e-003 1.946000e+009

Displayed in fixed format:
0.001235 1946000000.000000
```

### 15.7.6. Uppercase/Lowercase Control (`uppercase`)

Stream manipulator `uppercase` outputs an uppercase `x` or `E` with hexadecimal-integer values or with scientific-notation floating-point values, respectively (Fig. 15.19). Using stream manipulator `uppercase` also causes all letters in a hexadecimal value to be uppercase. By default, the letters for hexadecimal values and the exponents in scientific-notation floating-point values appear in lowercase. To reset the `uppercase` setting, output the stream-manipulator `nouppercase`.

[Page 794]

**Figure 15.19. Stream manipulator `uppercase`.**

(This item is displayed on pages 794 - 795 in the print version)

```

1 // Fig. 15.19: Fig15_19.cpp
2 // Stream manipulator uppercase.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::hex;
7 using std::showbase;
8 using std::uppercase;
9
10 int main()
11 {
12 cout << "Printing uppercase letters in scientific"
13 << endl
14 << "notation exponents and hexadecimal values:" << endl;
15
16 // use std::uppercase to display uppercase letters; use std::hex and
17 // std::showbase to display hexadecimal value and its base
18 cout << uppercase << 4.345e10 << endl
19 << hex << showbase << 123456789 << endl;
20 return 0;
21 } // end main

```

Printing uppercase letters in scientific  
 notation exponents and hexadecimal values:  
 4.345E+010  
 0X75BCD15

[Page 795]

### 15.7.7. Specifying Boolean Format (`boolalpha`)

C++ provides data type `bool`, whose values may be `false` or `TRUE`, as a preferred alternative to the old style of using 0 to indicate `false` and nonzero to indicate `TRUE`. A `bool` variable outputs as 0 or 1 by default. However, we can use stream manipulator `boolalpha` to set the output stream to display `bool` values as the strings "`TRUE`" and "`false`". Use stream manipulator `noboolalpha` to set the output stream to display `bool` values as integers (i.e., the default setting). The program of Fig. 15.20 demonstrates these stream manipulators. Line 14 displays the `bool` value, which line 11 sets to `true`, as an integer. Line 18 uses manipulator `boolalpha` to display the `bool` value as a string. Lines 2122 then change the `bool`'s value and use manipulator `noboolalpha`, so line 25 can display the `bool` value as an integer. Line 29 uses manipulator `boolalpha` to display the `bool` value as a string. Both `boolalpha` and

`noboolalpha` are "sticky" settings.

### Figure 15.20. Stream manipulators `boolalpha` and `noboolalpha`.

(This item is displayed on pages 795 - 796 in the print version)

```
1 // Fig. 15.20: Fig15_20.cpp
2 // Demonstrating stream manipulators boolalpha and noboolalpha.
3 #include <iostream>
4 using std::boolalpha;
5 using std::cout;
6 using std::endl;
7 using std::noboolalpha;
8
9 int main()
10 {
11 bool booleanValue = true;
12
13 // display default true booleanValue
14 cout << "booleanValue is " << booleanValue << endl;
15
16 // display booleanValue after using boolalpha
17 cout << "booleanValue (after using boolalpha) is "
18 << boolalpha << booleanValue << endl << endl;
19
20 cout << "switch booleanValue and use noboolalpha" << endl;
21 booleanValue = false; // change booleanValue
22 cout << noboolalpha << endl; // use noboolalpha
23
24 // display default false booleanValue after using noboolalpha
25 cout << "booleanValue is " << booleanValue << endl;
26
27 // display booleanValue after using boolalpha again
28 cout << "booleanValue (after using boolalpha) is "
29 << boolalpha << booleanValue << endl;
30
31 } // end main
```

```

booleanValue is 1
booleanValue (after using boolalpha) is true

switch booleanValue and use noboolalpha

booleanValue is 0
booleanValue (after using boolalpha) is false

```

## Good Programming Practice 15.1



Displaying `bool` values as `true` or `false`, rather than nonzero or 0, respectively, makes program outputs clearer.

---

[Page 796]

### 15.7.8. Setting and Resetting the Format State via Member Function `flags`

Throughout [Section 15.7](#), we have been using stream manipulators to change output format characteristics. We now discuss how to return an output stream's format to its default state after having applied several manipulations. Member function `flags` without an argument returns the current format settings as a `fmtflags` data type (of class `ios_base`), which represents the **format state**. Member function `flags` with a `fmtflags` argument sets the format state as specified by the argument and returns the prior state settings. The initial settings of the value that `flags` returns might differ across several systems. The program of [Fig. 15.21](#) uses member function `flags` to save the stream's original format state (line 22), then restore the original format settings (line 30).

#### Figure 15.21. `flags` member function.

(This item is displayed on pages 796 - 797 in the print version)

```
1 // Fig. 15.21: Fig15_21.cpp
2 // Demonstrating the flags member function.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::ios_base;
7 using std::oct;
8 using std::scientific;
9 using std::showbase;
10
11 int main()
12 {
13 int integerValue = 1000;
14 double doubleValue = 0.0947628;
15
16 // display flags value, int and double values (original format)
17 cout << "The value of the flags variable is: " << cout.flags()
18 << "\nPrint int and double in original format:\n"
19 << integerValue << '\t' << doubleValue << endl << endl;
20
21 // use cout flags function to save original format
22 ios_base::fmtflags originalFormat = cout.flags();
23 cout << showbase << oct << scientific; // change format
24
25 // display flags value, int and double values (new format)
26 cout << "The value of the flags variable is: " << cout.flags()
27 << "\nPrint int and double in a new format:\n"
28 << integerValue << '\t' << doubleValue << endl << endl;
29
30 cout.flags(originalFormat); // restore format
31
32 // display flags value, int and double values (original format)
33 cout << "The restored value of the flags variable is: "
34 << cout.flags()
35 << "\nPrint values in original format again:\n"
36 << integerValue << '\t' << doubleValue << endl;
37 return 0;
38 } // end main
```

```
The value of the flags variable is: 513
Print int and double in original format:
1000 0.0947628
```

```
The value of the flags variable is: 012011
Print int and double in a new format:
01750 9.476280e-002
```

```
The restored value of the flags variable is: 513
Print values in original format again:
1000 0.0947628
```

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 798]

```
cin.eof()
```

returns `TRUE` if end-of-file has been encountered on `cin` and `false` otherwise.

The `failbit` is set for a stream when a format error occurs on the stream, such as when the program is inputting integers and a nondigit character is encountered in the input stream. When such an error occurs, the characters are not lost. The `fail` member function reports whether a stream operation has failed; usually, recovering from such errors is possible.

The `badbit` is set for a stream when an error occurs that results in the loss of data. The `bad` member function reports whether a stream operation failed. Generally, such serious failures are nonrecoverable.

[Page 799]

The `goodbit` is set for a stream if none of the bits `eofbit`, `failbit` or `badbit` is set for the stream.

The `good` member function returns `true` if the `bad`, `fail` and `eof` functions would all return `false`. I/O operations should be performed only on "good" streams.

The `rdstate` member function returns the error state of the stream. A call to `cout.rdstate`, for example, would return the state of the stream, which then could be tested by a `switch` statement that examines `eofbit`, `badbit`, `failbit` and `goodbit`. The preferred means of testing the state of a stream is to use member functions `eof`, `bad`, `fail` and `good`; using these functions does not require the programmer to be familiar with particular status bits.

The `clear` member function is used to restore a stream's state to "good," so that I/O may proceed on that stream. The default argument for `clear` is `goodbit`, so the statement

```
cin.clear();
```

clears `cin` and sets `goodbit` for the stream. The statement

```
cin.clear(ios::failbit)
```

sets the failbit. The programmer might want to do this when performing input on `cin` with a user-defined type and encountering a problem. The name `clear` might seem inappropriate in this context, but it is correct.

The program of Fig. 15.22 demonstrates member functions `rdstate`, `eof`, `fail`, `bad`, `good` and `clear`. [Note: The actual values output may differ across different compilers.]

The operator! member function of `basic_ios` returns `TRUE` if the `badbit` is set, the `failbit` is set or both are set. The operator `void *` member function returns `false (0)` if the `badbit` is set, the `failbit` is set or both are set. These functions are useful in file processing when a `TRUE/false` condition is being tested under the control of a selection statement or repetition statement.

 PREVNEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 800]

## 15.9. Tying an Output Stream to an Input Stream

Interactive applications generally involve an `istream` for input and an `ostream` for output. When a prompting message appears on the screen, the user responds by entering the appropriate data. Obviously, the prompt needs to appear before the input operation proceeds. With output buffering, outputs appear only when the buffer fills, when outputs are flushed explicitly by the program or automatically at the end of the program. C++ provides member function `tie` to synchronize (i.e., "tie together") the operation of an `istream` and an `ostream` to ensure that outputs appear before their subsequent inputs. The call

```
cin.tie(&cout);
```

ties `cout` (an `ostream`) to `cin` (an `istream`). Actually, this particular call is redundant, because C++ performs this operation automatically to create a user's standard input/output environment. However, the user would tie other `istream/ostream` pairs explicitly. To untie an input stream, `inputStream`, from an output stream, use the call

```
inputStream.tie(0);
```

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 800 (continued)]

## 15.10. Wrap-Up

This chapter summarized how C++ performs input/output using streams. You learned about the stream-I/O classes and objects, as well as the stream I/O template class hierarchy. We discussed `ostream`'s formatted and unformatted output capabilities performed by the `put` and `write` functions. You saw examples using `istream`'s formatted and unformatted input capabilities performed by the `eof`, `get`, `getline`, `peek`, `putback`, `ignore` and `read` functions. Next, we discussed stream manipulators and member functions that perform formatting tasks `dec`, `oct`, `hex` and `setbase` for displaying integers; `precision` and `setprecision` for controlling floating-point precision; and `width` and `setw` for setting field width. You also learned additional formatting `iostream` manipulators and member functions `showpoint` for displaying decimal point and trailing zeros; `left`, `right` and `internal` for justification; `fill` and `setfill` for padding; `scientific` and `fixed` for displaying floating-point numbers in scientific and fixed notation; `uppercase` for uppercase/lowercase control; `boolalpha` for specifying boolean format; and `flags` and `fmtflags` for resetting the format state.

In the next chapter, we introduce exception-handling, which allows programmers to deal with certain problems that may occur during a program's execution. We demonstrate basic exception-handling techniques that often permit a program to continue executing as if no problem had been encountered. We also present several classes that the C++ Standard Library provides for handling exceptions.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 801]

- C++ provides "low-level" and "high-level" I/O capabilities. Low-level I/O-capabilities specify that some number of bytes should be transferred device-to-memory or memory-to-device. High-level I/O is performed with bytes grouped into such meaningful units as integers, floats, characters, strings and user-defined types.
- C++ provides both unformatted-I/O and formatted-I/O operations. Unformatted-I/O transfers are fast, but process raw data that is difficult for people to use. Formatted I/O processes data in meaningful units, but requires extra processing time that can degrade the performance of high-volume data transfers.
- The `<iostream>` header file declares all stream-I/O operations.
- Header `<iomanip>` declares the parameterized stream manipulators.
- The `<fstream>` header declares file-processing operations.
- The `basic_istream` template supports stream-input operations.
- The `basic_ostream` template supports stream-output operations.
- The `basic_iostream` template supports both stream-input and stream-output operations.
- The `basic_istream` template and the `basic_ostream` template are each derived through single inheritance from the `basic_ios` template.
- The `basic_iostream` template is derived through multiple inheritance from both the `basic_istream` template and the `basic_ostream` template.
- The left-shift operator (`<<`) is overloaded to designate stream output and is referred to as the stream insertion operator.
- The right-shift operator (`>>`) is overloaded to designate stream input and is referred to as the stream extraction operator.
- The `istream` object `cin` is tied to the standard input device, normally the keyboard.
- The `ostream` object `cout` is tied to the standard output device, normally the screen.
- The `ostream` object `cerr` is tied to the standard error device. Outputs to `cerr` are unbuffered; each insertion to `cerr` appears immediately.
- The C++ compiler determines data types automatically for input and output.
- Addresses are displayed in hexadecimal format by default.
- To print the address in a pointer variable, cast the pointer to `void *`.
- Member function `put` outputs one character. Calls to `put` may be cascaded.
- Stream input is performed with the stream extraction operator `>>`. This operator automatically skips white-space characters in the input stream.
- The `>>` operator returns `false` after end-of-file is encountered on a stream.
- Stream extraction causes `failbit` to be set for improper input and `badbit` to be set if the operation fails.
- A series of values can be input using the stream extraction operation in a `while` loop header. The extraction returns `0` when end-of-file is encountered.
- The `get` member function with no arguments inputs one character and returns the character; `EOF` is returned if end-of-file is encountered on the stream.
- Member function `get` with a character-reference argument inputs the next character from the input stream and stores it in the character argument. This version of `get` returns a reference to the `istream` object for which the `get` member function is being invoked.

- Member function `get` with three arguments a character array, a size limit and a delimiter (with default value newline) reads characters from the input stream up to a maximum of limit 1 characters and terminates, or terminates when the delimiter is read. The input string is terminated with a null character. The delimiter is not placed in the character array but remains in the input stream.
- 

[Page 802]

- The `getline` member function operates like the three-argument `get` member function. The `getline` function removes the delimiter from the input stream but does not store it in the string.
- Member function `ignore` skips the specified number of characters (the default is 1) in the input stream; it terminates if the specified delimiter is encountered (the default delimiter is `EOF`).
- The `putback` member function places the previous character obtained by a `get` on a stream back onto that stream.
- The `peek` member function returns the next character from an input stream but does not extract (remove) the character from the stream.
- C++ offers type-safe I/O. If unexpected data is processed by the `<<` and `>>` operators, various error bits are set, which the user may test to determine whether an I/O operation succeeded or failed. If operator `<<` has not been overloaded for a user-defined type, a compiler error is reported.
- Unformatted I/O is performed with member functions `read` and `write`. These input or output some number of bytes to or from memory, beginning at a designated memory address. They are input or output as raw bytes with no formatting.
- The `gcount` member function returns the number of characters input by the previous `read` operation on that stream.
- Member function `read` inputs a specified number of characters into a character array. `failbit` is set if fewer than the specified number of characters are read.
- To change the base in which integers output, use the manipulator `hex` to set the base to hexadecimal (base 16) or `oct` to set the base to octal (base 8). Use manipulator `dec` to reset the base to decimal. The base remains the same until changed explicitly.
- The parameterized stream manipulator `setbase` also sets the base for integer output. `setbase` takes one integer argument of 10, 8 or 16 to set the base.
- Floating-point precision can be controlled using either the `setprecision` stream manipulator or the `precision` member function. Both set the precision for all subsequent output operations until the next precision-setting call. The `precision` member function with no argument returns the current precision value.
- Parameterized manipulators require the inclusion of the `<iomanip>` header file.
- Member function `width` sets the field width and returns the previous width. Values narrower than the field are padded with fill characters. The field-width setting applies only for the next insertion or extraction; the field width is set to 0 implicitly (subsequent values will be output as large as necessary). Values wider than a field are printed in their entirety. Function `width` with no argument returns the current width setting. Manipulator `setw` also sets the width.
- For input, the `setw` stream manipulator establishes a maximum string size; if a larger string is entered, the larger line is broken into pieces no larger than the designated size.
- Programmers may create their own stream manipulators.
- Stream manipulator `showpoint` forces a floating-point number to be output with a decimal point and with the number of significant digits specified by the precision.
- Stream manipulators `left` and `right` cause fields to be left justified with padding characters to the

right or right justified with padding characters to the left.

- Stream manipulator `internal` indicates that a number's sign (or base when using stream manipulator `showbase`) should be left justified within a field, its magnitude should be right justified and intervening spaces should be padded with the fill character.

[Page 803]

- Member function `fill` specifies the fill character to be used with stream manipulators `left`, `right` and `internal` (space is the default); the prior padding character is returned. Stream manipulator `setfill` also sets the fill character.
- Stream manipulators `oct`, `hex` and `dec` specify that integers are to be treated as octal, hexadecimal or decimal values, respectively. Integer output defaults to decimal if none of these bits is set; stream extractions process the data in the form the data is supplied.
- Stream manipulator `showbase` forces the base of an integral value to be output.
- Stream manipulator `scientific` is used to output a floating-point number in scientific format. Stream manipulator `fixed` is used to output a floating-point number with the precision specified by the `precision` member function.
- Stream manipulator `uppercase` forces an uppercase `x` or `E` to be output with hexadecimal integers or with scientific-notation floating-point values, respectively. When `set`, `uppercase` causes all letters in a hexadecimal value to be uppercase.
- Member function `flags` with no argument returns the `long` value of the current format state settings. Function `flags` with a `long` argument sets the format state specified by the argument.
- The state of a stream may be tested through bits in class `ios_base`.
- The `eofbit` is set for an input stream after end-of-file is encountered during an input operation. The `eof` member function reports whether the `eofbit` has been set.
- The `failbit` is set for a stream when a format error occurs on the stream. The `fail` member function reports whether a stream operation has failed; it is normally possible to recover from such errors.
- The `badbit` is set for a stream when an error occurs that results in data loss. The `bad` member function reports whether such a stream operation failed. Such serious failures are normally non-recoverable.
- The `good` member function returns true if the `bad`, `fail` and `eof` functions would all return `false`. I/O operations should be performed only on "good" streams.
- The `rdstate` member function returns the error state of the stream.
- Member function `clear` restores a stream's state to "good," so that I/O may proceed on that stream.
- C++ provides the `tie` member function to synchronize `istream` and `ostream` operations to ensure that outputs appear before subsequent inputs.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 804]

`gcount` member function of `basic_istream`

`get` member function of `basic_istream`

`getline` member function of `basic_istream`

`good` member function of `basic_ios`

`hex` stream manipulator

`ifstream`

`ignore` member function of `basic_istream`

in-memory formatting

internal stream manipulator

`<iomanip>` header file

`ios_base` class

`iostream`

`istream`

leading 0 (octal)

leading 0x or 0X (hexadecimal)

`left` stream manipulator

`noboolalpha` stream manipulator

`noshowbase` stream manipulator

`noshowpoint` stream manipulator

`noshowpos` stream manipulator

`noskipws` stream manipulator

`nouppercase` stream manipulator

`oct` stream manipulator

`ofstream`

`operator void*` member function of `basic_ios`

`operator!` member function of `basic_ios`

`ostream`

output buffering

padding

parameterized stream manipulator

`peek` member function of `basic_istream`

`precision` member function of `ios_base`

predefined streams

`put` member function of `basic_ostream`

`putback` member function of `basic_istream`

`rdstate` member function of `basic_ios`

`read` member function of `basic_istream`

`right` stream manipulator

`scientific` stream manipulator

`setbase` stream manipulator

`setfill` stream manipulator

`setprecision` stream manipulator

`setw` stream manipulator

`showbase` stream manipulator

`showpoint` stream manipulator

`showpos` stream manipulator

`skipws` stream manipulator

stream input

stream manipulator

stream output

stream extraction operator (`>>`)

stream insertion operator (`<<`)

`tie` member function of `basic_ios`

`typedef`

type-safe I/O

unbuffered output

unformatted I/O

`uppercase` stream manipulator

`width` stream manipulator

`write` member function of `basic_ostream`

 PREV

NEXT 

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 805]

The symbol for the stream extraction operator is \_\_\_\_\_.

The stream manipulators \_\_\_\_\_, \_\_\_\_\_ and \_\_\_\_\_ specify that integers should be displayed in octal, hexadecimal and decimal formats, respectively.

When used, the \_\_\_\_\_ stream manipulator causes positive numbers to display with a plus sign.

## 15.2

State whether the following are true or false. If the answer is false, explain why.

a.

The stream member function `flags` with a `long` argument sets the `flags` state variable to its argument and returns its previous value.

b.

The stream insertion operator `<<` and the stream-extraction operator `>>` are overloaded to handle all standard data types including strings and memory addresses (stream-insertion only) and all user-defined data types.

c.

The stream member function `flags` with no arguments resets the stream's format state.

d.

The stream extraction operator `>>` can be overloaded with an operator function that takes an `istream` reference and a reference to a user-defined type as arguments and returns an `istream` reference.

The stream insertion operator << can be overloaded with an operator function that takes an `istream` reference and a reference to a user-defined type as arguments and returns an `istream` reference.

f.

Input with the stream extraction operator >> always skips leading white-space characters in the input stream, by default.

g.

The stream member function `rdstate` returns the current state of the stream.

h.

The `cout` stream normally is connected to the display screen.

i.

The stream member function `good` returns `TRue` if the `bad`, `fail` and `eof` member functions all return `false`.

j.

The `cin` stream normally is connected to the display screen.

k.

If a nonrecoverable error occurs during a stream operation, the `bad` member function will return `TRue`.

l.

Output to `cerr` is unbuffered and output to `clog` is buffered.

m.

Stream manipulator `showpoint` forces floating-point values to print with the default six digits of precision unless the precision value has been changed, in which case floating-point values print with the specified precision.

n.

The `ostream` member function `put` outputs the specified number of characters.

**o.**

The stream manipulators `dec`, `oct` and `hex` affect only the next integer output operation.

**p.**

By default, memory addresses are displayed as `long` integers.

### 15.3

For each of the following, write a single statement that performs the indicated task.

**a.**

Output the string "Enter your name: ".

**b.**

Use a stream manipulator that causes the exponent in scientific notation and the letters in hexadecimal values to print in capital letters.

**c.**

Output the address of the variable `myString` of type `char *`.

**d.**

Use a stream manipulator to ensure floating-point values print in scientific notation.

**e.**

Output the address in variable `integerPtr` of type `int *`.

**f.**

Use a stream manipulator such that, when integer values are output, the integer base for octal and hexadecimal values is displayed.

**g.**

Output the value pointed to by `floatPtr` of type `float *`.

**h.**

Use a stream member function to set the fill character to '\*' for printing in field widths larger than the values being output. Write a separate statement to do this with a stream manipulator.

**i.**

Output the characters '0' and 'K' in one statement with `ostream` function `put`.

**j.**

Get the value of the next character in the input stream without extracting it from the stream.

---

[Page 806]

**k.**

Input a single character into variable `charValue` of type `char`, using the `istream` member function `get` in two different ways.

**l.**

Input and discard the next six characters in the input stream.

**m.**

Use `istream` member function `read` to input 50 characters into `char` array `line`.

**n.**

Read 10 characters into character array `name`. Stop reading characters if the '.' delimiter is encountered. Do not remove the delimiter from the input stream. Write another statement that performs this task and removes the delimiter from the input.

**o.**

Use the `istream` member function `gcount` to determine the number of characters input into character array `line` by the last call to `istream` member function `read`, and output that number of characters, using `ostream` member function `write`.

**p.**

Output the following values: 124, 18.376, 'z', 1000000 and "String".

**q.**

Print the current precision setting, using a member function of object cout.

**r.**

Input an integer value into int variable months and a floating-point value into float variable percentageRate.

**s.**

Print 1.92, 1.925 and 1.9258 separated by tabs and with 3 digits of precision, using a manipulator.

**t.**

Print integer 100 in octal, hexadecimal and decimal, using stream manipulators.

**u.**

Print integer 100 in decimal, octal and hexadecimal, using a stream manipulator to change the base.

**v.**

Print 1234 right justified in a 10-digit field.

**w.**

Read characters into character array line until the character 'z' is encountered, up to a limit of 20 characters (including a terminating null character). Do not extract the delimiter character from the stream.

**x.**

Use integer variables x and y to specify the field width and precision used to display the double value 87.4573, and display the value.

Identify the error in each of the following statements and explain how to correct it.

a.

```
cout << "Value of x <= y is: " << x <= y;
```

b.

The following statement should print the integer value of 'c'.

```
cout << 'c' ;
```

c.

```
cout << ""A string in quotes"" ;
```

## 15.5

For each of the following, show the output.

a.

```
cout << "12345" << endl;
cout.width(5);
cout.fill('*');
cout << 123 << endl << 123;
```

b.

```
cout << setw(10) << setfill('$') << 10000;
```

c.

```
cout << setw(8) << setprecision(3) << 1024.987654;
```

d.

```
cout << showbase << oct << 99 << endl << hex << 99;
```

e.

```
cout << 100000 << endl << showpos << 100000;
```

```
cout << setw(10) << setprecision(2) << scientific << 444.93738;
```

 PREV

NEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 807]

### 15.3

a.

```
cout << "Enter your name: ";
```

b.

```
cout << uppercase;
```

c.

```
cout << static_cast< void * >(myString);
```

d.

```
cout << scientific;
```

e.

```
cout << integerPtr;
```

f.

```
cout << showbase;
```

g.

```
cout << *floatPtr;
```

h.

```
cout.fill('*');
cout << setfill('*');
```

i.

```
cout.put('0').put('K');
```

j.

```
cin.peek();
```

k.

```
charValue = cin.get();
cin.get(charValue);
```

l.

```
cin.ignore(6);
```

m.

```
cin.read(line, 50);
```

n.

```
cin.get(name, 10, '.');
cin.getline(name, 10, '.');
```

o.

```
cout.write(line, cin.gcount());
```

p.

```
cout << 124 << ' ' << 18.376 << ' ' << "Z" << 1000000 << "String";
```

q.

```
cout << cout.precision();
```

r.

```
cin >> months >> percentageRate;
```

s.

```
cout << setprecision(3) << 1.92 << '\t' << 1.925 << '\t' << 1.9258;
```

t.

```
cout << oct << 100 << '\t' << hex << 100 << '\t' << dec << 100;
```

**u.**

```
cout << 100 << '\t' << setbase(8) << 100 << '\t' << setbase(16) << 100;
```

**v.**

```
cout << setw(10) << 1234;
```

**w.**

```
cin.get(line, 20, 'z');
```

**x.**

```
cout << setw(x) << setprecision(y) << 87.4573;
```

## 15.4

**a.**

Error: The precedence of the `<<` operator is higher than that of `=`, which causes the statement to be evaluated improperly and also causes a compiler error.

Correction: To correct the statement, place parentheses around the expression `x <= y`. This problem will occur with any expression that uses operators of lower precedence than the `<<` operator if the expression is not placed in parentheses.

**b.**

Error: In C++, characters are not treated as small integers, as they are in C.

Correction: To print the numerical value for a character in the computer's character set, the character must be cast to an integer value, as in the following:

```
cout << static_cast< int >('c');
```

**c.**

Error: Quote characters cannot be printed in a string unless an escape sequence is used.

Correction: Print the string in one of the following ways:

```
cout << " " << "A string in quotes" << " " ;
cout << "\\" "A string in quotes\\\" ;
```

---

[Page 808]

## 15.5

a.

```
12345
**123
123
```

b.

```
$$$$$$10000
```

c.

```
1024.988
```

d.

```
0143
0x63
```

e.

```
100000
+100000
```

f.

```
4.45e+002
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 809]

•

Input the entire phone number into an array. Test that the proper number of characters has been entered. There should be a total of 14 characters read for a phone number of the form (800) 555-1212. Use `ios_base-member-function clear` to set `failbit` for improper input.

•

The area code and exchange do not begin with 0 or 1. Test the first digit of the area-code and exchange portions of the phone number to be sure that neither begins with 0 or 1. Use `ios_base-member-function clear` to set `failbit` for improper input.

•

The middle digit of an area code used to be limited to 0 or 1 (although this has changed recently). Test the middle digit for a value of 0 or 1. Use the `ios_base-member-function clear` to set `failbit` for improper input. If none of the above operations results in `failbit` being set for improper input, copy the three parts of the telephone number into the `areaCode`, `exchange` and `line` members of the `PhoneNumber` object. In the main program, if `failbit` has been set on the input, have the program print an error message and end, rather than print the phone number.

## 15.15

Write a program that accomplishes each of the following:

a.

Create a user-defined class `Point` that contains the private integer data members `xCoordinate` and `yCoordinate` and declares stream insertion and stream extraction overloaded operator functions as friends of the class.

b.

Define the stream insertion and stream extraction operator functions. The stream extraction operator function should determine whether the data entered is valid, and, if not, it should set the `failbit` to indicate improper input. The stream insertion operator should not be able to display the point after an input error occurred.

c.

Write a `main` function that tests input and output of user-defined class `Point`, using the overloaded stream extraction and stream insertion operators.

### 15.16

Write a program that accomplishes each of the following:

a.

Create a user-defined class `Complex` that contains the private integer data members `real` and `imaginary` and declares stream insertion and stream extraction overloaded operator functions as friends of the class.

b.

Define the stream insertion and stream extraction operator functions. The stream extraction operator function should determine whether the data entered is valid, and, if not, it should set `failbit` to indicate improper input. The input should be of the form

3 + 8i

c.

The values can be negative or positive, and it is possible that one of the two values is not provided. If a value is not provided, the appropriate data member should be set to 0. The stream-insertion operator should not be able to display the point if an input error occurred. For negative imaginary values, a minus sign should be printed rather than a plus sign.

d.

Write a `main` function that tests input and output of user-defined class `Complex`, using the overloaded stream extraction and stream insertion operators.

### 15.17

Write a program that uses a `for` statement to print a table of ASCII values for the characters in the ASCII character set from 33 to 126. The program should print the decimal value, octal value, hexadecimal value and character value for each character. Use the stream manipulators `dec`, `oct` and `hex` to print the integer values.

### 15.18

Write a program to show that the `getline` and three-argument `get istream` member functions both

end the input string with a string-terminating null character. Also, show that `get` leaves the delimiter character on the input stream, whereas `getline` exTRacts the delimiter character and discards it. What happens to the unread characters in the stream?

 PREVNEXT **page footer**

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 811]

## Outline

[16.1 Introduction](#)

[16.2 Exception-Handling Overview](#)

[16.3 Example: Handling an Attempt to Divide by Zero](#)

[16.4 When to Use Exception Handling](#)

[16.5 Rethrowing an Exception](#)

[16.6 Exception Specifications](#)

[16.7 Processing Unexpected Exceptions](#)

[16.8 Stack Unwinding](#)

[16.9 Constructors, Destructors and Exception Handling](#)

[16.10 Exceptions and Inheritance](#)

[16.11 Processing `new` Failures](#)

[16.12 Class `auto\_ptr` and Dynamic Memory Allocation](#)

[16.13 Standard Library Exception Hierarchy](#)

[16.14 Other Error-Handling Techniques](#)

[16.15 Wrap-Up](#)

[Summary](#)

[Terminology](#)

## Self-Review Exercises

### Answers to Self-Review Exercises

### Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 812]

The chapter begins with an overview of exception-handling concepts, then demonstrates basic exception-handling techniques. We show these techniques via an example that demonstrates handling an exception that occurs when a function attempts to divide by zero. We then discuss additional exception-handling issues, such as how to handle exceptions that occur in a constructor or destructor and how to handle exceptions that occur if operator `new` fails to allocate memory for an object. We conclude the chapter by introducing several classes that the C++ Standard Library provides for handling exceptions.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 812 (continued)]

## 16.2. Exception-Handling Overview

Program logic frequently tests conditions that determine how program execution proceeds. Consider the following pseudocode:

Perform a task

If the preceding task did not execute correctly

    Perform error processing

Perform next task

If the preceding task did not execute correctly

    Perform error processing

...

In this pseudocode, we begin by performing a task. We then test whether that task executed correctly. If not, we perform error processing. Otherwise, we continue with the next task. Although this form of error handling works, intermixing program logic with error-handling logic can make the program difficult to read, modify, maintain and debug especially in large applications.

### Performance Tip 16.1



If the potential problems occur infrequently, intermixing program logic and error-handling logic can degrade a program's performance, because the program must (potentially frequently) perform tests to determine whether the task executed correctly and the next task can be performed.

Exception handling enables the programmer to remove error-handling code from the "main line" of the program's execution, which improves program clarity and enhances modifiability. Programmers can decide to handle any exceptions they choose all exceptions, all exceptions of a certain type or all exceptions of a group of related types (e.g., exception types that belong to an inheritance hierarchy). Such flexibility reduces the likelihood that errors will be overlooked and thereby makes a program more robust.

With programming languages that do not support exception handling, programmers often delay writing error-processing code or sometimes forget to include it. This results in less robust software products. C++ enables the programmer to deal with exception handling easily from the inception of a project.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 813]

**Figure 16.1. Class DivideByZeroException definition.**

```

1 // Fig. 16.1: DivideByZeroException.h
2 // Class DivideByZeroException definition.
3 #include <stdexcept> // stdexcept header file contains runtime_error
4 using std::runtime_error; // standard C++ library class runtime_error
5
6 // DivideByZeroException objects should be thrown by functions
7 // upon detecting division-by-zero exceptions
8 class DivideByZeroException : public runtime_error
9 {
10 public:
11 // constructor specifies default error message
12 DivideByZeroException::DivideByZeroException()
13 : runtime_error("attempted to divide by zero") {}
14 };// end class DivideByZeroException

```

**Figure 16.2. Exception-handling example that throws exceptions on attempts to divide by zero.**

(This item is displayed on pages 814 - 815 in the print version)

```

1 // Fig. 16.2: Fig16_02.cpp
2 // A simple exception-handling example that checks for
3 // divide-by-zero exceptions.
4 #include <iostream>
5 using std::cin;
6 using std::cout;
7 using std::endl;
8
9 #include "DivideByZeroException.h" // DivideByZeroException class
10
11 // perform division and throw DivideByZeroException object if
12 // divide-by-zero exception occurs
13 double quotient(int numerator, int denominator)
14 {
15 // throw DivideByZeroException if trying to divide by zero
16 if (denominator == 0)

```

```
17 throw DivideByZeroException(); // terminate function
18
19 // return division result
20 return static_cast< double >(numerator) / denominator;
21 } // end function quotient
22
23 int main()
24 {
25 int number1; // user-specified numerator
26 int number2; // user-specified denominator
27 double result; // result of division
28
29 cout << "Enter two integers (end-of-file to end): ";
30
31 // enable user to enter two integers to divide
32 while (cin >> number1 >> number2)
33 {
34 // try block contains code that might throw exception
35 // and code that should not execute if an exception occurs
36 try
37 {
38 result = quotient(number1, number2);
39 cout << "The quotient is: " << result << endl;
40 } // end try
41
42 // exception handler handles a divide-by-zero exception
43 catch (DivideByZeroException ÷ByZeroException)
44 {
45 cout << "Exception occurred: "
46 << divideByZeroException.what() << endl;
47 } // end catch
48
49 cout << "\nEnter two integers (end-of-file to end): ";
50 } // end while
51
52 cout << endl;
53 return 0; // terminate normally
54 } // end main
```

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z
```

In this example, we define a function named `quotient` that receives two integers input by the user and divides its first `int` parameter by its second `int` parameter. Before performing the division, the function casts the first `int` parameter's value to type `double`. Then, the second `int` parameter's value is promoted to type `double` for the calculation. So function `quotient` actually performs the division using two `double` values and returns a `double` result.

Although division by zero is allowed in floating-point arithmetic, for the purpose of this example, we treat any attempt to divide by zero as an error. Thus, function `quotient` tests its second parameter to ensure that it is not zero before allowing the division to proceed. If the second parameter is zero, the function uses an exception to indicate to the caller that a problem occurred. The caller (`main` in this example) can then process this exception and allow the user to type two new values before calling function `quotient` again. In this way, the program can continue to execute even after an improper value is entered, thus making the program more robust.

The example consists of two files `DivideByZeroException.h` (Fig. 16.1) defines an exception class that represents the type of the problem that might occur in the example, and `fig16_02.cpp` (Fig. 16.2) defines the `quotient` function and the `main` function that calls it. Function `main` contains the code that demonstrates exception handling.

## Defining an Exception Class to Represent the Type of Problem That Might Occur

Figure 16.1 defines class `DivideByZeroException` as a derived class of Standard Library class `runtime_error` (defined in header file `<stdexcept>`). Class `runtime_error` is a derived class of Standard Library class `exception` (defined in header file `<exception>`) is the C++ standard base class for representing runtime errors. Class `exception` is the standard C++ base class for all exceptions. (Section 16.13 discusses class `exception` and its derived classes in detail.) A typical exception class that derives from the `runtime_error` class defines only a constructor (e.g., lines 1213) that passes an error-message string to the base-class `runtime_error` constructor. Every exception class that derives directly or indirectly from `exception` contains the virtual function `what`, which returns an exception object's error message. Note that you are not required to derive a custom exception class, such as `DivideByZeroException`, from the standard exception classes provided by C++. However, doing so

allows programmers to use the `virtual` function `what` to obtain an appropriate error message. We use an object of this `DivideByZeroException` class in Fig. 16.2 to indicate when an attempt is made to divide by zero.

---

[Page 815]

## Demonstrating Exception Handling

The program in Fig. 16.2 uses exception handling to wrap code that might throw a "divide-by-zero" exception and to handle that exception, should one occur. The application enables the user to enter two integers, which are passed as arguments to function `quotient` (lines 1321). This function divides the first number (numerator) by the second number (denominator). Assuming that the user does not specify 0 as the denominator for the division, function `quotient` returns the division result. However, if the user inputs a 0 value as the denominator, function `quotient` throws an exception. In the sample output, the first two lines show a successful calculation, and the next two lines show a failed calculation due to an attempt to divide by zero. When the exception occurs, the program informs the user of the mistake and prompts the user to input two new integers. After we discuss the code, we will consider the user inputs and flow of program control that yield these outputs.

## Enclosing Code in a TRY Block

The program begins by prompting the user to enter two integers. The integers are input in the condition of the `while` loop (line 32). After the user inputs values that represent the numerator and denominator, program control proceeds into the loop's body (lines 3350). Line 38 passes these values to function `quotient` (lines 1321), which either divides the integers and returns a result, or **throws an exception** (i.e., indicates that an error occurred) on an attempt to divide by zero. Exception handling is geared to situations in which the function that detects an error is unable to handle it.

C++ provides **try blocks** to enable exception handling. A `try` block consists of keyword `try` followed by braces (`{ }`) that define a block of code in which exceptions might occur. The `TRY` block encloses statements that might cause exceptions and statements that should be skipped if an exception occurs.

---

[Page 816]

Note that a `TRY` block (lines 3640) encloses the invocation of function `quotient` and the statement that displays the division result. In this example, because the invocation to function `quotient` (line 38) can throw an exception, we enclose this function invocation in a `TRY` block. Enclosing the output statement (line 39) in the `try` block ensures that the output will occur only if function `quotient` returns a result.

## Software Engineering Observation 16.2



Exceptions may surface through explicitly mentioned code in a `try` block, through calls to other functions and through deeply nested function calls initiated by code in a `try` block.

## Defining a `catch` Handler to Process a `DivideByZeroException`

Exceptions are processed by **catch handlers** (also called **exception handlers**), which catch and handle exceptions. At least one `catch` handler (lines 4347) must immediately follow each `try` block. Each `catch` handler begins with the keyword `catch` and specifies in parentheses an **exception parameter** that represents the type of exception the `catch` handler can process (`DivideByZeroException` in this case). When an exception occurs in a `try` block, the `catch` handler that executes is the one whose type matches the type of the exception that occurred (i.e., the type in the `catch` block matches the thrown exception type exactly or is a base class of it). If an exception parameter includes an optional parameter name, the `catch` handler can use that parameter name to interact with a caught exception object in the body of the `catch` handler, which is delimited by braces (`{` and `}`). A `catch` handler typically reports the error to the user, logs it to a file, terminates the program gracefully or tries an alternate strategy to accomplish the failed task. In this example, the `catch` handler simply reports that the user attempted to divide by zero. Then the program prompts the user to enter two new integer values.

### Common Programming Error 16.1



It is a syntax error to place code between a `try` block and its corresponding `catch` handlers.

### Common Programming Error 16.2



Each `catch` handler can have only a single parameter specifying a comma-separated list of exception parameters is a syntax error.

### Common Programming Error 16.3



It is a logic error to catch the same type in two different `catch` handlers following a single `try` block.

## Termination Model of Exception Handling

If an exception occurs as the result of a statement in a `TRY` block, the `try` block expires (i.e., terminates immediately). Next, the program searches for the first `catch` handler that can process the type of exception that occurred. The program locates the matching `catch` by comparing the thrown exception's type to each `catch`'s exception-parameter type until the program finds a match. A match occurs if the types are identical or if the thrown exception's type is a derived class of the exception-parameter type. When a match occurs, the code contained within the matching `catch` handler executes. When a `catch` handler finishes processing by reaching its closing right brace (`}`), the exception is considered handled and the local variables defined within the `catch` handler (including the `catch` parameter) go out of scope. Program control does not return to the point at which the exception occurred (known as the **throw point**), because the `try` block has expired. Rather, control resumes with the first statement (line 49) after the last `catch` handler following the `try` block. This is known as the **termination model of exception handling**. [Note: Some languages use the **resumption model of exception handling**, in which, after an exception is handled, control resumes just after the throw point.] As with any other block of code, when a `try` block terminates, local variables defined in the block go out of scope.

[Page 817]

### Common Programming Error 16.4



Logic errors can occur if you assume that after an exception is handled, control will return to the first statement after the throw point.

### Error-Prevention Tip 16.2



With exception handling, a program can continue executing (rather than terminating) after dealing with a problem. This helps ensure the kind of robust applications that contribute to what is called mission-critical computing or business-critical computing.

If the `try` block completes its execution successfully (i.e., no exceptions occur in the `TRY` block), then the program ignores the `catch` handlers and program control continues with the first statement after the last `catch` following that `TRY` block. If no exceptions occur in a `try` block, the program ignores the `catch` handler(s) for that block.

If an exception that occurs in a `TRY` block has no matching `catch` handler, or if an exception occurs in a statement that is not in a `try` block, the function that contains the statement terminates immediately, and the program attempts to locate an enclosing `try` block in the calling function. This process is called **stack unwinding** and is discussed in [Section 16.8](#).

## Flow of Program Control When the User Enters a Nonzero Denominator

Consider the flow of control when the user inputs the numerator 100 and the denominator 7 (i.e., the first two lines of output in Fig. 16.2). In line 16, function `quotient` determines that the denominator does not equal zero, so line 20 performs the division and returns the result (14.2857) to line 38 as a `double` (the `static_cast< double >` in line 20 ensures the proper return value type). Program control then continues sequentially from line 38, so line 39 displays the division result and line 40 is the end of the `try` block. Because the `try` block completed successfully and did not throw an exception, the program does not execute the statements contained in the `catch` handler (lines 4347), and control continues to line 49 (the first line of code after the `catch` handler), which prompts the user to enter two more integers.

## Flow of Program Control When the User Enters a Denominator of Zero

Now let us consider a more interesting case in which the user inputs the numerator 100 and the denominator 0 (i.e., the third and fourth lines of output in Fig. 16.2). In line 16, `quotient` determines that the denominator equals zero, which indicates an attempt to divide by zero. Line 17 throws an exception, which we represent as an object of class `DivideByZeroException` (Fig. 16.1).

[Page 818]

Note that, to throw an exception, line 17 uses keyword `throw` followed by an operand that represents the type of exception to throw. Normally, a `throw` statement specifies one operand. (In Section 16.5, we discuss how to use a `throw` statement that specifies no operands.) The operand of a `throw` can be of any type. If the operand is an object, we call it an **exception object**; in this example, the exception object is an object of type `DivideByZeroException`. However, a `throw` operand also can assume other values, such as the value of an expression (e.g., `throw x > 5`) or the value of an `int` (e.g., `throw 5`). The examples in this chapter focus exclusively on throwing exception objects.

### Common Programming Error 16.5



Use caution when throwing the result of a conditional expression (`? :`), because promotion rules could cause the value to be of a type different from the one expected. For example, when throwing an `int` or a `double` from the same conditional expression, the conditional expression converts the `int` to a `double`. However, the `catch` handler always catches the result as a `double`, rather than catching the result as a `double` when a `double` is thrown, and catching the result as an `int` when an `int` is thrown.

As part of throwing an exception, the `throw` operand is created and used to initialize the parameter in the `catch` handler, which we discuss momentarily. In this example, the `throw` statement in line 17

creates an object of class `DivideByZeroException`. When line 17 throws the exception, function `quotient` exits immediately. Therefore, line 17 throws the exception before function `quotient` can perform the division in line 20. This is a central characteristic of exception handling: A function should throw an exception before the error has an opportunity to occur.

Because we decided to enclose the invocation of function `quotient` (line 38) in a `try` block, program control enters the `catch` handler (lines 4347) that immediately follows the `try` block. This `catch` handler serves as the exception handler for the divide-by-zero exception. In general, when an exception is thrown within a `try` block, the exception is caught by a `catch` handler that specifies the type matching the thrown exception. In this program, the `catch` handler specifies that it catches `DivideByZeroException` objects; this type matches the object type thrown in function `quotient`. Actually, the `catch` handler catches a reference to the `DivideByZeroException` object created by function `quotient`'s `throw` statement (line 17).

## Performance Tip 16.2



Catching an exception object by reference eliminates the overhead of copying the object that represents the thrown exception.

## Good Programming Practice 16.1



Associating each type of runtime error with an appropriately named exception object improves program clarity.

The `catch` handler's body (lines 4546) prints the associated error message returned by calling function `what` of base-class `runtime_error`. This function returns the string that the `DivideByZeroException` constructor (lines 1213 in Fig. 16.1) passed to the `runtime_error` base-class constructor.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[PREV](#)

[NEXT](#)

[Page 819]

## 16.4. When to Use Exception Handling

Exception handling is designed to process **synchronous errors**, which occur when a statement executes. Common examples of these errors are out-of-range array subscripts, arithmetic overflow (i.e., a value outside the representable range of values), division by zero, invalid function parameters and unsuccessful memory allocation (due to lack of memory). Exception handling is not designed to process errors associated with **asynchronous** events (e.g., disk I/O completions, network message arrivals, mouse clicks and keystrokes), which occur in parallel with, and independent of, the program's flow of control.

Software Engineering Observation 16.3



Incorporate your exception-handling strategy into your system from the design process's inception. Including effective exception handling after a system has been implemented can be difficult.

Software Engineering Observation 16.4



Exception handling provides a single, uniform technique for processing problems. This helps programmers working on large projects understand each other's error-processing code.

Software Engineering Observation 16.5



Avoid using exception handling as an alternate form of flow of control. These "additional" exceptions can "get in the way" of genuine error-type exceptions.

Software Engineering Observation 16.6



Exception handling simplifies combining software components and enables them to work together effectively by enabling predefined components to communicate problems to application-specific components, which can then process the problems in an application-specific manner.

The exception-handling mechanism also is useful for processing problems that occur when a program interacts with software elements, such as member functions, constructors, destructors and classes. Rather than handling problems internally, such software elements often use exceptions to notify programs when problems occur. This enables programmers to implement customized error handling for each application.

### Performance Tip 16.3



When no exceptions occur, exception-handling code incurs little or no performance penalties. Thus, programs that implement exception handling operate more efficiently than do programs that intermix error-handling code with program logic.

### Software Engineering Observation 16.7



Functions with common error conditions should return 0 or `NULL` (or other appropriate values) rather than throw exceptions. A program calling such a function can check the return value to determine success or failure of the function call.

Complex applications normally consist of predefined software components and application-specific components that use the predefined components. When a predefined component encounters a problem, that component needs a mechanism to communicate the problem to the application-specific component—the predefined component cannot know in advance how each application processes a problem that occurs.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 821]

**Figure 16.3. Rethrowing an exception.**

(This item is displayed on pages 820 - 821 in the print version)

```
1 // Fig. 16.3: Fig16_03.cpp
2 // Demonstrating exception rethrowing.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <exception>
8 using std::exception;
9
10 // throw, catch and rethrow exception
11 void throwException()
12 {
13 // throw exception and catch it immediately
14 try
15 {
16 cout << " Function throwException throws an exception\n";
17 throw exception(); // generate exception
18 } // end try
19 catch (exception &) // handle exception
20 {
21 cout << " Exception handled in function throwException"
22 << "\n Function throwException rethrows exception";
23 throw; // rethrow exception for further processing
24 } // end catch
25
26 cout << "This also should not print\n";
27 } // end function throwException
28
29 int main()
30 {
31 // throw exception
32 try
33 {
34 cout << "\nmain invokes function throwException\n";
35 throwException();
36 cout << "This should not print\n";
37 }
```

```
37 } // end try
38 catch (exception &) // handle exception
39 {
40 cout << "\n\nException handled in main\n";
41 } // end catch
42
43 cout << "Program control continues after catch in main\n";
44 return 0;
45 } // end main
```

main invokes function throwException  
Function throwException throws an exception  
Exception handled in function throwException  
Function throwException rethrows exception

Exception handled in main  
Program control continues after catch in main

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 822]

A function that does not provide an exception specification can `throw` any exception. Placing `throw()` an **empty exception specification** after a function's parameter list states that the function does not `throw` exceptions. If the function attempts to `throw` an exception, function `unexpected` is invoked. [Section 16.7](#) shows how function `unexpected` can be customized by calling function `set_unexpected`.

### Common Programming Error 16.7



Throwing an exception that has not been declared in a function's exception specification causes a call to function `unexpected`.

### Error-Prevention Tip 16.3



The compiler will not generate a compilation error if a function contains a `throw` expression for an exception not listed in the function's exception specification. An error occurs only when that function attempts to throw that exception at execution time. To avoid surprises at execution time, carefully check your code to ensure that functions do not throw exceptions not listed in their exception specifications.

[PREV](#)

[NEXT](#)

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 822 (continued)]

## 16.7. Processing Unexpected Exceptions

Function `unexpected` calls the function registered with function `set_unexpected` (defined in header file `<exception>`). If no function has been registered in this manner, function `terminate` is called by default. Cases in which function `terminate` is called include:

1.

the exception mechanism cannot find a matching `catch` for a thrown exception

2.

a destructor attempts to `throw` an exception during stack unwinding

3.

an attempt is made to rethrow an exception when there is no exception currently being handled

4.

a call to function `unexpected` defaults to calling function `terminate`

(Section 15.5.1 of the C++ Standard Document discusses several additional cases.) Function `set_terminate` can specify the function to invoke when `terminate` is called. Otherwise, `terminate` calls `abort`, which terminates the program without calling the destructors of any remaining objects of automatic or static storage class. This could lead to resource leaks when a program terminates prematurely.

Function `set_terminate` and function `set_unexpected` each return a pointer to the last function called by `terminate` and `unexpected`, respectively (0, the first time each is called). This enables the programmer to save the function pointer so it can be restored later. Functions `set_terminate` and `set_unexpected` take as arguments pointers to functions with `void` return types and no arguments.

If the last action of a programmer-defined termination function is not to exit a program, function `abort` will be called to end program execution after the other statements of the programmer-defined termination

function are executed.

 PREV

NEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 824]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 825]

Exceptions thrown by a constructor cause destructors to be called for any objects built as part of the object being constructed before the exception is thrown. Destructors are called for every automatic object constructed in a `try` block before an exception is thrown. Stack unwinding is guaranteed to have been completed at the point that an exception handler begins executing. If a destructor invoked as a result of stack unwinding throws an exception, `terminate` is called.

If an object has member objects, and if an exception is thrown before the outer object is fully constructed, then destructors will be executed for the member objects that have been constructed prior to the occurrence of the exception. If an array of objects has been partially constructed when an exception occurs, only the destructors for the constructed objects in the array will be called.

An exception could preclude the operation of code that would normally release a resource, thus causing a resource leak. One technique to resolve this problem is to initialize a local object to acquire the resource. When an exception occurs, the destructor for that object will be invoked and can free the resource.

#### Error-Prevention Tip 16.4



When an exception is thrown from the constructor for an object that is created in a new expression, the dynamically allocated memory for that object is released.

PREV

page footer

NEXT

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 825 (continued)]

## 16.10. Exceptions and Inheritance

Various exception classes can be derived from a common base class, as we discussed in [Section 16.3](#), when we created class `DivideByZeroException` as a derived class of class `exception`. If a catch handler catches a pointer or reference to an exception object of a base-class type, it also can catch a pointer or reference to all objects of classes publicly derived from that base classthis allows for polymorphic processing of related errors.

### Error-Prevention Tip 16.5



Using inheritance with exceptions enables an exception handler to catch related errors with concise notation. One approach is to catch each type of pointer or reference to a derived-class exception object individually, but a more concise approach is to catch pointers or references to base-class exception objects instead. Also, catching pointers or references to derived-class exception objects individually is error prone, especially if the programmer forgets to test explicitly for one or more of the derived-class pointer or reference types.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 826]

In this section, we present three examples of `new` failing. The first example returns 0 when `new` fails. The second example uses the version of `new` that throws a `bad_alloc` exception when `new` fails. The third example uses function `set_new_handler` to handle `new` failures. [Note: The examples in Figs. 16.516.7 allocate large amounts of dynamic memory, which could cause your computer to become sluggish.]

## new Returning 0 on Failure

Figure 16.5 demonstrates `new` returning 0 on failure to allocate the requested amount of memory. The `for` statement at lines 1324 should loop 50 times and, on each pass, allocate an array of 50,000,000 `double` values (i.e., 400,000,000 bytes, because a `double` is normally 8 bytes). The `if` statement at line 17 tests the result of each `new` operation to determine whether `new` allocated the memory successfully. If `new` fails and returns 0, line 19 prints an error message, and the loop terminates. [Note: We used Microsoft Visual C++ 6.0 to run this example, because Microsoft Visual Studio .NET throws a `bad_alloc` exception on `new` failure instead of returning 0.]

**Figure 16.5. new returning 0 on failure.**

```

1 // Fig. 16.5: Fig16_05.cpp
2 // Demonstrating pre-standard new returning 0 when memory
3 // is not allocated.
4 #include <iostream>
5 using std::cerr;
6 using std::cout;
7
8 int main()
9 {
10 double *ptr[50];
11
12 // allocate memory for ptr
13 for (int i = 0; i < 50; i++)
14 {
15 ptr[i] = new double[50000000];
16
17 if (ptr[i] == 0) // did new fail to allocate memory
18 {
19 cerr << "Memory allocation failed for ptr[" << i << "]\n";
20 break;
21 } // end if
22 } // successful memory allocation

```

```
23 cout << "Allocated 50000000 doubles in ptr[" << i << "]\n";
24 } // end for
25
26 return 0;
27 } // end main
```

```
Allocated 50000000 doubles in ptr[0]
Allocated 50000000 doubles in ptr[1]
Allocated 50000000 doubles in ptr[2]
Memory allocation failed for ptr[3]
```

---

[Page 827]

The output shows that the program performed only three iterations before `new` failed, and the loop terminated. Your output might differ based on the physical memory, disk space available for virtual memory on your system and the compiler you are using.

## **new Throwing bad\_alloc on Failure**

Figure 16.6 demonstrates `new` throwing `bad_alloc` on failure to allocate the requested memory. The `for` statement (lines 2024) inside the `try` block should loop 50 times and, on each pass, allocate an array of 50,000,000 double values. If `new` fails and throws a `bad_alloc` exception, the loop terminates, and the program continues at line 28, where the `catch` handler catches and processes the exception. Lines 3031 print the message "Exception occurred:" followed by the message returned from the base-class-exception version of function `what` (i.e., an implementation-defined exception-specific message, such as "Allocation Failure" in Microsoft Visual Studio .NET 2003). The output shows that the program performed only three iterations of the loop before `new` failed and threw the `bad_alloc` exception. Your output might differ based on the physical memory, disk space available for virtual memory on your system and the compiler you are using.

---

[Page 828]

**Figure 16.6. `new` throwing `bad_alloc` on failure.**

(This item is displayed on page 827 in the print version)

```

1 // Fig. 16.6: Fig16_06.cpp
2 // Demonstrating standard new throwing bad_alloc when memory
3 // cannot be allocated.
4 #include <iostream>
5 using std::cerr;
6 using std::cout;
7 using std::endl;
8
9 #include <new> // standard operator new
10 using std::bad_alloc;
11
12 int main()
13 {
14 double *ptr[50];
15
16 // allocate memory for ptr
17 try
18 {
19 // allocate memory for ptr[i]; new throws bad_alloc on failure
20 for (int i = 0; i < 50; i++)
21 {
22 ptr[i] = new double[50000000]; // may throw exception
23 cout << "Allocated 50000000 doubles in ptr[" << i << "]\n";
24 } // end for
25 } // end try
26
27 // handle bad_alloc exception
28 catch (bad_alloc &memoryAllocationException)
29 {
30 cerr << "Exception occurred: "
31 << memoryAllocationException.what() << endl;
32 } // end catch
33
34 return 0;
35 } // end main

```

```

Allocated 50000000 doubles in ptr[0]
Allocated 50000000 doubles in ptr[1]
Allocated 50000000 doubles in ptr[2]
Exception occurred: bad allocation

```

The C++ standard specifies that standard-compliant compilers can continue to use a version of `new` that returns 0 upon failure. For this purpose, header file `<new>` defines object `nothrow` (of type `nothrow_t`), which is used as follows:

```
double *ptr = new(noexcept) double[50000000];
```

The preceding statement uses the version of `new` that does not throw `bad_alloc` exceptions (i.e., `nothrow`) to allocate an array of 50,000,000 doubles.

## Software Engineering Observation 16.8



To make programs more robust, use the version of `new` that throws `bad_alloc` exceptions on failure.

## Handling new Failures Using Function `set_new_handler`

An additional feature for handling `new` failures is function `set_new_handler` (prototyped in standard header file `<new>`). This function takes as its argument a pointer to a function that takes no arguments and returns `void`. This pointer points to the function that will be called if `new` fails. This provides the programmer with a uniform approach to handling all `new` failures, regardless of where a failure occurs in the program. Once `set_new_handler` registers a `new handler` in the program, operator `new` does not throw `bad_alloc` on failure; rather, it defers the error handling to the `new-handler` function.

If `new` allocates memory successfully, it returns a pointer to that memory. If `new` fails to allocate memory and `set_new_handler` did not register a `new-handler` function, `new` throws a `bad_alloc` exception. If `new` fails to allocate memory and a `new-handler` function has been registered, the `new-handler` function is called. The C++ standard specifies that the `new-handler` function should perform one of the following tasks:

1.

Make more memory available by deleting other dynamically allocated memory (or telling the user to close other applications) and return to operator `new` to attempt to allocate memory again.

2.

Throw an exception of type `bad_alloc`.

3.

Call function `abort` or `exit` (both found in header file `<cstdlib>`) to terminate the program.

**Figure 16.7** demonstrates `set_new_handler`. Function `customNewHandler` (lines 1418) prints an error message (line 16), then terminates the program via a call to `abort` (line 17). The output shows that the program performed only three iterations of the loop before `new` failed and invoked function `customNewHandler`. Your output might differ based on the physical memory, disk space available for virtual memory on your system and the compiler you use to compile the program.

[Page 829]

**Figure 16.7. `set_new_handler` specifying the function to call when `new` fails.**

```

1 // Fig. 16.7: Fig16_07.cpp
2 // Demonstrating set_new_handler.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6
7 #include <new> // standard operator new and set_new_handler
8 using std::set_new_handler;
9
10 #include <cstdlib> // abort function prototype
11 using std::abort;
12
13 // handle memory allocation failure
14 void customNewHandler()
15 {
16 cerr << "customNewHandler was called";
17 abort();
18 } // end function customNewHandler
19
20 // using set_new_handler to handle failed memory allocation
21 int main()
22 {
23 double *ptr[50];
24
25 // specify that customNewHandler should be called on
26 // memory allocation failure
27 set_new_handler(customNewHandler);
28
29 // allocate memory for ptr[i]; customNewHandler will be
30 // called on failed memory allocation
31 for (int i = 0; i < 50; i++)
32 {
33 ptr[i] = new double[50000000]; // may throw exception

```

```
34 cout << "Allocated 50000000 doubles in ptr[" << i << "]\n";
35 } // end for
36
37 return 0;
38 } // end main
```

```
Allocated 50000000 doubles in ptr[0]
Allocated 50000000 doubles in ptr[1]
Allocated 50000000 doubles in ptr[2]
customNewHandler was called
```

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 830]

An object of class `auto_ptr` maintains a pointer to dynamically allocated memory. When an `auto_ptr` object destructor is called (for example, when an `auto_ptr` object goes out of scope), it performs a `delete` operation on its pointer data member. Class template `auto_ptr` provides overloaded operators `*` and `->` so that an `auto_ptr` object can be used just as a regular pointer variable is. [Figure 16.10](#) demonstrates an `auto_ptr` object that points to a dynamically allocated object of class `Integer` ([Figs. 16.8](#)[16.9](#)).

**Figure 16.8. Class Integer definition.**

```
1 // Fig. 16.8: Integer.h
2 // Integer class definition.
3
4 class Integer
5 {
6 public:
7 Integer(int i = 0); // Integer default constructor
8 ~Integer(); // Integer destructor
9 void setInteger(int i); // functions to set Integer
10 int getInteger() const; // function to return Integer
11 private:
12 int value;
13 } // end class Integer
```

**Figure 16.9. Member function definition of class Integer.**

(This item is displayed on pages 830 - 831 in the print version)

```
1 // Fig. 16.9: Integer.cpp
2 // Integer member function definition.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Integer.h"
8
9 // Integer default constructor
10 Integer::Integer(int i)
11 : value(i)
12 {
13 cout << "Constructor for Integer " << value << endl;
14 } // end Integer constructor
15
16 // Integer destructor
17 Integer::~Integer()
18 {
19 cout << "Destructor for Integer " << value << endl;
20 } // end Integer destructor
21
22 // set Integer value
23 void Integer::setInteger(int i)
24 {
25 value = i;
26 } // end function setInteger
27
28 // return Integer value
29 int Integer::getInteger() const
30 {
31 return value;
32 } // end function getInteger
```

**Figure 16.10. auto\_ptr object manages dynamically allocated memory.**

(This item is displayed on pages 831 - 832 in the print version)

```

1 // Fig. 16.10: Fig16_10.cpp
2 // Demonstrating auto_ptr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <memory>
8 using std::auto_ptr; // auto_ptr class definition
9
10 #include "Integer.h"
11
12 // use auto_ptr to manipulate Integer object
13 int main()
14 {
15 cout << "Creating an auto_ptr object that points to an Integer\n";
16
17 // "aim" auto_ptr at Integer object
18 auto_ptr< Integer > ptrToInteger(new Integer(7));
19
20 cout << "\nUsing the auto_ptr to manipulate the Integer\n";
21 ptrToInteger->setInteger(99); // use auto_ptr to set Integer value
22
23 // use auto_ptr to get Integer value
24 cout << "Integer after setInteger: " << (*ptrToInteger).getInteger();
25
26 } // end main

```

Creating an auto\_ptr object that points to an Integer  
Constructor for Integer 7

Using the auto\_ptr to manipulate the Integer  
Integer after setInteger: 99

Terminating program  
Destructor for Integer 99

Line 18 of Fig. 16.10 creates auto\_ptr object ptrToInteger and initializes it with a pointer to a dynamically allocated Integer object that contains the value 7. Line 21 uses the auto\_ptr overloaded -> operator to invoke function setInteger on the Integer object pointed to by ptrToInteger. Line 24 uses the auto\_ptr overloaded \* operator to dereference ptrToInteger, then uses the dot (.) operator to invoke function getInteger on the Integer object pointed to by ptrToInteger. Like a

regular pointer, an `auto_ptr`'s `->` and `*` overloaded operators can be used to access the object to which the `auto_ptr` points.

Because `ptrToInteger` is a local automatic variable in `main`, `ptrToInteger` is destroyed when `main` terminates. The `auto_ptr` destructor forces a `delete` of the `Integer` object pointed to by `ptrToInteger`, which in turn calls the `Integer` class destructor. The memory that `Integer` occupies is released, regardless of how control leaves the block (e.g., by a `return` statement or by an exception). Most importantly, using this technique can prevent memory leaks. For example, suppose a function returns a pointer aimed at some object. Unfortunately, the function caller that receives this pointer might not delete the object, thus resulting in a memory leak. However, if the function returns an `auto_ptr` to the object, the object will be deleted automatically when the `auto_ptr` object's destructor gets called.

---

[Page 832]

An `auto_ptr` can pass ownership of the dynamic memory it manages via its overloaded assignment operator or copy constructor. The last `auto_ptr` object that maintains the pointer to the dynamic memory will delete the memory. This makes `auto_ptr` an ideal mechanism for returning dynamically allocated memory to client code. When the `auto_ptr` goes out of scope in the client code, the `auto_ptr`'s destructor deletes the dynamic memory.

### Software Engineering Observation 16.9



An `auto_ptr` has restrictions on certain operations. For example, an `auto_ptr` cannot point to an array or a standard-container class.

PREV

NEXT

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 833]

Class `logic_error` is the base class of several standard exception classes that indicate errors in program logic. For example, class `invalid_argument` indicates that an invalid argument was passed to a function. (Proper coding can, of course, prevent invalid arguments from reaching a function.) Class `length_error` indicates that a length larger than the maximum size allowed for the object being manipulated was used for that object. Class `out_of_range` indicates that a value, such as a subscript into an array, exceeded its allowed range of values.

Class `runtime_error`, which we used briefly in [Section 16.8](#), is the base class of several other standard exception classes that indicate execution-time errors. For example, class `overflow_error` describes an **arithmetic overflow error** (i.e., the result of an arithmetic operation is larger than the largest number that can be stored in the computer) and class `underflow_error` describes an **arithmetic underflow error** (i.e., the result of an arithmetic operation is smaller than the smallest number that can be stored in the computer).

### Common Programming Error 16.9



Programmer-defined exception classes need not be derived from class `exception`. Thus, writing `catch ( exception anyException )` is not guaranteed to catch all exceptions a program could encounter.

### Error-Prevention Tip 16.6



To catch all exceptions potentially thrown in a `try` block, use `catch( ... )`. One weakness with catching exceptions in this way is that the type of the caught exception is unknown at compile time. Another weakness is that, without a named parameter, there is no way to refer to the exception object inside the exception handler.

### Software Engineering Observation 16.10



The standard exception hierarchy is a good starting point for creating exceptions. Programmers can build programs that can throw standard exceptions, throw exceptions derived from the standard exceptions or `throw` their own exceptions not derived from the standard exceptions.

---

[Page 834]

## Software Engineering Observation 16.11



Use `catch( . . . )` to perform recovery that does not depend on the exception type (e.g., releasing common resources). The exception can be rethrown to alert more specific enclosing `catch` handlers.

[◀ PREV](#)

page footer

[NEXT ▶](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 834 (continued)]

## 16.14. Other Error-Handling Techniques

We have discussed several ways to deal with exceptional situations prior to this chapter. The following summarizes these and other error-handling techniques:

- Ignore the exception. If an exception occurs, the program might fail as a result of the uncaught exception. This is devastating for commercial software products or for special-purpose software designed for mission-critical situations, but, for software developed for your own purposes, ignoring many kinds of errors is common.

### Common Programming Error 16.10



Aborting a program component due to an uncaught exception could leave a resources such as a file stream or an I/O device in a state in which other programs are unable to acquire the resource. This is known as a "resource leak."

- Abort the program. This, of course, prevents a program from running to completion and producing incorrect results. For many types of errors, this is appropriate, especially for nonfatal errors that enable a program to run to completion (potentially misleading the programmer to think that the program functioned correctly). This strategy is inappropriate for mission-critical applications. Resource issues also are important here. If a program obtains a resource, the program should release that resource before program termination.
- Set error indicators. The problem with this approach is that programs might not check these error indicators at all points at which the errors could be troublesome.
- Test for the error condition, issue an error message and call `exit` (in `<cstdlib>`) to pass an appropriate error code to the program's environment.
- Use functions `setjmp` and `longjmp`. These `<csetjmp>` library functions enable the programmer to specify an immediate jump from a deeply nested function call to an error handler. Without using `setjmp` or `longjmp`, a program must execute several returns to exit the deeply nested function calls. Functions `setjmp` and `longjmp` are dangerous, because they unwind the stack without calling destructors for automatic objects. This can lead to serious problems.
- Certain specific kinds of errors have dedicated capabilities for handling them. For example, when operator `new` fails to allocate memory, it can cause a `new_handler` function to execute to handle the error. This function can be customized by supplying a function name as the argument to `set_new_handler`, as we discuss in [Section 16.11](#).

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 835]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 836]

- Exception handling is not designed to process errors associated with asynchronous events, which occur in parallel with, and independent of, the program's flow of control.
- To throw an exception, use keyword `throw` followed by an operand that represents the type of exception to throw. Normally, a `throw` statement specifies one operand.
- The operand of a `throw` can be of any type.
- The exception handler can defer the exception handling (or perhaps a portion of it) to another exception handler. In either case, the handler achieves this by rethrowing the exception.
- Common examples of exceptions are out-of-range array subscripts, arithmetic overflow, division by zero, invalid function parameters and unsuccessful memory allocations.
- Class `exception` is the standard C++ base class for exceptions. Class `exception` provides virtual function `what` that returns an appropriate error message and can be overridden in derived classes.
- An optional exception specification enumerates a list of exceptions that a function can throw. A function can throw only exceptions of the types indicated by the exception specification or exceptions of any type derived from these types. If the function throws an exception that does not belong to a specified type, function `unexpected` is called and the program normally terminates.
- A function with no exception specification can throw any exception. The empty exception specification `throw()` indicates that a function does not throw exceptions. If a function with an empty exception specification attempts to throw an exception, function `unexpected` is invoked.
- Function `unexpected` calls the function registered with function `set_unexpected`. If no function has been registered in this manner, function `terminate` is called by default.
- Function `set_terminate` can specify the function to invoke when `terminate` is called. Otherwise, `terminate` calls `abort`, which terminates the program without calling the destructors of objects that are declared `static` and `auto`.
- Functions `set_terminate` and `set_unexpected` each return a pointer to the last function called by `terminate` and `unexpected`, respectively (0, the first time each is called). This enables the programmer to save the function pointer so it can be restored later.
- Functions `set_terminate` and `set_unexpected` take as arguments pointers to functions with `void` return types and no arguments.
- If a programmer-defined termination function does not exit a program, function `abort` will be called after the programmer-defined termination function completes execution.
- Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function.
- Class `runtime_error` (defined in header `<stdexcept>`) is the C++ standard base class for representing runtime errors.
- Exceptions thrown by a constructor cause destructors to be called for any objects built as part of the object being constructed before the exception is thrown.
- Destructors are called for every automatic object constructed in a `try` block before an exception is thrown.
- Stack unwinding completes before an exception handler begins executing.
- If a destructor invoked as a result of stack unwinding throws an exception, `terminate` is called.
- If an object has member objects, and if an exception is thrown before the outer object is fully

constructed, then destructors will be executed for the member objects that have been constructed before the exception occurs.

[Page 837]

- If an array of objects has been partially constructed when an exception occurs, only the destructors for the constructed array element objects will be called.
- When an exception is thrown from the constructor for an object that is created in a `new` expression, the dynamically allocated memory for that object is released.
- If a catch handler catches a pointer or reference to an exception object of a base-class type, it also can catch a pointer or reference to all objects of classes derived publicly from that base class—this allows for polymorphic processing of related errors.
- The C++ standard document specifies that, when operator `new` fails, it throws a `bad_alloc` exception (defined in header file `<new>`).
- Function `set_new_handler` takes as its argument a pointer to a function that takes no arguments and returns `void`. This pointer points to the function that will be called if `new` fails.
- Once `set_new_handler` registers a new handler in the program, operator `new` does not throw `bad_alloc` on failure; rather, it defers the error handling to the `new_handler` function.
- If `new` allocates memory successfully, it returns a pointer to that memory.
- If an exception occurs after successful memory allocation but before the `delete` statement executes, a memory leak could occur.
- The C++ Standard Library provides class template `auto_ptr` to deal with memory leaks.
- An object of class `auto_ptr` maintains a pointer to dynamically allocated memory. An `auto_ptr` object's destructor performs a `delete` operation on the `auto_ptr`'s pointer data member.
- Class template `auto_ptr` provides overloaded operators `*` and `->` so that an `auto_ptr` object can be used just as a regular pointer variable is. An `auto_ptr` also transfers ownership of the dynamic memory it manages via its copy constructor and overloaded assignment operator.
- The C++ Standard Library includes a hierarchy of exception classes. This hierarchy is headed by base-class `exception`.
- Immediate derived classes of base class `exception` include `runtime_error` and `logic_error` (both defined in header `<stdexcept>`), each of which has several derived classes.
- Several operators throw standard exceptions: operator `new` throws `bad_alloc`, operator `dynamic_cast` throws `bad_cast` and operator `typeid` throws `bad_typeid`.
- Including `bad_exception` in the throw list of a function means that, if an unexpected exception occurs, function `unexpected` can throw `bad_exception` rather than terminating the program's execution or calling another function specified by `set_unexpected`.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 838]

`length_error` exception

`logic_error` exception

`<memory>` header file

`new` failure handler

`nothrow` object

`out_of_range` exception

`overflow_error` exception

resumption model of exception handling

rethrow an exception

robust application

`runtime_error` exception

`set_new_handler` function

`set_terminate` function

`set_unexpected` function

stack unwinding

`<stdexcept>` header file

synchronous errors

terminate function

termination model of exception handling

throw an exception

tHRow an unexpected exception

tHRow keyword

tHRow list

tHRow without arguments

throw point

TRY block

TRY keyword

underflow\_error exception

unexpected function

what virtual function of class exception

 PREV

NEXT 

page footer

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 838 (continued)]

## Self-Review Exercises

- 16.1** List five common examples of exceptions.
- 16.2** Give several reasons why exception-handling techniques should not be used for conventional program control.
- 16.3** Why are exceptions appropriate for dealing with errors produced by library functions?
- 16.4** What is a "resource leak"?
- 16.5** If no exceptions are thrown in a `TRY` block, where does control proceed to after the `try` block completes execution?
- 16.6** What happens if an exception is thrown outside a `TRY` block?
- 16.7** Give a key advantage and a key disadvantage of using `catch( . . . )`.
- 16.8** What happens if no `catch` handler matches the type of a thrown object?
- 16.9** What happens if several handlers match the type of the thrown object?
- 16.10** Why would a programmer specify a base-class type as the type of a `catch` handler, then `throw` objects of derived-class types?
- 16.11** Suppose a `catch` handler with a precise match to an exception object type is available. Under what circumstances might a different handler be executed for exception objects of that type?
- 16.12** Must throwing an exception cause program termination?
- 16.13** What happens when a `catch` handler `throws` an exception?
- 16.14** What does the statement `throw;` do?

- 16.15** How does the programmer restrict the exception types that a function can throw?
- 16.16** What happens if a function throws an exception of a type not allowed by the exception specification for the function?
- 16.17** What happens to the automatic objects that have been constructed in a TRY block when that block throws an exception?

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 839]

## 16.2

- (a) Exception handling is designed to handle infrequently occurring situations that often result in program termination, so compiler writers are not required to implement exception handling to perform optimally.
- (b) Flow of control with conventional control structures generally is clearer and more efficient than with exceptions.
- (c) Problems can occur because the stack is unwound when an exception occurs and resources allocated prior to the exception might not be freed.
- (d) The "additional" exceptions make it more difficult for the programmer to handle the larger number of exception cases.

## 16.3

It is unlikely that a library function will perform error processing that will meet the unique needs of all users.

## 16.4

A program that terminates abruptly could leave a resource in a state in which other programs would not be able to acquire the resource, or the program itself might not be able to reacquire a "leaked" resource.

## 16.5

The exception handlers (in the `catch` handlers) for that `TRY` block are skipped, and the program resumes execution after the last `catch` handler.

## 16.6

An exception thrown outside a `try` block causes a call to `terminate`.

## 16.7

The form `catch( . . . )` catches any type of exception thrown in a `try` block. An advantage is that all possible exceptions will be caught. A disadvantage is that the `catch` has no parameter, so it cannot reference information in the thrown object and cannot know the cause of the exception.

## 16.8

This causes the search for a match to continue in the next enclosing `TRY` block if there is one. As this process continues, it might eventually be determined that there is no handler in the program that matches

the type of the thrown object; in this case, `terminate` is called, which by default calls `abort`. An alternative `terminate` function can be provided as an argument to `set_terminate`.

## 16.9

The first matching exception handler after the `try` block is executed.

## 16.10

This is a nice way to catch related types of exceptions.

## 16.11

A base-class handler would catch objects of all derived-class types.

## 16.12

No, but it does terminate the block in which the exception is thrown.

## 16.13

The exception will be processed by a `catch` handler (if one exists) associated with the `try` block (if one exists) enclosing the `catch` handler that caused the exception.

## 16.14

It rethrows the exception if it appears in a `catch` handler; otherwise, function `unexpected` is called.

## 16.15

Provide an exception specification listing the exception types that the function can throw.

## 16.16

Function `unexecpted` is called.

## 16.17

The `try` block expires, causing destructors to be called for each of these objects.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 840]

### 16.20

A program contains the statement

`throw;`

Where would you normally expect to find such a statement? What if that statement appeared in a different part of the program?

### 16.21

Compare and contrast exception handling with the various other error-processing schemes discussed in the text.

### 16.22

Why should exceptions not be used as an alternate form of program control?

### 16.23

Describe a technique for handling related exceptions.

### 16.24

Until this chapter, we have found that dealing with errors detected by constructors can be awkward. Exception handling gives us a better means of handling such errors. Consider a constructor for a `String` class. The constructor uses `new` to obtain space from the free store. Suppose `new` fails. Show how you would deal with this without exception handling. Discuss the key issues. Show how you would deal with such memory exhaustion with exception handling. Explain why the exception-handling approach is superior.

### 16.25

Suppose a program `throws` an exception and the appropriate exception handler begins executing. Now suppose that the exception handler itself `throws` the same exception. Does this create infinite recursion? Write a program to check your observation.

### 16.26

Use inheritance to create various derived classes of `runtime_error`. Then show that a `catch` handler specifying the base class can catch derived-class exceptions.

### 16.27

Write a conditional expression that returns either a `double` or an `int`. Provide an `int` `catch` handler and a `double` `catch` handler. Show that only the `double` `catch` handler executes, regardless of whether the `int` or the `double` is returned.

### 16.28

Write a program that generates and handles a memory-exhaustion exception. Your program should loop on a request to create dynamic memory through operator `new`.

### 16.29

Write a program illustrating that all destructors for objects constructed in a block are called before an exception is thrown from that block.

### 16.30

Write a program illustrating that member object destructors are called for only those member objects that were constructed before an exception occurred.

### 16.31

Write a program that demonstrates several exception types being caught with the `catch( . . . )` exception handler.

### 16.32

Write a program illustrating that the order of exception handlers is important. The first matching handler is the one that executes. Attempt to compile and run your program two different ways to show that two different handlers execute with two different effects.

### 16.33

Write a program that shows a constructor passing information about constructor failure to an exception handler after a `try` block.

### 16.34

Write a program that illustrates rethrowing an exception.

**16.35**

Write a program that illustrates that a function with its own `try` block does not have to catch every possible error generated within the `TRY`. Some exceptions can slip through to, and be handled in, outer scopes.

**16.36**

Write a program that throws an exception from a deeply nested function and still has the `catch` handler following the `TRY` block enclosing the call chain catch the exception.

 PREVNEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 842]

## Outline

[17.1 Introduction](#)

[17.2 The Data Hierarchy](#)

[17.3 Files and Streams](#)

[17.4 Creating a Sequential File](#)

[17.5 Reading Data from a Sequential File](#)

[17.6 Updating Sequential Files](#)

[17.7 Random-Access Files](#)

[17.8 Creating a Random-Access File](#)

[17.9 Writing Data Randomly to a Random-Access File](#)

[17.10 Reading from a Random-Access File Sequentially](#)

[17.11 Case Study: A Transaction-Processing Program](#)

[17.12 Input/Output of Objects](#)

[17.13 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

## Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 842 (continued)]

## 17.1. Introduction

Storage of data in variables and arrays is temporary. **Files** are used for **data persistence** permanent retention of large amounts of data. Computers store files on **secondary storage devices**, such as magnetic disks, optical disks and tapes. In this chapter, we explain how to build C++ programs that create, update and process data files. We consider both sequential files and random-access files. We compare formatted-data file processing and raw-data file processing. We examine techniques for input of data from, and output of data to, **string streams** rather than files in [Chapter 18](#).

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

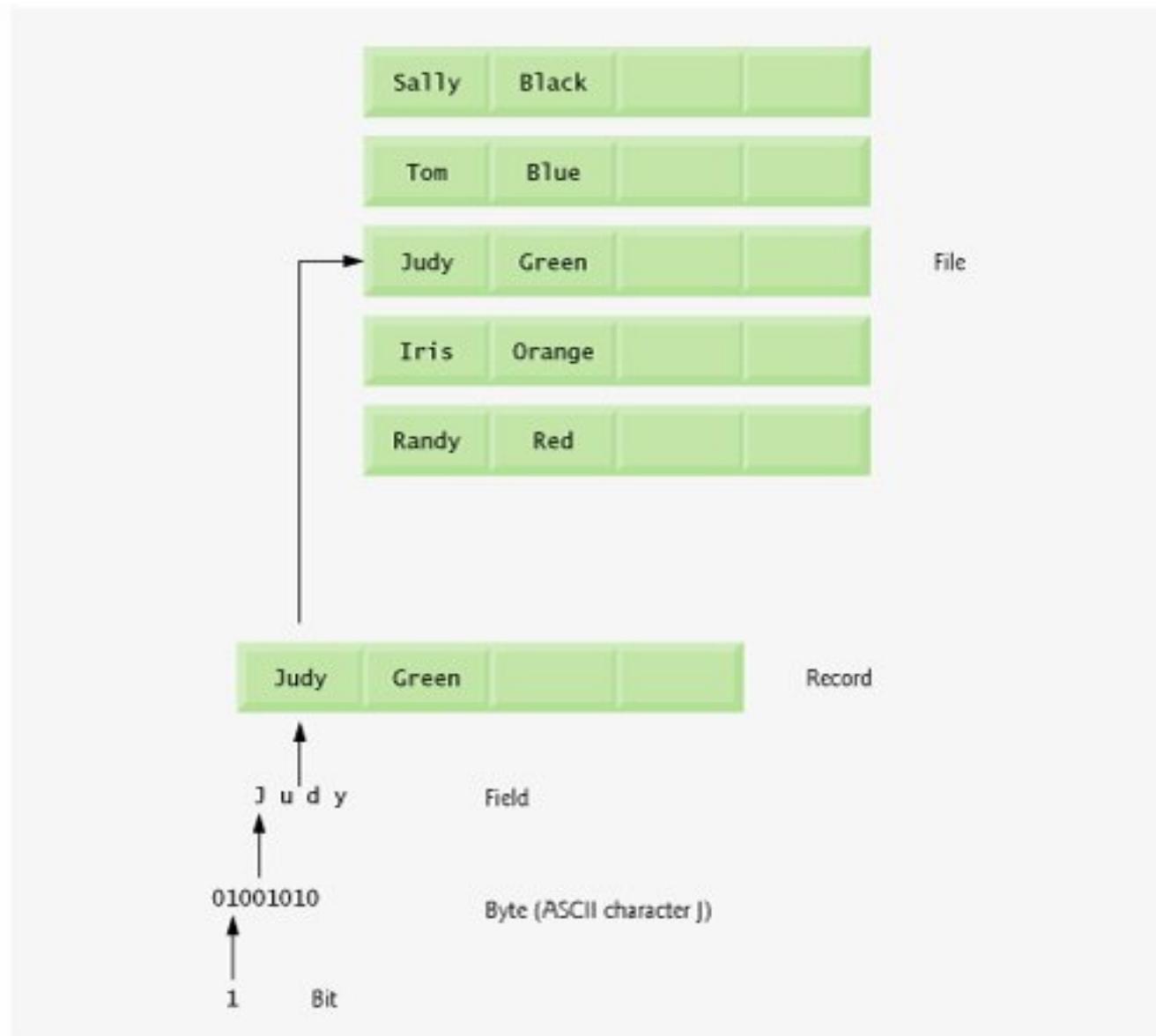
[Page 843]

Just as characters are composed of bits, **fields** are composed of characters. A field is a group of characters that conveys some meaning. For example, a field consisting of uppercase and lowercase letters can represent a person's name.

Data items processed by computers form a **data hierarchy** (Fig. 17.1), in which data items become larger and more complex in structure as we progress from bits, to characters, to fields and to larger data aggregates.

**Figure 17.1. Data hierarchy.**

[\[View full size image\]](#)



Typically, a **record** (which can be represented as a **class** in C++) is composed of several fields (called data members in C++). In a payroll system, for example, a record for a particular employee might include the following fields:

1.

Employee identification number

2.

Name

3.

Address

4.

Hourly pay rate

5.

Number of exemptions claimed

6.

Year-to-date earnings

7.

Amount of taxes withheld

Thus, a record is a group of related fields. In the preceding example, each field is associated with the same employee. A file is a group of related records.<sup>[1]</sup> A company's payroll file normally contains one record for each employee. Thus, a payroll file for a small company might contain only 22 records, whereas one for a large company might contain 100,000 records. It is not unusual for a company to have many files, some containing millions, billions, or even trillions of characters of information.

<sup>[1]</sup> Generally, a file can contain arbitrary data in arbitrary formats. In some operating systems, a file is viewed as nothing more than a collection of bytes. In such an operating system, any organization of the bytes in a file (such as organizing the data into records) is a view created by the application programmer.

To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a **record key**. A record key identifies a record as belonging to a particular person or entity and distinguishes that record from all others. In the payroll record described previously, the employee identification number normally would be chosen as the record key.

There are many ways of organizing records in a file. A common type of organization is called a **sequential file**, in which records typically are stored in order by a record-key field. In a payroll file, records usually are placed in order by employee identification number. The first employee record in the file contains the lowest employee identification number, and subsequent records contain increasingly higher ones.

Most businesses use many different files to store data. For example, a company might have payroll files, accounts-receivable files (listing money due from clients), accounts-payable files (listing money due to suppliers), inventory files (listing facts about all the items handled by the business) and many other types

of files. A group of related files often are stored in a **database**. A collection of programs designed to create and manage databases is called a **database management system (DBMS)**.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

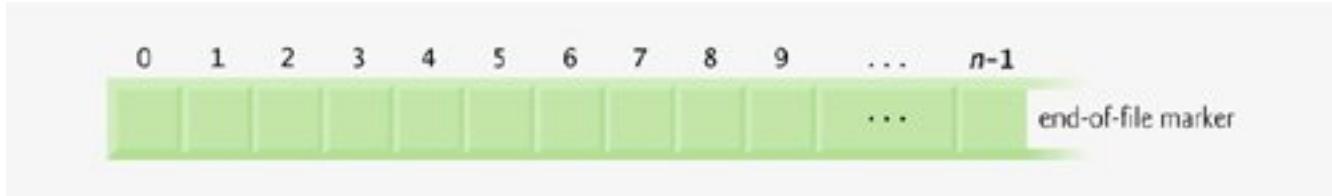
Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 845]

**Figure 17.2. C++'s view of a file of n bytes.**

(This item is displayed on page 844 in the print version)

[\[View full size image\]](#)

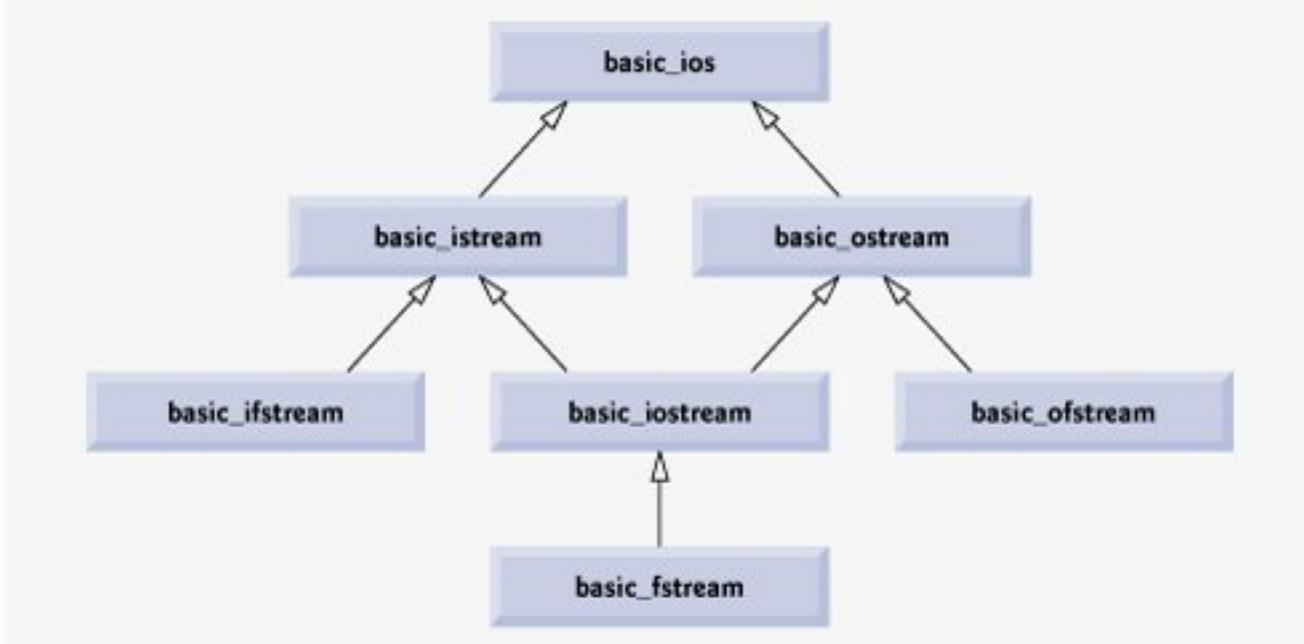


To perform file processing in C++, header files `<iostream>` and `<fstream>` must be included. Header `<fstream>` includes the definitions for the stream class templates `basic_ifstream` (for file input), `basic_ofstream` (for file output) and `basic_fstream` (for file input and output). Each class template has a predefined template specialization that enables `char` I/O. In addition, the `fstream` library provides a set of `typedefs` that provide aliases for these template specializations. For example, the `typedef ifstream` represents a specialization of `basic_ifstream` that enables `char` input from a file. Similarly, `typedef ofstream` represents a specialization of `basic_ofstream` that enables `char` output to files. Also, `typedef fstream` represents a specialization of `basic_fstream` that enables `char` input from, and output to, files.

Files are opened by creating objects of these stream template specializations. These templates "derive" from class templates `basic_istream`, `basic_ostream` and `basic_iostream`, respectively. Thus, all member functions, operators and manipulators that belong to these templates (which we described in [Chapter 15](#)) also can be applied to file streams. [Figure 17.3](#) summarizes the inheritance relationships of the I/O classes that we have discussed to this point.

**Figure 17.3. Portion of stream I/O template hierarchy.**

[\[View full size image\]](#)



[◀ PREV](#)

page footer

[NEXT ▶](#)

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 846]

**Figure 17.4** creates a sequential file that might be used in an accounts-receivable system to help manage the money owed by a company's credit clients. For each client, the program obtains the client's account number, name and balance (i.e., the amount the client owes the company for goods and services received in the past). The data obtained for each client constitutes a record for that client. The account number serves as the record key in this application; that is, the program creates and maintains the file in account number order. This program assumes the user enters the records in account number order. In a comprehensive accounts receivable system, a sorting capability would be provided for the user to enter records in any orderthe records then would be sorted and written to the file.

[Page 847]

#### **Figure 17.4. Creating a sequential file.**

(This item is displayed on pages 846 - 847 in the print version)

```
1 // Fig. 17.4: Fig17_04.cpp
2 // Create a sequential file.
3 #include <iostream>
4 using std::cerr;
5 using std::cin;
6 using std::cout;
7 using std::endl;
8 using std::ios;
9
10 #include <fstream> // file stream
11 using std::ofstream; // output file stream
12
13 #include <cstdlib>
14 using std::exit; // exit function prototype
15
16 int main()
17 {
18 // ofstream constructor opens file
19 ofstream outClientFile("clients.dat", ios::out);
20
21 // exit program if unable to create file
22 if (!outClientFile) // overloaded ! operator
23 {
24 cerr << "File could not be opened" << endl;
```

```

25 exit(1);
26 } // end if
27
28 cout << "Enter the account, name, and balance." << endl
29 << "Enter end-of-file to end input.\n? ";
30
31 int account;
32 char name[30];
33 double balance;
34
35 // read account, name and balance from cin, then place in file
36 while (cin >> account >> name >> balance)
37 {
38 outFile << account << ' ' << name << ' ' << balance << endl;
39 cout << "? ";
40 } // end while
41
42 return 0; // ofstream destructor closes file
43 } // end main

```

```

Enter the account, name, and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z

```

Let us examine this program. As stated previously, files are opened by creating `ifstream`, `ofstream` or `fstream` objects. In Fig. 17.4, the file is to be opened for output, so an `ofstream` object is created.

Two arguments are passed to the object's constructor—the **filename** and the **file-open mode** (line 19). For an `ofstream` object, the file-open mode can be either `ios::out` to output data to a file or `ios::app` to append data to the end of a file (without modifying any data already in the file). Existing files opened with mode `ios::out` are **truncated**; all data in the file is discarded. If the specified file does not yet exist, then `ofstream` creates the file, using that filename.

Line 19 creates an `ofstream` object named `outClientFile` associated with the file `clients.dat` that is opened for output. The arguments "`clients.dat`" and `ios::out` are passed to the `ofstream` constructor, which opens the file. This establishes a "line of communication" with the file. By default,

`ofstream` objects are opened for output, so line 19 could have executed the statement

```
ofstream outClientFile("clients.dat");
```

to open `clients.dat` for output. [Figure 17.5](#) lists the file-open modes.

**Figure 17.5. File open modes.**

| Mode                     | Description                                                                                                                                |
|--------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ios::app</code>    | Append all output to the end of the file.                                                                                                  |
| <code>ios::ate</code>    | Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file. |
| <code>ios::in</code>     | Open a file for input.                                                                                                                     |
| <code>ios::out</code>    | Open a file for output.                                                                                                                    |
| <code>ios::trunc</code>  | Discard the file's contents if they exist (this also is the default action for <code>ios::out</code> ).                                    |
| <code>ios::binary</code> | Open a file for binary (i.e., nontext) input or output.                                                                                    |

### Common Programming Error 17.1



Use caution when opening an existing file for output (`ios::out`), especially when you want to preserve the file's contents, which will be discarded without warning.

---

[Page 848]

An `ofstream` object can be created without opening a specific file; a file can be attached to the object later. For example, the statement

```
ofstream outClientFile;
```

creates an `ofstream` object named `outClientFile`. The `ofstream` member function `open` opens a

file and attaches it to an existing `ofstream` object as follows:

```
outClientFile.open("clients.dat", ios::out);
```

### Common Programming Error 17.2



Not opening a file before attempting to reference it in a program will result in an error.

After creating an `ofstream` object and attempting to open it, the program tests whether the open operation was successful. The `if` statement at lines 2226 uses the overloaded `ios` operator member function `operator!` to determine whether the `open` operation succeeded. The condition returns a `true` value if either the `failbit` or the `badbit` is set for the stream on the `open` operation. Some possible errors are attempting to open a nonexistent file for reading, attempting to open a file for reading or writing without permission and opening a file for writing when no disk space is available.

If the condition indicates an unsuccessful attempt to open the file, line 24 outputs the error message "File could not be opened," and line 25 invokes function `exit` to terminate the program. The argument to `exit` is returned to the environment from which the program was invoked. Argument 0 indicates that the program terminated normally; any other value indicates that the program terminated due to an error. The calling environment (most likely the operating system) uses the value returned by `exit` to respond appropriately to the error.

Another overloaded `ios` operator member function `operator void *` converts the stream to a pointer, so it can be tested as 0 (i.e., the null pointer) or nonzero (i.e., any other pointer value). When a pointer value is used as a condition, C++ converts a null pointer to the `bool` value `false` and converts a non-null pointer to the `bool` value `TRUE`. If the `failbit` or `badbit` (see [Chapter 15](#)) has been set for the stream, 0 (`false`) is returned. The condition in the `while` statement of lines 3640 invokes the `operator void *` member function on `cin` implicitly. The condition remains `TRUE` as long as neither the `failbit` nor the `badbit` has been set for `cin`. Entering the end-of-file indicator sets the `failbit` for `cin`. The `operator void *` function can be used to test an input object for end-of-file instead of calling the `eof` member function explicitly on the input object.

If line 19 opened the file successfully, the program begins processing data. Lines 2829 prompt the user to enter either the various fields for each record or the end-of-file indicator when data entry is complete. [Figure 17.6](#) lists the keyboard combinations for entering end-of-file for various computer systems.

**Figure 17.6. End-of-file key combinations for various popular computer systems.**

(This item is displayed on page 849 in the print version)

| Computer system     | Keyboard combination                            |
|---------------------|-------------------------------------------------|
| UNIX/Linux/Mac OS X | <ctrl-d> (on a line by itself)                  |
| Microsoft Windows   | <ctrl-z> (sometimes followed by pressing Enter) |
| VAX (VMS)           | <ctrl-z>                                        |

Line 36 extracts each set of data and determines whether end-of-file has been entered. When end-of-file is encountered or bad data is entered, operator `void *` returns the null pointer (which converts to the `bool` value `false`) and the `while` statement terminates. The user enters end-of-file to inform the program to process no additional data. The end-of-file indicator is set when the user enters the end-of-file key combination. The `while` statement loops until the end-of-file indicator is set.

---

[Page 849]

Line 38 writes a set of data to the file `clients.dat`, using the stream insertion operator `<<` and the `outClientFile` object associated with the file at the beginning of the program. The data may be retrieved by a program designed to read the file (see [Section 17.5](#)). Note that, because the file created in [Fig. 17.4](#) is simply a text file, it can be viewed by any text editor.

Once the user enters the end-of-file indicator, `main` terminates. This invokes the `outClientFile` object's destructor function implicitly, which closes the `clients.dat` file. The programmer also can close the `ofstream` object explicitly, using member function `close` in the statement

```
outClientFile.close();
```

### Performance Tip 17.1



Closing files explicitly when the program no longer needs to reference them can reduce resource usage (especially if the program continues execution after closing the files).

In the sample execution for the program of [Fig. 17.4](#), the user enters information for five accounts, then signals that data entry is complete by entering end-of-file (^Z is displayed for Microsoft Windows). This dialog window does not show how the data records appear in the file. To verify that the program created the file successfully, the next section shows how to create a program that reads this file and prints its

contents.

 PREV

NEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 851]

Objects of class `ifstream` are opened for input by default. We could have used the statement

```
ifstream inClientFile("clients.dat");
```

to open `clients.dat` for input. Just as with an `ofstream` object, an `ifstream` object can be created without opening a specific file, because a file can be attached to it later.

The program uses the condition `!inClientFile` to determine whether the file was opened successfully before attempting to retrieve data from the file. Line 48 reads a set of data (i.e., a record) from the file. After the preceding line is executed the first time, `account` has the value 100, `name` has the value "Jones" and `balance` has the value 24.98. Each time line 48 executes, it reads another record from the file into the variables `account`, `name` and `balance`. Line 49 displays the records, using function `outputLine` (lines 5559), which uses parameterized stream manipulators to format the data for display. When the end of file has been reached, the implicit call to operator `void *` in the `while` condition returns the null pointer (which converts to the `bool` value `false`), the `ifstream` destructor function closes the file and the program terminates.

To retrieve data sequentially from a file, programs normally start reading from the beginning of the file and read all the data consecutively until the desired data is found. It might be necessary to process the file sequentially several times (from the beginning of the file) during the execution of a program. Both `istream` and `ostream` provide member functions for repositioning the **file-position pointer** (the byte number of the next byte in the file to be read or written). These member functions are `seekg` ("seek get") for `istream` and `seekp` ("seek put") for `ostream`. Each `istream` object has a "get pointer," which indicates the byte number in the file from which the next input is to occur, and each `ostream` object has a "put pointer," which indicates the byte number in the file at which the next output should be placed. The statement

```
inClientFile.seekg(0);
```

repositions the file-position pointer to the beginning of the file (location 0) attached to `inClientFile`. The argument to `seekg` normally is a `long` integer. A second argument can be specified to indicate the **seek direction**. The seek direction can be `ios::beg` (the default) for positioning relative to the beginning of a stream, `ios::cur` for positioning relative to the current position in a stream or `ios::end` for positioning relative to the end of a stream. The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location (this is also referred to as the **offset** from the beginning of the file). Some examples of positioning the "get" file-position pointer are

[Page 852]

```

// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg(n);

// position n bytes forward in fileObject
fileObject.seekg(n, ios::cur);

// position n bytes back from end of fileObject
fileObject.seekg(n, ios::end);

// position at end of fileObject
fileObject.seekg(0, ios::end);

```

The same operations can be performed using `ostream` member function `seekp`. Member functions `tellg` and `tellp` are provided to return the current locations of the "get" and "put" pointers, respectively. The following statement assigns the "get" file-position pointer value to variable `location` of type `long`:

```
location = fileObject.tellg();
```

**Figure 17.8** enables a credit manager to display the account information for those customers with zero balances (i.e., customers who do not owe the company any money), credit (negative) balances (i.e., customers to whom the company owes money), and debit (positive) balances (i.e., customers who owe the company money for goods and services received in the past). The program displays a menu and allows the credit manager to enter one of three options to obtain credit information. Option 1 produces a list of accounts with zero balances. Option 2 produces a list of accounts with credit balances. Option 3 produces a list of accounts with debit balances. Option 4 terminates program execution. Entering an invalid option displays the prompt to enter another choice.

### Figure 17.8. Credit inquiry program.

(This item is displayed on pages 852 - 855 in the print version)

```

1 // Fig. 17.8: Fig17_08.cpp
2 // Credit inquiry program.
3 #include <iostream>
4 using std::cerr;
5 using std::cin;
6 using std::cout;
7 using std::endl;
8 using std::fixed;
9 using std::ios;
10 using std::left;
11 using std::right;
12 using std::showpoint;
13
14 #include <fstream>

```

```

15 using std::ifstream;
16
17 #include <iomanip>
18 using std::setw;
19 using std::setprecision;
20
21 #include <string>
22 using std::string;
23
24 #include <cstdlib>
25 using std::exit; // exit function prototype
26
27 enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE, DEBIT_BALANCE, END };
28 int getRequest();
29 bool shouldDisplay(int, double);
30 void outputLine(int, const string, double);
31
32 int main()
33 {
34 // ifstream constructor opens the file
35 ifstream inClientFile("clients.dat", ios::in);
36
37 // exit program if ifstream could not open file
38 if (!inClientFile)
39 {
40 cerr << "File could not be opened" << endl;
41 exit(1);
42 } // end if
43
44 int request;
45 int account;
46 char name[30];
47 double balance;
48
49 // get user's request (e.g., zero, credit or debit balance)
50 request = getRequest();
51
52 // process user's request
53 while (request != END)
54 {
55 switch (request)
56 {
57 case ZERO_BALANCE:
58 cout << "\nAccounts with zero balances:\n";
59 break;
60 case CREDIT_BALANCE:
61 cout << "\nAccounts with credit balances:\n";
62 break;
63 case DEBIT_BALANCE:
64 cout << "\nAccounts with debit balances:\n";

```

```

65 break;
66 } // end switch
67
68 // read account, name and balance from file
69 inClientFile >> account >> name >> balance;
70
71 // display file contents (until eof)
72 while (!inClientFile.eof())
73 {
74 // display record
75 if (shouldDisplay(request, balance))
76 outputLine(account, name, balance);
77
78 // read account, name and balance from file
79 inClientFile >> account >> name >> balance;
80 } // end inner while
81
82 inClientFile.clear(); // reset eof for next input
83 inClientFile.seekg(0); // reposition to beginning of file
84 request = getRequest(); // get additional request from user
85 } // end outer while
86
87 cout << "End of run." << endl;
88 return 0; // ifstream destructor closes the file
89 } // end main
90
91 // obtain request from user
92 int getRequest()
93 {
94 int request; // request from user
95
96 // display request options
97 cout << "\nEnter request" << endl
98 << " 1 - List accounts with zero balances" << endl
99 << " 2 - List accounts with credit balances" << endl
100 << " 3 - List accounts with debit balances" << endl
101 << " 4 - End of run" << fixed << showpoint;
102
103 do // input user request
104 {
105 cout << "\n? ";
106 cin >> request;
107 } while (request < ZERO_BALANCE && request > END);
108
109 return request;
110 } // end function getRequest
111
112 // determine whether to display given record
113 bool shouldDisplay(int type, double balance)
114 {

```

```

115 // determine whether to display zero balances
116 if (type == ZERO_BALANCE && balance == 0)
117 return true;
118
119 // determine whether to display credit balances
120 if (type == CREDIT_BALANCE && balance < 0)
121 return true;
122
123 // determine whether to display debit balances
124 if (type == DEBIT_BALANCE && balance > 0)
125 return true;
126
127 return false;
128 } // end function shouldDisplay
129
130 // display single record from file
131 void outputLine(int account, const string name, double balance)
132 {
133 cout << left << setw(10) << account << setw(13) << name
134 << setw(7) << setprecision(2) << right << balance << endl;
135 } // end function outputLine

```

Enter request  
 1 - List accounts with zero balances  
 2 - List accounts with credit balances  
 3 - List accounts with debit balances  
 4 - End of run  
 ? 1

Accounts with zero balances:  
 300 White 0.00

Enter request  
 1 - List accounts with zero balances  
 2 - List accounts with credit balances  
 3 - List accounts with debit balances  
 4 - End of run  
 ? 2

Accounts with credit balances:  
 400 Stone -42.16

Enter request  
 1 - List accounts with zero balances  
 2 - List accounts with credit balances  
 3 - List accounts with debit balances  
 4 - End of run  
 ? 3

```
Accounts with debit balances:
100 Jones 24.98
200 Doe 345.67
500 Rich 224.62

Enter request
1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - End of run
? 4
End of run.
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

PREV

NEXT

[Page 856]

## 17.6. Updating Sequential Files

Data that is formatted and written to a sequential file as shown in [Section 17.4](#) cannot be modified without the risk of destroying other data in the file. For example, if the name "White" needs to be changed to "Worthington," the old name cannot be overwritten without corrupting the file. The record for White was written to the file as

300 White 0.00

If this record were rewritten beginning at the same location in the file using the longer name, the record would be

300 Worthington 0.00

The new record contains six more characters than the original record. Therefore, the characters beyond the second "o" in "Worthington" would overwrite the beginning of the next sequential record in the file. The problem is that, in the formatted input/output model using the stream insertion operator `<<` and the stream extraction operator `>>`, fields and hence records can vary in size. For example, values 7, 14, 117, 2074, and 27383 are all `ints`, which store the same number of "raw data" bytes internally (typically four bytes on today's popular 32-bit machines). However, these integers become different-sized fields when output as formatted text (character sequences). Therefore, the formatted input/output model usually is not used to update records in place.

Such updating can be done awkwardly. For example, to make the preceding name change, the records before 300 White 0.00 in a sequential file could be copied to a new file, the updated record then would be written to the new file, and the records after 300 White 0.00 would be copied to the new file. This requires processing every record in the file to update one record. If many records are being updated in one pass of the file, though, this technique can be acceptable.

PREV

NEXT

page footer

The CHM file was converted to HTML by **chm2web** software.

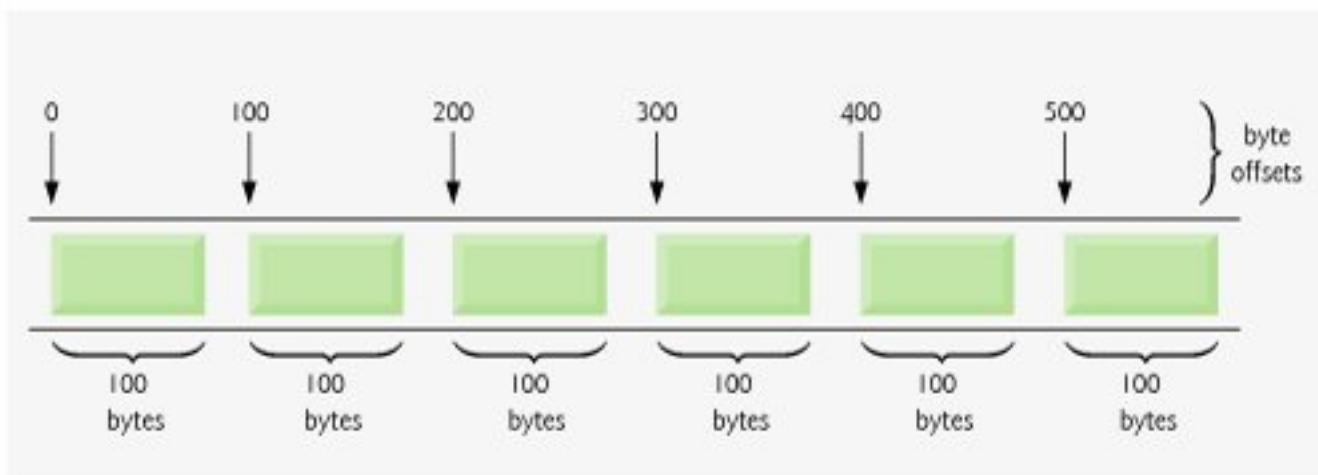
Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 857]

Figure 17.9 illustrates C++'s view of a random-access file composed of fixed-length records (each record, in this case, is 100 bytes long). A random-access file is like a railroad train with many same-size cars some empty and some with contents.

**Figure 17.9. C++ view of a random-access file.**

[\[View full size image\]](#)



Data can be inserted into a random-access file without destroying other data in the file. Data stored previously also can be updated or deleted without rewriting the entire file. In the following sections, we explain how to create a random-access file, enter data into the file, read the data both sequentially and randomly, update the data and delete data that is no longer needed.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 858]

## Converting Between Pointer Types with the `reinterpret_cast` Operator

Unfortunately, most pointers that we pass to function `write` as the first argument are not of type `const char *`. To output objects of other types, we must convert the pointers to those objects to type `const char *`; otherwise, the compiler will not compile calls to function `write`. C++ provides the `reinterpret_cast` operator for cases like this in which a pointer of one type must be cast to an unrelated pointer type. You can also use this cast operator to convert between pointer and integer types, and vice versa. Without a `reinterpret_cast`, the `write` statement that outputs the integer number will not compile because the compiler does not allow a pointer of type `int *` (the type returned by the expression `&number`) to be passed to a function that expects an argument of type `const char *` as far as the compiler is concerned, these types are incompatible.

A `reinterpret_cast` is performed at compile time and does not change the value of the object to which its operand points. Instead, it requests that the compiler reinterpret the operand as the target type (specified in the angle brackets following the keyword `reinterpret_cast`). In Fig. 17.12, we use `reinterpret_cast` to convert a `ClientData` pointer to a `const char *`, which reinterprets a `ClientData` object as bytes to be output to a file. Random-access file-processing programs rarely write a single field to a file. Normally, they write one object of a class at a time, as we show in the following examples.

### Error-Prevention Tip 17.1



It is easy to use `reinterpret_cast` to perform dangerous manipulations that could lead to serious execution-time errors.

### Portability Tip 17.1



Using `reinterpret_cast` is compiler-dependent and can cause programs to behave differently on different platforms. The `reinterpret_cast` operator should not be used unless absolute necessary.

### Portability Tip 17.2



A program that reads unformatted data (written by `write`) must be compiled and executed on a system compatible with the program that wrote the data, because different systems may represent internal data differently.

## Credit Processing Program

Consider the following problem statement:

Create a credit-processing program capable of storing at most 100 fixed-length records for a company that can have up to 100 customers. Each record should consist of an account number that acts as the record key, a last name, a first name and a balance. The program should be able to update an account, insert a new account, delete an account and insert all the account records into a formatted text file for printing.

The next several sections introduce the techniques for creating this credit-processing program. Figure 17.12 illustrates opening a random-access file, defining the record format using an object of class `ClientData` (Figs. 17.1017.11) and writing data to the disk in binary format. This program initializes all 100 records of the file `credit.dat` with empty objects, using function `write`. Each empty object contains 0 for the account number, the null string (represented by empty quotation marks) for the last and first name and 0 . 0 for the balance. Each record is initialized with the amount of empty space in which the account data will be stored.

---

[Page 859]

**Figure 17.10. ClientData class header file.**

```

1 // Fig. 17.10: ClientData.h
2 // Class ClientData definition used in Fig. 17.12Fig. 17.15.
3 #ifndef CLIENTDATA_H
4 #define CLIENTDATA_H
5
6 #include <string>
7 using std::string;
8
9 class ClientData
10 {
11 public:
12 // default ClientData constructor
13 ClientData(int = 0, string = "", string = "", double = 0.0);
14
15 // accessor functions for accountNumber
16 void setAccountNumber(int);

```

```

17 int getAccountNumber() const;
18
19 // accessor functions for lastName
20 void setLastName(string);
21 string getLastName() const;
22
23 // accessor functions for firstName
24 void setFirstName(string);
25 string getFirstName() const;
26
27 // accessor functions for balance
28 void setBalance(double);
29 double getBalance() const;
30 private:
31 int accountNumber;
32 char lastName[15];
33 char firstName[10];
34 double balance;
35 } // end class ClientData
36
37 #endif

```

**Figure 17.11. ClientData class represents a customer's credit information.**

(This item is displayed on pages 860 - 861 in the print version)

```

1 // Fig. 17.11: ClientData.cpp
2 // Class ClientData stores customer's credit information.
3 #include <string>
4 using std::string;
5
6 #include "ClientData.h"
7
8 // default ClientData constructor
9 ClientData::ClientData(int accountNumberValue,
10 string lastNameValue, string firstNameValue, double balanceValue)
11 {
12 setAccountNumber(accountNumberValue);
13 setLastName(lastNameValue);
14 setFirstName(firstNameValue);
15 setBalance(balanceValue);
16 } // end ClientData constructor
17
18 // get account-number value

```

```
19 int ClientData::getAccountNumber() const
20 {
21 return accountNumber;
22 } // end function getAccountNumber
23
24 // set account-number value
25 void ClientData::setAccountNumber(int accountNumberValue)
26 {
27 accountNumber = accountNumberValue; // should validate
28 } // end function setAccountNumber
29
30 // get last-name value
31 string ClientData::getLastName() const
32 {
33 return lastName;
34 } // end function getLastname
35
36 // set last-name value
37 void ClientData::setLastName(string lastNameString)
38 {
39 // copy at most 15 characters from string to lastName
40 const char *lastNameValue = lastNameString.data();
41 int length = lastNameString.size();
42 length = (length < 15 ? length : 14);
43 strncpy(lastName, lastNameValue, length);
44 lastName[length] = '\0'; // append null character to lastName
45 } // end function setLastName
46
47 // get first-name value
48 string ClientData::getFirstName() const
49 {
50 return firstName;
51 } // end function getFirstName
52
53 // set first-name value
54 void ClientData::setFirstName(string firstNameString)
55 {
56 // copy at most 10 characters from string to firstName
57 const char *firstNameValue = firstNameString.data();
58 int length = firstNameString.size();
59 length = (length < 10 ? length : 9);
60 strncpy(firstName, firstNameValue, length);
61 firstName[length] = '\0'; // append null character to firstName
62 } // end function setFirstName
63
64 // get balance value
65 double ClientData::getBalance() const
66 {
```

```

67 return balance;
68 } // end function getBalance
69
70 // set balance value
71 void ClientData::setBalance(double balanceValue)
72 {
73 balance = balanceValue;
74 } // end function setBalance

```

**Figure 17.12. Creating a random-access file with 100 blank records sequentially.**

(This item is displayed on pages 861 - 862 in the print version)

```

1 // Fig. 17.12: Fig17_12.cpp
2 // Creating a randomly accessed file.
3 #include <iostream>
4 using std::cerr;
5 using std::endl;
6 using std::ios;
7
8 #include <fstream>
9 using std::ofstream;
10
11 #include <cstdlib>
12 using std::exit; // exit function prototype
13
14 #include "ClientData.h" // ClientData class definition
15
16 int main()
17 {
18 ofstream outCredit("credit.dat", ios::binary);
19
20 // exit program if ofstream could not open file
21 if (!outCredit)
22 {
23 cerr << "File could not be opened." << endl;
24 exit(1);
25 } // end if
26
27 ClientData blankClient; // constructor zeros out each data member
28
29 // output 100 blank records to file
30 for (int i = 0; i < 100; i++)
31 outCredit.write(reinterpret_cast< const char * >(&blankClient),

```

```

32 sizeof(ClientData));
33
34 return 0;
35 } // end main

```

Objects of class `string` do not have uniform size because they use dynamically allocated memory to accommodate strings of various lengths. This program must maintain fixed-length records, so class `ClientData` stores the client's first and last name in fixed-length `char` arrays. Member functions `setLastName` (Fig. 17.11, lines 3745) and `setFirstName` (Fig. 17.11, lines 5462) each copy the characters of a `string` object into the corresponding `char` array. Consider function `setLastName`. Line 40 initializes the `const char * lastNameValue` with the result of a call to `string` member function `data`, which returns an array containing the characters of the `string`. [Note: This array is not guaranteed to be null terminated.] Line 41 invokes `string` member function `size` to get the length of `lastNameString`. Line 42 ensures that `length` is fewer than 15 characters, then line 43 copies `length` characters from `lastNameValue` into the `char` array `lastName`. Member function `setFirstName` performs the same steps for the first name.

---

[Page 861]

In Fig. 17.12, line 18 creates an `ofstream` object for the file `credit.dat`. The second argument to the constructor `ios::binary` indicates that we are opening the file for output in binary mode, which is required if we are to write fixed-length records. Lines 3132 cause the `blankClient` to be written to the `credit.dat` file associated with `ofstream` object `outCredit`. Remember that operator `sizeof` returns the size in bytes of the object contained in parentheses (see Chapter 8). The first argument to function `write` on line 31 must be of type `const char *`. However, the data type of `&blankClient` is `ClientData *`. To convert `&blankClient` to `const char *`, line 31 uses the cast operator `reinterpret_cast`, so the call to `write` compiles without issuing a compilation error.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 864]

Lines 5960 position the "put" file-position pointer for object `outCredit` to the byte location calculated by

```
(client.getAccountNumber() - 1) * sizeof(ClientData)
```

Because the account number is between 1 and 100, 1 is subtracted from the account number when calculating the byte location of the record. Thus, for record 1, the file-position pointer is set to byte 0 of the file. Note that line 28 uses the `fstream` object `outCredit` to open the existing `credit.dat` file. The file is opened for input and output in binary mode by combining the file-open modes `ios::in`, `ios::out` and `ios::binary`. Multiple fileopen modes are combined by separating each open mode from the next with the bitwise inclusive OR operator (`|`). Opening the existing `credit.dat` file in this manner ensures that this program can manipulate the records written to the file by the program of [Fig. 17.12](#), rather than creating the file from scratch. [Chapter 22](#), Bits, Characters, Strings and `structs`, discusses the bitwise inclusive OR operator in detail.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 865]

### Figure 17.14. Reading a random-access file sequentially.

(This item is displayed on pages 865 - 866 in the print version)

```
1 // Fig. 17.14: Fig17_14.cpp
2 // Reading a random access file sequentially.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8 using std::ios;
9 using std::left;
10 using std::right;
11 using std::showpoint;
12
13 #include <iomanip>
14 using std::setprecision;
15 using std::setw;
16
17 #include <fstream>
18 using std::ifstream;
19 using std::ostream;
20
21 #include <cstdlib>
22 using std::exit; // exit function prototype
23
24 #include "ClientData.h" // ClientData class definition
25
26 void outputLine(ostream&, const ClientData &); // prototype
27
28 int main()
29 {
30 ifstream inCredit("credit.dat", ios::in);
31
32 // exit program if ifstream cannot open file
33 if (!inCredit)
34 {
35 cerr << "File could not be opened." << endl;
36 exit(1);
37 }
```

```

37 } // end if
38
39 cout << left << setw(10) << "Account" << setw(16)
40 << "Last Name" << setw(11) << "First Name" << left
41 << setw(10) << right << "Balance" << endl;
42
43 ClientData client; // create record
44
45 // read first record from file
46 inCredit.read(reinterpret_cast< char * >(&client) ,
47 sizeof(ClientData));
48
49 // read all records from file
50 while (inCredit && !inCredit.eof())
51 {
52 // display record
53 if (client.getAccountNumber() != 0)
54 outputLine(cout, client);
55
56 // read next from file
57 inCredit.read(reinterpret_cast< char * >(&client) ,
58 sizeof(ClientData));
59 } // end while
60
61 return 0;
62 } // end main
63
64 // display single record
65 void outputLine(ostream &output, const ClientData &record)
66 {
67 output << left << setw(10) << record.getAccountNumber()
68 << setw(16) << record.getLastName()
69 << setw(11) << record.getFirstName()
70 << setw(10) << setprecision(2) << right << fixed
71 << showpoint << record.getBalance() << endl;
72 } // end function outputLine

```

| Account | Last Name | First Name | Balance |
|---------|-----------|------------|---------|
| 29      | Brown     | Nancy      | -24.54  |
| 33      | Dunn      | Stacey     | 314.33  |
| 37      | Barker    | Doug       | 0.00    |
| 88      | Smith     | Dave       | 258.34  |
| 96      | Stone     | Sam        | 34.98   |

Figure 17.14 reads every record in the `credit.dat` file sequentially, checks each record to determine whether it contains data, and displays formatted outputs for records containing data. The condition in line 50 uses the `ios` member function `eof` to determine when the end of file is reached and causes execution of the while statement to terminate. Also, if an error occurs when reading from the file, the loop terminates, because `inCredit` evaluates to `false`. The data input from the file is output by function `outputLine` (lines 6572), which takes two arguments an `ostream` object and a `clientData` structure to be output. The `ostream` parameter type is interesting, because any `ostream` object (such as `cout`) or any object of a derived class of `ostream` (such as an object of type `ofstream`) can be supplied as the argument. This means that the same function can be used, for example, to perform output to the standard-output stream and to a file stream without writing separate functions.

---

[Page 866]

What about that additional benefit we promised? If you examine the output window, you will notice that the records are listed in sorted order (by account number). This is a consequence of how we stored these records in the file, using direct-access techniques. Compared to the insertion sort we used in Chapter 7, sorting using direct-access techniques is relatively fast. The speed is achieved by making the file large enough to hold every possible record that might be created. This, of course, means that the file could be occupied sparsely most of the time, resulting in a waste of storage. This is another example of the space-time trade-off: By using large amounts of space, we are able to develop a much faster sorting algorithm. Fortunately, the continuous reduction in price of storage units has made this less of an issue.

---

[Page 867]

 NEXT

 PREV  
page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 872]

The program has five options (option 5 is for terminating the program). Option 1 calls function `createTextFile` to store a formatted list of all the account information in a text file called `print.txt` that may be printed. Function `createTextFile` (lines 100135) takes an `fstream` object as an argument to be used to input data from the `credit.dat` file. Function `createTextFile` invokes `istream` member function `read` (lines 132133) and uses the sequential-file-access techniques of Fig. 17.14 to input data from `credit.dat`. Function `outputLine`, discussed in Section 17.10, is used to output the data to file `print.txt`. Note that `createTextFile` uses `istream` member function `seekg` (line 117) to ensure that the file-position pointer is at the beginning of the file. After choosing Option 1, the `print.txt` file contains

[Page 873]

| Account | Last Name | First Name | Balance |
|---------|-----------|------------|---------|
| 29      | Brown     | Nancy      | -24.54  |
| 33      | Dunn      | Stacey     | 314.33  |
| 37      | Barker    | Doug       | 0.00    |
| 88      | Smith     | Dave       | 258.34  |
| 96      | Stone     | Sam        | 34.98   |

Option 2 calls `updateRecord` (lines 138176) to update an account. This function updates only an existing record, so the function first determines whether the specified record is empty. Lines 148149 read data into object `client`, using `istream` member function `read`. Then line 152 compares the value returned by `getAccountNumber` of the `client` structure to zero to determine whether the record contains information. If this value is zero, lines 174175 print an error message indicating that the record is empty. If the record contains information, line 154 displays the record, using function `outputLine`, line 159 inputs the transaction amount and lines 162171 calculate the new balance and rewrite the record to the file. A typical output for Option 2 is

```
Enter account to update (1 - 100): 37
37 Barker Doug 0.00

Enter charge (+) or payment (-): +87.99
37 Barker Doug 87.99
```

Option 3 calls function `newRecord` (lines 179221) to add a new account to the file. If the user enters an account number for an existing account, `newRecord` displays an error message indicating that the account exists (lines 219220). This function adds a new account in the same manner as the program of Fig. 17.12. A typical output for Option 3 is

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

Option 4 calls function `deleteRecord` (lines 224255) to delete a record from the file. Line 227 prompts the user to enter the account number. Only an existing record may be deleted, so, if the specified account is empty, line 254 displays an error message. If the account exists, lines 247249 reinitialize that account by copying an empty record (`blankClient`) to the file. Line 251 displays a message to inform the user that the record has been deleted. A typical output for Option 4 is

```
Enter account to delete (1 - 100): 29
Account #29 deleted.
```

Note that line 43 opens the `credit.dat` file by creating an `fstream` object for both reading and writing, using modes `ios::in` and `ios::out` "or-ed" together.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 874]

## 17.12. Input/Output of Objects

This chapter and [Chapter 15](#) introduced C++'s object-oriented style of input/output. However, our examples concentrated on I/O of traditional data types rather than objects of user-defined types. In [Chapter 11](#), we showed how to input and output objects using operator overloading. We accomplished object input by overloading the stream extraction operator `>>` for the appropriate `istream`. We accomplished object output by overloading the stream insertion operator `<<` for the appropriate `ostream`. In both cases, only an object's data members were input or output, and, in each case, they were in a format meaningful only for objects of that particular abstract data type. An object's member functions are not input and output with the object's data; rather, one copy of the class's member functions remains available internally and is shared by all objects of the class.

When object data members are output to a disk file, we lose the object's type information. We store only data bytes, not type information, on the disk. If the program that reads this data knows the object type to which the data corresponds, the program will read the data into objects of that type.

An interesting problem occurs when we store objects of different types in the same file. How can we distinguish them (or their collections of data members) as we read them into a program? The problem is that objects typically do not have type fields (we studied this issue carefully in [Chapter 13](#)).

One approach would be to have each overloaded output operator output a type code preceding each collection of data members that represents one object. Then object input would always begin by reading the type-code field and using a `switch` statement to invoke the proper overloaded function. Although this solution lacks the elegance of polymorphic programming, it provides a workable mechanism for retaining objects in files and retrieving them as needed.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 874 (continued)]

## 17.13. Wrap-Up

In this chapter, we presented various file-processing techniques to manipulate persistent data. You learned that data is stored in computers in the form of 0s and 1s, and that combinations of these values form bytes, fields, records and eventually files. You were introduced to the differences between character-based and byte-based streams, and to several file-processing class templates in header file `<fstream>`. Then, you learned how to use sequential file processing to manipulate records stored in order, by the record-key field. You also learned how to use random-access files to instantly retrieve and manipulate fixed-length records. Finally, we presented a substantial transaction-processing case study using a random-access file to achieve "instant" access processing. In the next chapter, we discuss typical string-manipulation operations provided by class template `basic_string`. We also introduce string stream-processing capabilities that allow strings to be input from and output to memory.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 875]

- The set of all characters used to write programs and represent data items on a particular computer is called that computer's character set.
- Bytes are composed of eight bits.
- Just as characters are composed of bits, fields are composed of characters. A field is a group of characters that conveys some meaning.
- Typically, a record (i.e., a class in C++) is composed of several fields (i.e., data members in C++).
- At least one field in a record is chosen as a record key to identify a record as belonging to a particular person or entity that is distinct from all other records in the file.
- In a sequential file, records typically are stored in order by a record-key field.
- A group of related files often are stored in a database.
- A collection of programs designed to create and manage databases is called a database management system (DBMS).
- C++ views each file as a sequential stream of bytes.
- Each file ends either with an end-of-file marker or at a specific byte number recorded in a system-maintained, administrative data structure.
- Header <fstream> includes the definitions for the stream class templates `basic_ifstream` (for file input), `basic_ofstream` (for file output) and `basic_fstream` (for file input and output).
- For an `ofstream` object, the file-open mode can be either `ios::out` to output data to a file or `ios::app` to append data to the end of a file (without modifying any data already in the file).
- The file-open mode `ios::ate` opens a file for output and moves to the end of the file. This is normally used to append data to a file, but data can be written anywhere in the file.
- Existing files opened with mode `ios::out` are truncated (i.e., all data in the file is discarded).
- By default, `ofstream` objects are opened for output.
- The `ofstream` member function `open` opens a file and attaches it to an existing `ofstream` object.
- An overloaded `ios` operator member function `operator void*` converts the stream to a pointer, so it can be tested as 0 (i.e., the null pointer) or nonzero (i.e., any other pointer value).
- You can use the `ofstream` member function `close` to close the `ofstream` object explicitly.
- Both `istream` and `ostream` provide member functions for repositioning the file-position pointer (the byte number of the next byte in the file to be read or written). These member functions are `seekg` ("seek get") for `istream` and `seekp` ("seek put") for `ostream`.
- The seek direction can be `ios::beg` (the default) for positioning relative to the beginning of a stream, `ios::cur` for positioning relative to the current position in a stream or `ios::end` for positioning relative to the end of a stream.
- Member functions `tellg` and `tellp` are provided to return the current locations of the "get" and "put" pointers, respectively.
- Individual records of a random-access file can be accessed directly (and quickly) without the need to search other records.
- The `ostream` member function `write` outputs a fixed number of bytes, beginning at a specific location in memory, to the specified stream. When the stream is associated with a file, function `write` writes the data at the location in the file specified by the "put" file-position pointer.
- The `istream` member function `read` inputs a fixed number of bytes from the specified stream to an area in memory beginning at a specified address. If the stream is associated with a file, function `read`

inputs bytes at the location in the file specified by the "get" file-position pointer.

- `string` member function `data` converts a string to a non-null-terminated C-style character array.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 876]

## Terminology

binary digit

bit

byte

character field

character set

`cin` (standard input)

`clog` (standard error buffered)

`close` member function of `ofstream`

database

database management system (DBMS)

data function of `string`

data hierarchy

data persistence

decimal digit

end-of-file

field

file

file name

file-position pointer

fstream

<fstream> header file

instant-access application

ios::app file open mode

ios::ate file open mode

ios::beg seek starting point

ios::binary file open mode

ios::cur seek direction

ios::end seek direction

ios::in file open mode

ios::out file open mode

ios::trunc file open mode

offset from the beginning of a file

open a file

open member function of ofstream

random-access file

record

record key

secondary storage device

seek direction

`seekg istream` member function

`seekp ostream` member function

sequential file

`size` function of `string`

special symbol

`tellg istream` member function

`tellp ostream` member function

transaction-processing system

truncate an existing file

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 877]

- If the file-position pointer points to a location in a sequential file other than the beginning of the file, the file must be closed and reopened to read from the beginning of the file.
- The `ostream` member function `write` can write to standard-output stream `cout`.
- Data in sequential files always is updated without overwriting nearby data.
- Searching all records in a random-access file to find a specific record is unnecessary.
- Records in random-access files must be of uniform length.
- Member functions `seekp` and `seekg` must seek relative to the beginning of a file.

### 17.3

Assume that each of the following statements applies to the same program.

a.

Write a statement that opens file `oldmast.dat` for input; use an `ifstream` object called `inOldMaster`.

b.

Write a statement that opens file `trans.dat` for input; use an `ifstream` object called `inTransaction`.

c.

Write a statement that opens file newmast.dat for output (and creation); use ofstream object outNewMaster.

d.

Write a statement that reads a record from the file oldmast.dat. The record consists of integer accountNumber, string name and floating-point currentBalance; use ifstream object inOldMaster.

e.

Write a statement that reads a record from the file TRans.dat. The record consists of integer accountNum and floating-point dollarAmount; use ifstream object inTransaction.

f.

Write a statement that writes a record to the file newmast.dat. The record consists of integer accountNum, string name, and floating-point currentBalance; use ofstream object outNewMaster.

## 17.4

Find the error(s) and show how to correct it (them) in each of the following.

a.

File payables.dat referred to by ofstream object outPayable has not been opened.

```
outPayable << account << company << amount << endl;
```

b.

The following statement should read a record from the file payables.dat. The ifstream object inPayable refers to this file, and istream object inReceivable refers to the file receivables.dat.

```
inReceivable >> account >> company >> amount;
```

c.

The file `tools.dat` should be opened to add data to the file without discarding the current data.

```
ofstream outTools("tools.dat", ios::out);
```

 PREV

NEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 878]

•

True.

•

False. In most cases, sequential file records are not of uniform length. Therefore, it is possible that updating a record will cause other data to be overwritten.

•

True.

•

False. Records in a random-access file normally are of uniform length.

•

False. It is possible to seek from the beginning of the file, from the end of the file and from the current position in the file.

## 17.3

a.

```
ifstream inOldMaster("oldmast.dat", ios::in);
```

b.

```
ifstream inTransaction("trans.dat", ios::in);
```

c.

```
ofstream outNewMaster("newmast.dat", ios::out);
```

d.

```
inOldMaster >> accountNumber >> name >> currentBalance;
```

e.

```
inTransaction >> accountNum >> dollarAmount;
```

f.

```
outNewMaster << accountNum << name << currentBalance;
```

## 17.4

a.

Error: The file `payables.dat` has not been opened before the attempt is made to output data to the stream.

Correction: Use `ostream` function `open` to open `payables.dat` for output.

b.

Error: The incorrect `istream` object is being used to read a record from the file named `payables.dat`.

Correction: Use `istream` object `inPayable` to refer to `payables.dat`.

c.

Error: The file's contents are discarded because the file is opened for output (`ios::out`).

Correction: To add data to the file, open the file either for updating (`ios::ate`) or for appending (`ios::app`).

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 879]

•

Data items represented in computers form a data hierarchy in which data items become larger and more complex as we progress from fields to characters to bits, etc.

•

A record key identifies a record as belonging to a particular field.

•

Most organizations store all information in a single file to facilitate computer processing.

•

When a program creates a file, the file is automatically retained by the computer for future reference; i.e., files are said to be persistent.

## 17.7

Exercise 17.3 asked the reader to write a series of single statements. Actually, these statements form the core of an important type of file-processing program, namely, a file-matching program. In commercial data processing, it is common to have several files in each application system. In an accounts-receivable system, for example, there is generally a master file containing detailed information about each customer, such as the customer's name, address, telephone number, outstanding balance, credit limit, discount terms, contract arrangements and, possibly, a condensed history of recent purchases and cash payments.

As transactions occur (e.g., sales are made and cash payments arrive), they are entered into a file. At the end of each business period (a month for some companies, a week for others and a day in some cases), the file of transactions (called `trans.dat` in Exercise 17.3) is applied to the master file (called `oldmast.dat` in Exercise 17.3), thus updating each account's record of purchases and payments. During an updating run, the master file is rewritten as a new file (`newmast.dat`), which is then used at the end of the next business period to begin the updating process again.

File-matching programs must deal with certain problems that do not exist in single-file programs. For example, a match does not always occur. A customer on the master file might not have made any purchases or cash payments in the current business period, and therefore no record for this customer will appear on the transaction file. Similarly, a customer who did make some purchases or cash payments may have just moved to this community, and the company may not have had a chance to create a master

record for this customer.

Use the statements from [Exercise 17.3](#) as a basis for writing a complete file-matching accounts-receivable program. Use the account number on each file as the record key for matching purposes. Assume that each file is a sequential file with records stored in increasing order by account number.

When a match occurs (i.e., records with the same account number appear on both the master and transaction files), add the dollar amount on the transaction file to the current balance on the master file, and write the newmast.dat record. (Assume purchases are indicated by positive amounts on the transaction file and payments are indicated by negative amounts.) When there is a master record for a particular account but no corresponding transaction record, merely write the master record to newmast.dat. When there is a transaction record but no corresponding master record, print the error message "Unmatched transaction record for account number . . ." (fill in the account number from the transaction record).

## 17.8

After writing the program of [Exercise 17.7](#), write a simple program to create some test data for checking out the program. Use the following sample account data:

| <b>Master file Account number</b> | <b>Name</b> | <b>Balance</b> |
|-----------------------------------|-------------|----------------|
| 100                               | Alan Jones  | 348.17         |
| 300                               | Mary Smith  | 27.19          |
| 500                               | Sam Sharp   | 0.00           |
| 700                               | Suzy Green  | 14.22          |

---

[Page 880]

| <b>Transaction file Account number</b> | <b>Transaction amount</b> |
|----------------------------------------|---------------------------|
| 100                                    | 27.14                     |
| 300                                    | 62.11                     |
| 400                                    | 100.56                    |
| 900                                    | 82.17                     |

**17.9**

Run the program of [Exercise 17.7](#), using the files of test data created in [Exercise 17.8](#). Print the new master file. Check that the accounts have been updated correctly.

**17.10**

It is possible (actually common) to have several transaction records with the same record key. This occurs because a particular customer might make several purchases and cash payments during a business period. Rewrite your accounts-receivable file-matching program of [Exercise 17.7](#) to provide for the possibility of handling several transaction records with the same record key. Modify the test data of [Exercise 17.8](#) to include the following additional transaction records:

| <b>Account number</b> | <b>Dollar amount</b> |
|-----------------------|----------------------|
| 300                   | 83.89                |
| 700                   | 80.78                |
| 700                   | 1.53                 |

**17.11**

Write a series of statements that accomplish each of the following. Assume that we have defined class Person that contains private data members

```
char lastName[15];
char firstName[15];
char age[4];
```

and public member functions

```
// accessor functions for lastName
void setLastName(string);
string getLastname() const;

// accessor functions for firstName
void setFirstName(string);
string getFirstName() const;

// accessor functions for age
```

```
void setAge(string);
string getAge() const;
```

Also assume that any random-access files have been opened properly.

**a.**

Initialize the file `nameage.dat` with 100 records that store values `lastName = "unassigned"`, `firstName = ""` and `age = "0"`.

**b.**

Input 10 last names, first names and ages, and write them to the file.

[Page 881]

**c.**

Update a record that already contains information. If the record does not contain information, inform the user "No info".

**d.**

Delete a record that contains information by reinitializing that particular record.

## 17.12

You are the owner of a hardware store and need to keep an inventory that can tell you what different tools you have, how many of each you have on hand and the cost of each one. Write a program that initializes the random-access file `hardware.dat` to 100 empty records, lets you input the data concerning each tool, enables you to list all your tools, lets you delete a record for a tool that you no longer have and lets you update any information in the file. The tool identification number should be the record number. Use the following information to start your file:

| Record# | Tool name       | Quantity | Cost  |
|---------|-----------------|----------|-------|
| 3       | Electric sander | 7        | 57.98 |
| 17      | Hammer          | 76       | 11.99 |
| 24      | Jig saw         | 21       | 11.00 |

|    |               |     |       |
|----|---------------|-----|-------|
| 39 | Lawn mower    | 3   | 79.50 |
| 56 | Power saw     | 18  | 99.99 |
| 68 | Screwdriver   | 106 | 6.99  |
| 77 | Sledge hammer | 11  | 21.50 |
| 83 | Wrench        | 34  | 7.50  |

**17.13**

(Telephone Number Word Generator) Standard telephone keypads contain the digits 0 through 9. The numbers 2 through 9 each have three letters associated with them, as is indicated by the following table:

| Digit | Letter |
|-------|--------|
| 2     | A B C  |
| 3     | D E F  |
| 4     | G H I  |
| 5     | J K L  |
| 6     | M N O  |
| 7     | P R S  |
| 8     | T U V  |
| 9     | W X Y  |

Many people find it difficult to memorize phone numbers, so they use the correspondence between digits and letters to develop seven-letter words that correspond to their phone numbers. For example, a person whose telephone number is 686-2377 might use the correspondence indicated in the above table to develop the seven-letter word "NUMBERS."

Businesses frequently attempt to get telephone numbers that are easy for their clients to remember. If a business can advertise a simple word for its customers to dial, then no doubt the business will receive a few more calls.

Each seven-letter word corresponds to exactly one seven-digit telephone number. The restaurant wishing to increase its take-home business could surely do so with the number 825-3688 (i.e., "TAKEOUT").

Each seven-digit phone number corresponds to many separate seven-letter words. Unfortunately, most of these represent unrecognizable juxtapositions of letters. It is possible, however, that the owner of a barber shop would be pleased to know that the shop's telephone number, 424-7288, corresponds to "HAIRCUT." The owner of a liquor store would, no doubt, be delighted to find that the store's telephone number, 233-7226, corresponds to "BEERCAN." A veterinarian with the phone number 738-2273 would be pleased to know that the number corresponds to the letters "PETCARE."

Write a C++ program that, given a seven-digit number, writes to a file every possible seven-letter word corresponding to that number. There are 2187 (3 to the seventh power) such words. Avoid phone numbers with the digits 0 and 1.

## 17.14

Write a program that uses the `sizeof` operator to determine the sizes in bytes of the various data types on your computer system. Write the results to the file `datasize.dat`, so that you may print the results later. The results should be displayed in two-column format with the type name in the left column and the size of the type in right column, as in:

|                                 |    |
|---------------------------------|----|
| <code>char</code>               | 1  |
| <code>unsigned char</code>      | 1  |
| <code>short int</code>          | 2  |
| <code>unsigned short int</code> | 2  |
| <code>int</code>                | 4  |
| <code>unsigned int</code>       | 4  |
| <code>long int</code>           | 4  |
| <code>unsigned long int</code>  | 4  |
| <code>float</code>              | 4  |
| <code>double</code>             | 8  |
| <code>long double</code>        | 10 |

[Note: The sizes of the built-in data types on your computer might differ from those listed above.]

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 884]

## Outline

[18.1 Introduction](#)

[18.2 string Assignment and Concatenation](#)

[18.3 Comparing strings](#)

[18.4 Substrings](#)

[18.5 Swapping strings](#)

[18.6 string Characteristics](#)

[18.7 Finding Strings and Characters in a string](#)

[18.8 Replacing Characters in a string](#)

[18.9 Inserting Characters into a string](#)

[18.10 Conversion to C-Style char \\* Strings](#)

[18.11 Iterators](#)

[18.12 String Stream Processing](#)

[18.13 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

## Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 885]

Remember that operator = in the preceding declaration is not an assignment; rather it is an implicit call to the `string` class constructor, which does the conversion.

Note that class `string` provides no conversions from `int` or `char` to `string` in a `string` definition. For example, the definitions

```
string error1 = 'c';
string error2('u');
string error3 = 22;
string error4(8);
```

result in syntax errors. Note that assigning a single character to a `string` object is permitted in an assignment statement as in

```
string1 = 'n';
```

### Common Programming Error 18.1



Attempting to convert an `int` or `char` to a `string` via an initialization in a declaration or via a constructor argument is a compilation error.

Unlike C-style `char *` strings, `strings` are not necessarily null terminated. [Note: The C++ standard document provides only a description of the interface for class `string`; implementation is platform dependent.] The length of a `string` can be retrieved with member function `length` and with member function `size`. The subscript operator, `[ ]`, can be used with `strings` to access and modify individual characters. Like C-style strings, `strings` have a first subscript of 0 and a last subscript of `length() - 1`.

Most `string` member functions take as arguments a starting subscript location and the number of characters on which to operate.

The stream extraction operator (`>>`) is overloaded to support `strings`. The statement

```
string stringObject;
```

```
cin >> stringObject;
```

reads a `string` from the standard input device. Input is delimited by white-space characters. When a delimiter is encountered, the input operation is terminated. Function `getline` also is overloaded for `strings`. The statement

```
string string1;
getline(cin, string1);
```

reads a `string` from the keyboard into `string1`. Input is delimited by a newline ('`\n`'), so `getLine` can read a line of text into a `string` object.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 886]

where sourceString is the string to be copied, start is the starting subscript and numberOfCharacters is the number of characters to copy.

### Figure 18.1. Demonstrating string assignment and concatenation.

(This item is displayed on pages 886 - 887 in the print version)

```
1 // Fig. 18.1: Fig18_01.cpp
2 // Demonstrating string assignment and concatenation.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string string1("cat");
13 string string2;
14 string string3;
15
16 string2 = string1; // assign string1 to string2
17 string3.assign(string1); // assign string1 to string3
18 cout << "string1: " << string1 << "\nstring2: " << string2
19 << "\nstring3: " << string3 << "\n\n";
20
21 // modify string2 and string3
22 string2[0] = string3[2] = 'r';
23
24 cout << "After modification of string2 and string3:\n" << "string1: "
25 << string1 << "\nstring2: " << string2 << "\nstring3: ";
26
27 // demonstrating member function at
28 for (int i = 0; i < string3.length(); i++)
29 cout << string3.at(i);
30
31 // declare string4 and string5
32 string string4(string1 + "apult"); // concatenation
33 string string5;
```

```

34
35 // overloaded +=
36 string3 += "pet"; // create "carpet"
37 string1.append("acomb"); // create "catacomb"
38
39 // append subscript locations 4 through end of string1 to
40 // create string "comb" (string5 was initially empty)
41 string5.append(string1, 4, string1.length() - 4);
42
43 cout << "\n\nAfter concatenation:\nstring1: " << string1
44 << "\nstring2: " << string2 << "\nstring3: " << string3
45 << "\nstring4: " << string4 << "\nstring5: " << string5 << endl;
46 return 0;
47 } // end main

```

```

string1: cat
string2: cat
string3: cat

After modification of string2 and string3:
string1: cat
string2: rat
string3: car

After concatenation:
string1: catacomb
string2: rat
string3: carpet
string4: catapult
string5: comb

```

Line 22 uses the subscript operator to assign 'r' to `string3[ 2 ]` (forming "car") and to assign 'r' to `string2[ 0 ]` (forming "rat"). The strings are then output.

---

[Page 887]

Lines 2829 output the contents of `string3` one character at a time using member function `at`. Member function `at` provides **checked access** (or **range checking**); i.e., going past the end of the `string` throws an `out_of_range` exception. (See [Chapter 16](#) for a detailed discussion of exception handling.)

Note that the subscript operator, [ ], does not provide checked access. This is consistent with its use on arrays.

### Common Programming Error 18.2



Accessing a `string` subscript outside the bounds of the `string` using function `at` is a logic error that causes an `out_of_range` exception.

### Common Programming Error 18.3



Accessing an element beyond the size of the `string` using the subscript operator is an unreported logic error.

String `string4` is declared (line 32) and initialized to the result of concatenating `string1` and "apult" using the overloaded addition operator, +, which for class `string` denotes concatenation. Line 36 uses the addition assignment operator, +=, to concatenate `string3` and "pet". Line 37 uses member function **append** to concatenate `string1` and "acomb".

Line 41 appends the string "comb" to empty string `string5`. This member function is passed the string (`string1`) to retrieve characters from, the starting subscript in the string (4) and the number of characters to append (the value returned by `string1.length() - 4`).



**page footer**



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

---

[Page 888]

[Page 889]

Line 32 uses `string` member function `compare` to compare `string1` to `string2`. Variable `result` is assigned 0 if the strings are equivalent, a positive number if `string1` is **lexicographically** greater than `string2` or a negative number if `string1` is lexicographically less than `string2`. Because a string starting with 'T' is considered lexicographically greater than a string starting with 'H', `result` is assigned a value greater than 0, as confirmed by the output. A lexicon is a dictionary. When we say that a string is lexicographically less than another, we mean that the first string is alphabetically less than the second. The computer uses the same criterion as you would use in alphabetizing a list of names.

---

[Page 890]

Line 45 uses an overloaded version of member function `compare` to compare portions of `string1` and `string3`. The first two arguments (2 and 5) specify the starting subscript and length of the portion of `string1` ("sting") to compare with `string3`. The third argument is the comparison `string`. The last two arguments (0 and 5) are the starting subscript and length of the portion of the comparison `string` being compared (also "sting"). The value assigned to `result` is 0 for equality, a positive number if `string1` is lexicographically greater than `string3` or a negative number if `string1` is lexicographically less than `string3`. Because the two pieces of `strings` being compared here are identical, `result` is assigned 0.

Line 58 uses another overloaded version of function `compare` to compare `string4` and `string2`. The first two arguments are the same the starting subscript and length. The last argument is the comparison `string`. The value returned is also the same 0 for equality, a positive number if `string4` is lexicographically greater than `string2` or a negative number if `string4` is lexicographically less than `string2`. Because the two pieces of `strings` being compared here are identical, `result` is assigned 0.

Line 74 calls member function `compare` to compare the first 3 characters in `string2` to `string4`. Because "He1" is less than "Hello", a value less than zero is returned.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 891]

The program declares and initializes a `string` on line 12. Line 16 uses member function `substr` to retrieve a substring from `string1`. The first argument specifies the beginning subscript of the desired substring; the second argument specifies the substring's length.

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 891 (continued)]

## 18.5. Swapping strings

Class `string` provides member function `swap` for swapping strings. [Figure 18.4](#) swaps two strings.

**Figure 18.4. Using function swap to swap two strings.**

```
1 // Fig. 18.4: Fig18_04.cpp
2 // Using the swap function to swap two strings.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string first("one");
13 string second("two");
14
15 // output strings
16 cout << "Before swap:\n first: " << first << "\nsecond: " << second;
17
18 first.swap(second); // swap strings
19
20 cout << "\n\nAfter swap:\n first: " << first
21 << "\nsecond: " << second << endl;
22 return 0;
23 } // end main
```

```
Before swap:
first: one
second: two

After swap:
first: two
second: one
```

Lines 1213 declare and initialize strings `first` and `second`. Each string is then output. Line 18 uses `string` member function `swap` to swap the values of `first` and `second`. The two strings are printed again to confirm that they were indeed swapped. The `string` member function `swap` is useful for implementing programs that sort strings.

[◀ PREV](#)[NEXT ▶](#)**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 894]

The program declares empty string `string1` (line 16) and passes it to function `printStatistics` (line 19). Function `printStatistics` (lines 4955) takes a reference to a `const string` as an argument and outputs the capacity (using member function `capacity`), maximum size (using member function `max_size`), size (using member function `size`), length (using member function `length`) and whether the `string` is empty (using member function `empty`). The initial call to `printStatistics` indicates that the initial values for the capacity, size and length of `string1` are 0.

The size and length of 0 indicate that there are no characters stored in `string`. Because the initial capacity is 0, when characters are placed in `string1`, memory is allocated to accommodate the new characters. Recall that the size and length are always identical. In this implementation, the maximum size is 4294967293. Object `string1` is an empty `string`, so function `empty` returns `TRUE`.

Line 23 reads a string from the command line. In this example, "tomato soup" is input. Because a space character is a delimiter, only "tomato" is stored in `string1`; however, "soup" remains in the input buffer. Line 27 calls function `printStatistics` to output statistics for `string1`. Notice in the output that the length is 6 and that the capacity is 15.

#### Performance Tip 18.1



To minimize the number of times memory is allocated and deallocated, some `string` class implementations provide a default capacity above and beyond the length of the `string`.

Line 30 reads "soup" from the input buffer and stores it in `string1`, thereby replacing "tomato". Line 32 passes `string1` to `printStatistics`.

Line 35 uses the overloaded `+=` operator to concatenate a 46-character-long string to `string1`. Line 37 passes `string1` to `printStatistics`. Notice that the capacity has increased to 63 elements and the length is now 50.

Line 40 uses member function `resize` to increase the length of `string1` by 10 characters. The additional elements are set to null characters. Notice that in the output the capacity has not changed and the length is now 60.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 896]

String `string1` is declared and initialized in line 12. Line 17 attempts to find "is" in `string1` using function `find`. If "is" is found, the subscript of the starting location of that string is returned. If the string is not found, the value `string::npos` (a public static constant defined in class `string`) is returned. This value is returned by the `string` `find` related functions to indicate that a substring or character was not found in the `string`.

Line 18 uses member function `rfind` to search `string1` backward (i.e., right-to-left). If "is" is found, the subscript location is returned. If the string is not found, `string::npos` is returned. [Note: The rest of the `find` functions presented in this section return the same type unless otherwise noted.]

Line 21 uses member function `find_first_of` to locate the first occurrence in `string1` of any character in "misop". The searching is done from the beginning of `string1`. The character 'o' is found in element 1.

Line 26 uses member function `find_last_of` to find the last occurrence in `string1` of any character in "misop". The searching is done from the end of `string1`. The character 'o' is found in element 29.

Line 31 uses member function `find_first_not_of` to find the first character in `string1` not contained in "noi spm". The character '1' is found in element 8. Searching is done from the beginning of `string1`.

Line 37 uses member function `find_first_not_of` to find the first character not contained in "12noi spm". The character '.' is found in element 12. Searching is done from the end of `string1`.

Lines 4344 use member function `find_first_not_of` to find the first character not contained in "noon is 12 pm; midnight is not.". In this case, the `string` being searched contains every character specified in the `string` argument. Because a character was not found, `string::npos` (which has the value 1 in this case) is returned.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)
[NEXT ▶](#)

[Page 896 (continued)]

## 18.8. Replacing Characters in a string

Figure 18.7 demonstrates `string` member functions for replacing and erasing characters. Lines 1317 declare and initialize `string string1`. Line 23 uses `string` member function `erase` to erase everything from (and including) the character in position 62 to the end of `string1`. [Note: Each newline character occupies one element in the `string`.]

**Figure 18.7. Demonstrating functions `erase` and `replace`.**

(This item is displayed on pages 897 - 898 in the print version)

```

1 // Fig. 18.7: Fig18_07.cpp
2 // Demonstrating string member functions erase and replace.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 // compiler concatenates all parts into one string
13 string string1("The values in any left subtree"
14 "\nare less than the value in the"
15 "\nparent node and the values in"
16 "\nany right subtree are greater"
17 "\nthan the value in the parent node");
18
19 cout << "Original string:\n" << string1 << endl << endl;
20
21 // remove all characters from (and including) location 62
22 // through the end of string1
23 string1.erase(62);
24
25 // output new string
26 cout << "Original string after erase:\n" << string1

```

```

27 << "\n\nAfter first replacement:\n";
28
29 int position = string1.find(" "); // find first space
30
31 // replace all spaces with period
32 while (position != string::npos)
33 {
34 string1.replace(position, 1, ".");
35 position = string1.find(" ", position + 1);
36 } // end while
37
38 cout << string1 << "\n\nAfter second replacement:\n";
39
40 position = string1.find("."); // find first period
41
42 // replace all periods with two semicolons
43 // NOTE: this will overwrite characters
44 while (position != string::npos)
45 {
46 string1.replace(position, 2, "xxxxx;yyy", 5, 2);
47 position = string1.find(".", position + 1);
48 } // end while
49
50 cout << string1 << endl;
51 return 0;
52 } // end main

```

Original string:

The values in any left subtree  
are less than the value in the  
parent node and the values in  
any right subtree are greater  
than the value in the parent node

Original string after erase:

The values in any left subtree  
are less than the value in the

After first replacement:

The.values.in.any.left.subtree  
are.less.than.the.value.in.the

After second replacement:

The;;values;;n;;ny;;eft;;ubtree  
are;;ess;;han;;he;;alue;;n;;he

Lines 2936 use `find` to locate each occurrence of the space character. Each space is then replaced with a period by a call to `string` member function `replace`. Function `replace` takes three arguments: the subscript of the character in the `string` at which replacement should begin, the number of characters to replace and the replacement string. Member function `find` returns `string::npos` when the search character is not found. In line 35, 1 is added to `position` to continue searching at the location of the next character.

Lines 4048 use function `find` to find every period and another overloaded function `replace` to replace every period and its following character with two semicolons. The arguments passed to this version of `replace` are the subscript of the element where the replace operation begins, the number of characters to replace, a replacement character string from which a substring is selected to use as replacement characters, the element in the character string where the replacement substring begins and the number of characters in the replacement character string to use.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 898]

## 18.9. Inserting Characters into a string

Class `string` provides member functions for inserting characters into a `string`. Figure 18.8 demonstrates the `string insert` capabilities.

**Figure 18.8. Demonstrating the `string insert` member functions.**

(This item is displayed on pages 898 - 899 in the print version)

```
1 // Fig. 18.8: Fig18_08.cpp
2 // Demonstrating class string insert member functions.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string string1("beginning end");
13 string string2("middle ");
14 string string3("12345678");
15 string string4("xx");
16
17 cout << "Initial strings:\nstring1: " << string1
18 << "\nstring2: " << string2 << "\nstring3: " << string3
19 << "\nstring4: " << string4 << "\n\n";
20
21 // insert "middle" at location 10 in string1
22 string1.insert(10, string2);
23
24 // insert "xx" at location 3 in string3
25 string3.insert(3, string4, 0, string::npos);
26
27 cout << "Strings after insert:\nstring1: " << string1
28 << "\nstring2: " << string2 << "\nstring3: " << string3
```

```
29 << "\nstring4: " << string4 << endl;
30 return 0;
31 } // end main
```

```
Initial strings:
string1: beginning end
string2: middle
string3: 12345678
string4: xx
```

```
Strings after insert:
string1: beginning middle end
string2: middle
string3: 123xx45678
string4: xx
```

The program declares, initializes and then outputs strings `string1`, `string2`, `string3` and `string4`. Line 22 uses `string` member function `insert` to insert `string2`'s content before element 10 of `string1`.

Line 25 uses `insert` to insert `string4` before `string3`'s element 3. The last two arguments specify the starting and last element of `string4` that should be inserted. Using `string::npos` causes the entire string to be inserted.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 900]

Lines 3233 use pointer arithmetic to output the character array pointed to by `ptr1`. In lines 3536, the C-style string pointed to by `ptr2` is output and the memory allocated for `ptr2` is deleted to avoid a memory leak.

[Page 901]

#### Common Programming Error 18.4



Not terminating the character array returned by `data` with a null character can lead to execution-time errors.

#### Good Programming Practice 18.1



Whenever possible, use the more robust `string` class objects rather than C-style pointer-based strings.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 902]

Lines 1213 declare `string string1` and `string::const_iterator iterator1`. A `const_iterator` is an iterator that cannot modify the `string` in this case the `string` through which it is iterating. Iterator `iterator1` is initialized to the beginning of `string1` with the `string` class member function `begin`. Two versions of `begin` exist one that returns an iterator for iterating through a non-`const` `string` and a `const` version that returns a `const_iterator` for iterating through a `const` `string`. Line 15 outputs `string1`.

Lines 1923 use iterator `iterator1` to "walk through" `string1`. Class `string` member function `end` returns an iterator (or a `const_iterator`) for the position past the last element of `string1`. Each element is printed by dereferencing the iterator much as you would dereference a pointer, and the iterator is advanced one position using operator `++`.

Class `string` provides member functions `rend` and `rbegin` for accessing individual `string` characters in reverse from the end of a `string` toward the beginning. Member functions `rend` and `rbegin` can return `reverse_iterators` and `const_reverse_iterators` (based on whether the `string` is `non-const` or `const`). In the exercises, we ask the reader to write a program that demonstrates these capabilities. We will use iterators and reverse iterators more in [Chapter 23](#).

### Error-Prevention Tip 18.1



Use `string` member function `at` (rather than iterators) when you want the benefit of range checking.

### Good Programming Practice 18.2



When the operations involving the iterator should not modify the data being processed, use a `const_iterator`. This is another example of employing the principle of least privilege.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 903]

An `ostringstream` object uses a `string` object to store the output data. The `str` member function of class `ostringstream` returns a copy of that `string`.

Figure 18.11 demonstrates an `ostringstream` object. The program creates `ostringstream` object `outputString` (line 15) and uses the stream insertion operator to output a series of strings and numerical values to the object.

### Figure 18.11. Using a dynamically allocated `ostringstream` object.

(This item is displayed on pages 903 - 904 in the print version)

```
1 // Fig. 18.11: Fig18_11.cpp
2 // Using a dynamically allocated ostringstream object.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 #include <sstream> // header file for string stream processing
11 using std::ostringstream; // stream insertion operators
12
13 int main()
14 {
15 ostringstream outputString; // create ostringstream instance
16
17 string string1("Output of several data types ");
18 string string2("to an ostringstream object:");
19 string string3("\n double: ");
20 string string4("\n int: ");
21 string string5("\naddress of int: ");
22
23 double double1 = 123.4567;
24 int integer = 22;
25
26 // output strings, double and int to ostringstream outputString
27 outputString << string1 << string2 << string3 << double1
28 << string4 << integer << string5 << &integer;
29
```

```

30 // call str to obtain string contents of the ostringstream
31 cout << "outputString contains:\n" << outputString.str();
32
33 // add additional characters and call str to output string
34 outputString << "\nmore characters added";
35 cout << "\n\nafter additional stream insertions,\n"
36 << "outputString contains:\n" << outputString.str() << endl;
37 return 0;
38 } // end main

```

```

outputString contains:
Output of several data types to an ostringstream object:
 double: 123.457
 int: 22
address of int: 0012F540

after additional stream insertions,
outputString contains:
Output of several data types to an ostringstream object:
 double: 123.457
 int: 22
address of int: 0012F540
more characters added

```

Lines 2728 output string string1, string string2, string string3, double double1, string string4, int integer, string string5 and the address of int integer all to outputString in memory. Line 31 uses the stream insertion operator and the call outputString.str() to display a copy of the string created in lines 2728. Line 34 demonstrates that more data can be appended to the string in memory by simply issuing another stream insertion operation to outputString. Lines 3536 display string outputString after appending additional characters.

---

[Page 904]

An istringstream object inputs data from a string in memory to program variables. Data is stored in an istringstream object as characters. Input from the istringstream object works identically to input from any file. The end of the string is interpreted by the istringstream object as end-of-file.

Figure 18.12 demonstrates input from an istringstream object. Lines 1516 create string input containing the data and istringstream object inputString constructed to contain the data in

string input. The string input contains the data

Input test 123 4.7 A

[Page 905]

which, when read as input to the program, consist of two strings ("Input" and "test"), an int (123), a double (4.7) and a char ('A'). These characters are extracted to variables string1, string2, integer, double1 and character in line 23.

**Figure 18.12. Demonstrating input from an `istringstream` object.**

(This item is displayed on pages 904 - 905 in the print version)

```

1 // Fig. 18.12: Fig18_12.cpp
2 // Demonstrating input from an istringstream object.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 #include <sstream>
11 using std::istringstream;
12
13 int main()
14 {
15 string input("Input test 123 4.7 A");
16 istringstream inputString(input);
17 string string1;
18 string string2;
19 int integer;
20 double double1;
21 char character;
22
23 inputString >> string1 >> string2 >> integer >> double1 >> character;
24
25 cout << "The following items were extracted\n"
26 << "from the istringstream object:" << "\nstring: " << string1
27 << "\nstring: " << string2 << "\n int: " << integer
28 << "\ndouble: " << double1 << "\n char: " << character;
29
30 // attempt to read from empty stream

```

```
31 long value;
32 inputString >> value;
33
34 // test stream results
35 if (inputString.good())
36 cout << "\n\nlong value is: " << value << endl;
37 else
38 cout << "\n\ninputString is empty" << endl;
39
40 return 0;
41 } // end main
```

The following items were extracted from the `istringstream` object:

string: Input

string: test

int: 123

double: 4.7

char: A

inputString is empty

The data is then output in lines 2528. The program attempts to read from `inputString` again in line 32. The `if` condition in line 35 uses function `good` ([Section 15.8](#)) to test if any data remains. Because no data remains, the function returns `false` and the `else` part of the `if...else` statement is executed.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 905 (continued)]

## 18.13. Wrap-Up

This chapter introduced the class `string` from the C++ Standard Library, which allows programs to treat strings as full-fledged objects. We discussed assigning, concatenating, comparing, searching and swapping strings. We also introduced a number of methods to determine string characteristics, find, replace and insert characters in a string and convert strings to C-style strings and vice versa. You also learned about string iterators and performing input from and output to strings in memory. In the next chapter, you will learn how to write CGI scripts with C++ to create dynamic Web pages with C++.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 907]

- Class `string` provides member functions `end` and `begin` to iterate through individual elements.
- Class `string` provides member functions `rrend` and `rbegin` for accessing individual `string` characters in reverse from the end of a `string` toward the beginning.
- Input from a `string` is supported by type `istringstream`. Output to a `string` is supported by type `ostringstream`.
- `ostringstream` member function `str` returns a `string` copy of a `string`.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 907 (continued)]

## Terminology

append member function of class `string`

assign member function of class `string`

at member function of class `string`

`basic_string` class template

begin member function of class `string`

`c_str` member function of class `string`

capacity of a string

capacity member function of class `string`

checked access

compare member function of class `string`

`const_iterator`

`const_reverse_iterator`

copy member function of class `string`

data member function of class `string`

empty string

end member function of class string

erase member function of class string

find member function of class string

find\_first\_not\_of member function of class string

find\_first\_of member function of class string

find\_last\_of member function of class string

getline member function of class string

in-memory I/O

insert member function of class string

istringstream class

iterator

length member function of class string

length of a string

lexicographical comparison

max\_size member function of class string

maximum size of a string

ostringstream class

range checking

rbegin member function of class string

rend member function of class string

replace member function of class string

resize member function of class string

reverse\_iterator

rfind member function of class string

size member function of class string

<sstream> header file

str member function of class ostringstream

string::npos constant

string stream processing

substr member function of class string

swap member function of class string

wchar\_t type

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 908]

## 18.3

Find the error(s) in each of the following, and explain how to correct it (them):

a.

```
string string1(28) ; // construct string1
string string2('z') ; // construct string2
```

b.

```
// assume std namespace is known
const char *ptr = name.data() ; // name is "joe bob"
ptr[3] = '-' ;
cout << ptr << endl;
```

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 908 (continued)]

## Answers to Self-Review Exercises

**18.1** a)<string>. b) std. c) erase. d) find\_first\_of.

**18.2**      **a.**

True.

**b.**

True.

**c.**

True.

**d.**

False. A string is an object that provides many different services. A C-style string does not provide any services. C-style strings are null terminated; strings are not necessarily null terminated. C-style strings are pointers and strings are not.

**18.3****a.**

Constructors for class `string` do not exist for integer and character arguments. Other valid constructors should be used converting the arguments to `strings` if need be.

**b.**

Function `data` does not add a null terminator. Also, the code attempts to modify a `const char`. Replace all of the lines with the code:

```
cout << name.substr(0, 3) + "-" + name.substr(4) << endl;
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 909]

•

Write a program that decrypts the scrambled message using 13 as the key.

•

After writing the programs of part (a) and part (b), briefly answer the following question: If you did not know the key for part (b), how difficult do you think it would be to break the code? What if you had access to substantial computing power (e.g., supercomputers)? In [Exercise 18.26](#) we ask you to write a program to accomplish this.

### **18.8**

Write a program using iterators that demonstrates the use of functions `rbegin` and `rend`.

### **18.9**

Write a program that reads in several `strings` and prints only those ending in "r" or "ay". Only lowercase letters should be considered.

### **18.10**

Write a program that demonstrates passing a `string` both by reference and by value.

### **18.11**

Write a program that separately inputs a first name and a last name and concatenates the two into a new `string`.

### **18.12**

Write a program that plays the game of Hangman. The program should pick a word (which is either coded directly into the program or read from a text file) and display the following:

Guess the word:               XXXXXX

Each x represents a letter. The user tries to guess the letters in the word. The appropriate response yes

or no should be displayed after each guess. After each incorrect guess, display the diagram with another body part filled. After seven incorrect guesses, the user should be hanged. The display should look as follows:



After each guess, display all user guesses. If the user guesses the word correctly, the program should display

Congratulations!!! You guessed my word. Play again? yes/no

### 18.13

Write a program that inputs a string and prints the string backward. Convert all uppercase characters to lowercase and all lowercase characters to uppercase.

### 18.14

Write a program that uses the comparison capabilities introduced in this chapter to alphabetize a series of animal names. Only uppercase letters should be used for the comparisons.

### 18.15

Write a program that creates a cryptogram out of a string. A cryptogram is a message or word in which each letter is replaced with another letter. For example the string

The bird was named squawk

might be scrambled to form

cin vrjs otz ethns zxqtop

Note that spaces are not scrambled. In this particular case, 'T' was replaced with 'x', each 'a' was replaced with 'h', etc. Uppercase letters become lowercase letters in the cryptogram. Use techniques similar to those in [Exercise 18.7](#).

### 18.16

Modify [Exercise 18.15](#) to allow the user to solve the cryptogram. The user should input two characters at a time: The first character specifies a letter in the cryptogram, and the second letter specifies the replacement letter. If the replacement letter is correct, replace the letter in the cryptogram with the replacement letter in uppercase.

---

[Page 910]

### 18.17

Write a program that inputs a sentence and counts the number of palindromes in it. A palindrome is a word that reads the same backward and forward. For example, "TRee" is not a palindrome, but "noon" is.

### 18.18

Write a program that counts the total number of vowels in a sentence. Output the frequency of each vowel.

### 18.19

Write a program that inserts the characters "\*\*\*\*\*" in the exact middle of a string.

### 18.20

Write a program that erases the sequences "by" and "BY" from a string.

### 18.21

Write a program that inputs a line of text, replaces all punctuation marks with spaces and uses the C-string library function `strtok` to tokenize the string into individual words.

### 18.22

Write a program that inputs a line of text and prints the text backwards. Use iterators in your solution.

### 18.23

Write a recursive version of [Exercise 18.22](#).

### 18.24

Write a program that demonstrates the use of the `erase` functions that take iterator arguments.

**18.25**

Write a program that generates the following from the string "abcdefghijklmnopqrstuvwxyz{":

```

 a
 bcb
 cdedc
 defgfed
efghihgfe
fghijkjihgf
ghijklmlkjihg
hijklmnnonmlkjih
ijklmnopqponmlkji
jklnopqrstuvwxyz{ qponmlkj
klmnopqrstutsrqponmlk
lmnopqrstuvwxyzwvutsrqponml
mnopqrstuvwxyz{ zyxwvutsrqpon
nopqrstuvwxyz{ zyxwvutsrqpon

```

**18.26**

In Exercise 18.7, we asked you to write a simple encryption algorithm. Write a program that will attempt to decrypt a "rot13" message using simple frequency substitution. (Assume that you do not know the key.) The most frequent letters in the encrypted phrase should be replaced with the most commonly used English letters (a, e, i, o, u, s, t, r, etc.). Write the possibilities to a file. What made the code breaking easy? How can the encryption mechanism be improved?

**18.27**

Write a version of the selection sort routine (Fig. 8.28) that sorts strings. Use function swap in your solution.

**18.28**

Modify class Employee in Figs. 13.613.7 by adding a private utility function called isValidSocialSecurityNumber. This member function should validate the format of a social security number (e.g., ###-##-####, where # is a digit). If the format is valid, return true; otherwise return false.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 912]

## Outline

[19.1 Introduction](#)

[19.2 HTTP Request Types](#)

[19.3 Multitier Architecture](#)

[19.4 Accessing Web Servers](#)

[19.5 Apache HTTP Server](#)

[19.6 Requesting XHTML Documents](#)

[19.7 Introduction to CGI](#)

[19.8 Simple HTTP Transactions](#)

[19.9 Simple CGI Scripts](#)

[19.10 Sending Input to a CGI Script](#)

[19.11 Using XHTML Forms to Send Input](#)

[19.12 Other Headers](#)

[19.13 Case Study: An Interactive Web Page](#)

[19.14 Cookies](#)

[19.15 Server-Side Files](#)

[19.16 Case Study: Shopping Cart](#)

[19.17 Wrap-Up](#)

## 19.18 Internet and Web Resources

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

[Exercises](#)

 PREV

NEXT 

**page footer**

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 913]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 914]

Request type

Description

delete

Such a request is normally used to delete a file from a server. This may not be available on some servers because of its inherent security risks (e.g., the client could delete a file that is critical to the execution of the server or an application).

head

Such a request is normally used when the client wants only the response's headers, such as its content type and content length.

options

Such a request returns information to the client indicating the HTTP options supported by the server, such as the HTTP version (1.0 or 1.1) and the request methods the server supports.

put

Such a request is normally used to store a file on the server. This may not be available on some servers because of its inherent security risks (e.g., the client could place an executable application on the server, which, if executed, could damage the server perhaps by deleting critical files or occupying resources).

TRace

Such a request is normally used for debugging. The implementation of this method automatically returns an XHTML document to the client containing the request header information (data sent by the browser as part of the request).

[Page 914]

An HTTP request often sends data to a **server-side form handler** program that resides on the Web server and is created by a server-side programmer to handle client requests. For example, when a user

participates in a Web-based survey, the Web server receives the information specified in the form as part of the request, and the form handler processes the survey. We demonstrate how to create server-side form handlers throughout the examples in this chapter.

Browsers often **cache** (i.e., save on a local disk) Web pages for quick reloading, to reduce the amount of data that the browser needs to download over the Internet. Web browsers often cache the server's responses to get requests. A static Web page, such as a course syllabus, is cached in the event that the user requests the same resource again. However, browsers typically do not cache the responses to post requests, because subsequent post requests might not contain the same information. For example, in an online survey, many users could visit the same Web page and respond to a question. The page could also display the survey results. Each new response changes the overall results of the survey, so any requests to view the survey results should be sent using the post method. Similarly, the post method should be used to request a Web page containing discussion forum posts, as these posts may change frequently. Otherwise, the browser may cache the results after the user's first visit and display these same results for each subsequent visit.



page footer



---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 915]

Business logic in the middle tier enforces **business rules** and ensures that data is reliable before updating the database or sending data to a user. Business rules dictate how clients can and cannot access application data and how applications process data. For example, a business rule can specify how to convert numeric grades to letter grades.

The **top tier** (also called the **client tier**) is the application's user interface. Users interact directly with the application through the user interface. The client tier interacts with the middle tier to make requests and to retrieve data from the information tier. The client tier then displays to the user the data retrieved from the middle tier.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 916]

Fully qualified domain name: A **fully qualified domain name (FQDN)**, also known as the machine name, contains a host (for example, `www` for World Wide Web) and a domain name, including a **top-level domain (TLD)**. The top-level domain is the last and most significant component of a fully qualified domain name.

Local Web servers can be accessed in two ways: through the machine name, or through `localhost` a host name that references the local machine. We use `localhost` in this book for demonstration purposes. To determine the machine name in Windows 98, right click Network Neighborhood, and select Properties from the context menu to display the Network dialog. In the Network dialog, click the Identification tab. The computer name displays in the Computer name: field. Click Cancel to close the Network dialog. In Windows 2000, right click My Network Places and select Properties from the context menu to display the Network and Dialup Connections explorer. In the explorer, click Network Identification. The Full computer name: field in the System Properties window displays the computer name. In Windows XP, select Start > Control Panel, which displays the Control Panel window. Double click System in the Control Panel window, which opens the System Properties window. Select the Computer Name tab in the System Properties window; the Full computer name: field displays the computer name.

To request documents from the Web server, users must know the fully qualified domain names (machine names) on which the Web server software resides. For example, to access the documents from Deitel's Web server, you must know the FQDN [www.deitel.com](http://www.deitel.com). The FQDN [www.deitel.com](http://www.deitel.com) indicates that the host is `www` and the toplevel domain is `com`. In a FQDN, the TLD often describes the type of organization that owns the domain. For example, usually the `com` TLD refers to a commercial business, the `org` TLD to a nonprofit organization and the `edu` TLD to an educational institution. In addition, each country has its own TLD, such as `cn` for China, `et` for Ethiopia, `om` for Oman and `us` for the United States.

Each FQDN corresponds to a numeric address called an **IP (Internet Protocol) address**, which is much like a street address. Just as people use street addresses to locate houses or businesses in a city, computers use IP addresses to locate other computers on the Internet. Each internet host computer has a unique IP address. Each address comprises four sets of numbers separated by periods, such as `63.110.43.82`. A **Domain Name System (DNS) server** is a computer that maintains a database of FQDNs and their corresponding IP addresses. The process of translating FQDNs to IP addresses is called a **DNS lookup**. For example, to access the Deitel Web site, type the FQDN [www.deitel.com](http://www.deitel.com) into a Web browser. The DNS lookup translates [www.deitel.com](http://www.deitel.com) into the IP address of the Deitel Web server (`63.110.43.82`). The IP address of `localhost` is always `127.0.0.1`. This address, also known as the **loopback address**, can be used to test Web applications on your local computer.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 917]

To download the Apache HTTP server, visit [httpd.apache.org](http://httpd.apache.org).<sup>[2]</sup> For instructions on installing Apache, visit [www.deitel.com](http://www.deitel.com) or [httpd.apache.org](http://httpd.apache.org). If the Apache HTTP server is installed as a service, then it is already running after installation. Otherwise, start the server by selecting the Start menu, then All Programs > Apache HTTP Server 2.0.52 > Control Apache Server > Start. To stop the Apache HTTP server, select Start > All Programs > Apache HTTP Server 2.0.52 > Control Apache Server > Stop. For Linux users, we put instructions on how to start/stop the Apache HTTP server and run the examples at our Web site, [www.deitel.com](http://www.deitel.com).

<sup>[2]</sup> In this chapter, we use version 2.0.52.

 PREV

NEXT 

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 917 (continued)]

## 19.6. Requesting XHTML Documents

This section shows how to request an XHTML document from the Apache HTTP server. In the Apache HTTP server directory structure, XHTML documents must be saved in the **htdocs** directory. On Windows platforms, the **htdocs** directory resides in `C:\Program Files\Apache Group\Apache2`. On Linux platforms, the **htdocs** directory resides in the `/usr/local/httpd` directory.<sup>[3]</sup> Copy the `test.html` document from the [Chapter 19](#) examples directory on the book's CD-ROM into the **htdocs** directory. To request the document, launch a Web browser, such as Internet Explorer or Netscape, and enter the URL in the Address field (i.e., `http://localhost/test.html`). [Figure 19.3](#) shows the result of requesting `test.html`. [Note: In Apache, the root directory of the Web server refers to the default directory, **htdocs**, so we do not enter the directory name before the file name (i.e., `test.html`) in the Address field.]

<sup>[3]</sup> Linux users may already have Apache installed by default. The **htdocs** directory may be found in a number of places, depending on the Linux distribution.

**Figure 19.3. Requesting `test.html` from Apache.**

[\[View full size image\]](#)

[◀ PREV](#)[NEXT ▶](#)

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 918]

The Common Gateway Interface is "common" in the sense that it is not specific to any particular operating system (such as Linux or Windows) or to any one programming language. CGI was designed to be used with virtually any programming language, such as C, C++, Perl, Python or Visual Basic.

CGI was developed in 1993 by [NCSA \(National Center for Supercomputing Applications www.ncsa.uiuc.edu\)](#) for use with its popular **HTTPd Web server**. Unlike Web protocols and languages that have formal specifications, the initial concise description of CGI written by NCSA proved simple enough that CGI was adopted as an unofficial standard worldwide. CGI support was incorporated quickly into other Web servers, including Apache.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

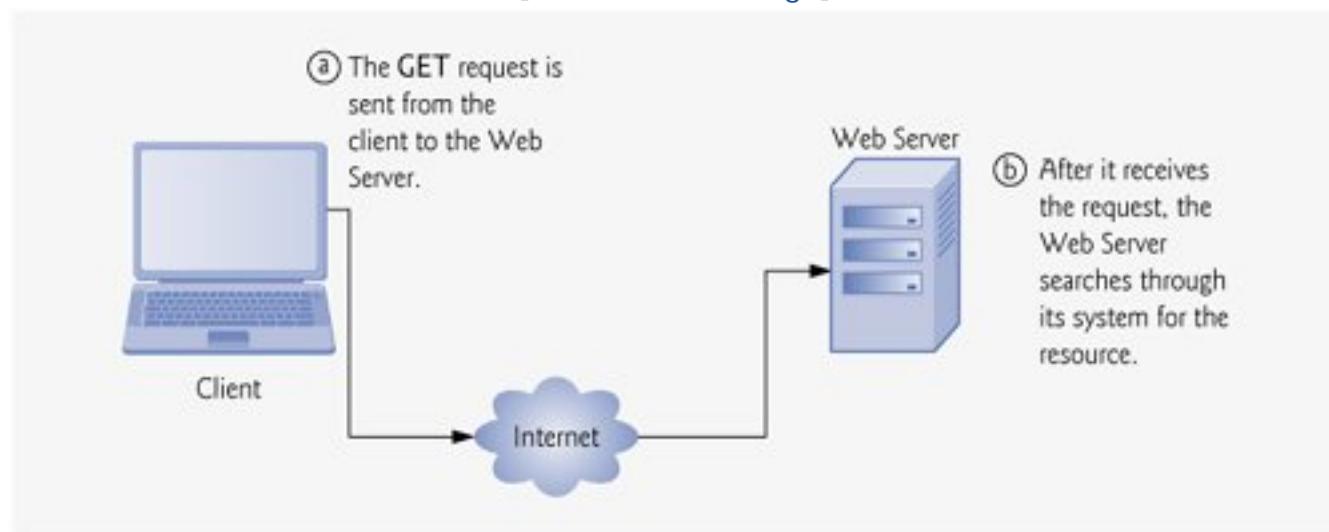
Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 919]

Now we consider how a browser, when given a URL, performs a simple HTTP transaction to retrieve and display a Web page. [Figure 19.4](#) illustrates the transaction in detail. The transaction is performed between a Web browser and a Web server.

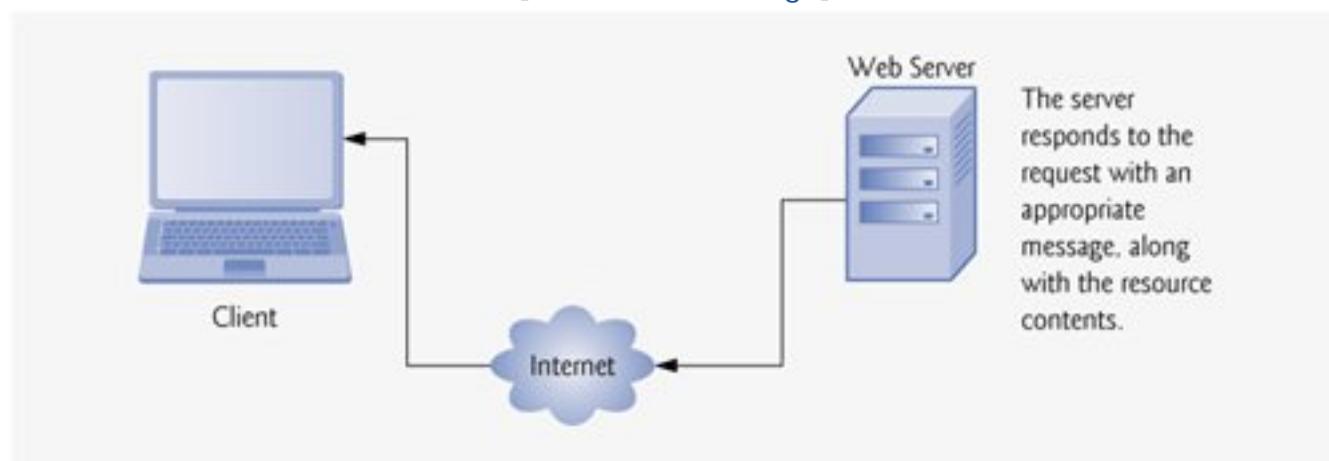
**Figure 19.4. Client interacting with server and Web server. Step 1: The get request, GET / books/downloads.html HTTP/1.1.**

[\[View full size image\]](#)



**Figure 19.4. Client interacting with server and Web server. Step 2: The HTTP response, HTTP/1.1 200 OK.**

[\[View full size image\]](#)



In Step 1 of Fig. 19.4, the browser sends an HTTP request to the server. The request (in its simplest form) looks like the following:

```
GET /books/downloads.html HTTP/1.1
Host: www.deitel.com
```

The word **GET**, an HTTP method, indicates that the client sends a `get` request and wishes to retrieve a resource. The remainder of the request provides the name and path of the resource (`/books/downloads.html`) and the protocol's name and version number (`HTTP/1.1`). After the Web server receives the request, it searches through the system for the resource.

---

[Page 920]

Any server that understands HTTP (version 1.1) will be able to translate this request and respond appropriately. Step 2 of Fig. 19.4 shows the results of a successful request. The server first sends a response indicating the HTTP version, followed by a numeric code and a phrase describing the status of the transaction. For example,

```
HTTP/1.1 200 OK
```

indicates success;

```
HTTP/1.1 404 Not found
```

informs the client that the requested resource was not found on the server in the specified location.

The server then sends one or more HTTP headers, which provide information about the data being sent to the client. In this case, the server is sending an XHTML document, so the HTTP header reads

```
Content-Type: text/html
```

The information in the **Content-Type header** identifies the **MIME (Multipurpose Internet Mail Extensions) type** of the content. Each document from the server has a MIME type by which the browser determines how to process the data it receives. For example, the MIME type `text/plain` indicates that the data contains text that should be displayed without attempting to interpret any of the content as XHTML markup. Similarly, the MIME type `image/gif` indicates that the content is a GIF image. When this MIME type is received by the browser, it attempts to display the data as an image.

The headers are followed by a blank line, which indicates to the client that the server has finished sending

HTTP headers. The server then sends the contents of the requested document (e.g., `downloads.html`). The connection is terminated when the transfer of the resource is complete (in this case, when the end of the document `downloads.html` is reached). The client-side browser interprets the XHTML it receives and renders (or displays) the results.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 921]

`http://localhost/cgi-bin/localtime.cgi`

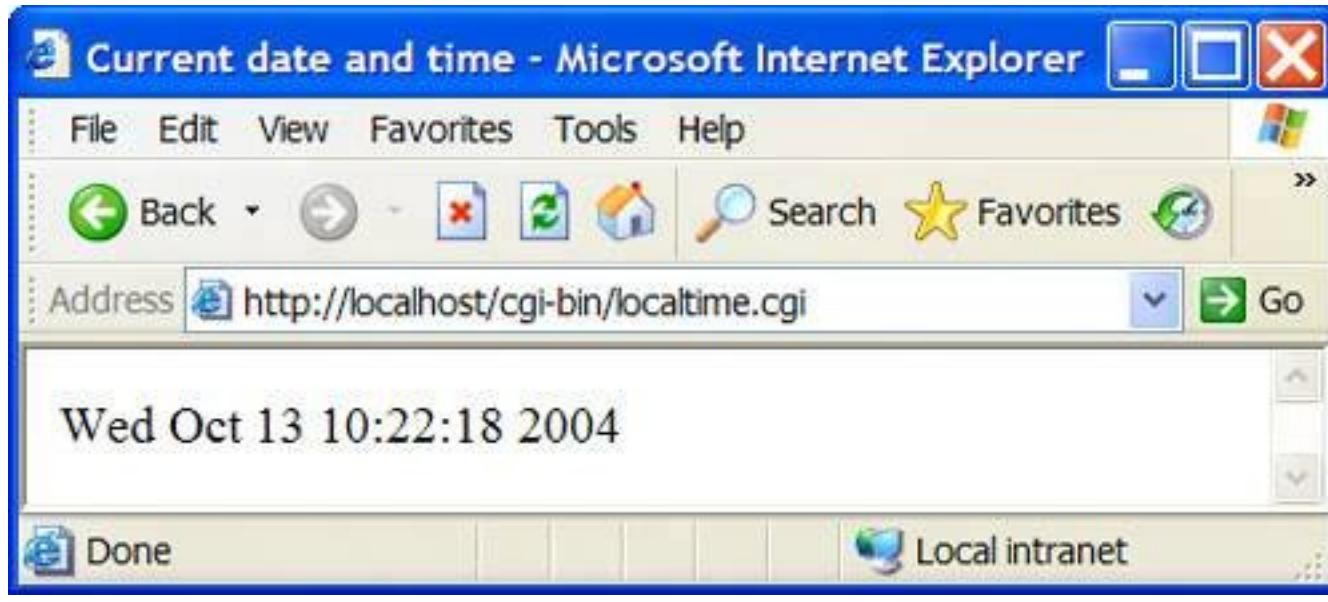
in your browser's Address or Location field. If you are requesting this script from a remote Web server, you will need to replace `localhost` with the server's machine name or IP address.

**Figure 19.5. First CGI script.**

```

1 // Fig. 19.5: localtime.cpp
2 // Displays the current date and time in a Web browser.
3 #include <iostream>
4 using std::cout;
5
6 #include <ctime> // definitions of time_t, time, localtime and asctime
7 using std::time_t;
8 using std::time;
9 using std::localtime;
10 using std::asctime;
11
12 int main()
13 {
14 time_t currentTime; // variable for storing time
15
16 cout << "Content-Type: text/html\n\n"; // output HTTP header
17
18 // output XML declaration and DOCTYPE
19 cout << "<?xml version = \"1.0\"?>"
20 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
21 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\"> ";
22
23 time(¤tTime); // store time in currentTime
24
25 // output html element and some of its contents
26 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
27 << "<head><title>Current date and time</title></head>"
28 << "<body><p>" << asctime(localtime(¤tTime))
29 << "</p></body></html>" ;
30
31 return 0;
32 } // end main

```

[\[View full size image\]](#)

[Page 922]

The notion of standard output is similar to that of standard input, which we have seen associated with `cin`. Just as standard input refers to the standard source of input into a program (normally, the keyboard), standard output refers to the standard destination of output from a program (normally, the screen). It is possible to redirect (or **pipe**) standard output to another destination. Thus, in our CGI script, when we output an HTTP header (line 16) or XHTML elements (lines 1921 and 2629), the output is sent to the Web server, as opposed to the screen. The server sends that output to the client over HTTP, which interprets the headers and elements as if they were part of a normal server response to an XHTML document request.

It is fairly straightforward to write a C++ program that outputs the current time and date (to the monitor of the local computer). In fact, this requires only a few lines of code (lines 14, 23 and 28). Line 14 declares `currentTime` as a variable of type `time_t`. Function `time` (line 23) gets the current time, which is represented as the number of seconds elapsed since midnight January 1, 1970, and stores the retrieved value to the location specified by the parameter (in this case, `currentTime`). C++ library function `localtime` (line 28), when passed a `time_t` variable (e.g., `currentTime`), returns a pointer to an object containing the "broken-down" local time (i.e., days, hours, etc. are placed in individual member variables). Function `asctime` (line 28), which takes a pointer to an object containing "broken-down" time, returns a string such as

Wed Oct 31 13:10:37 2004

What if we wish to send the current time to a client's browser window for display (rather than outputting it to the screen)? CGI makes this possible by redirecting the output of a program to the Web server itself, which then sends the output to a client's browser.

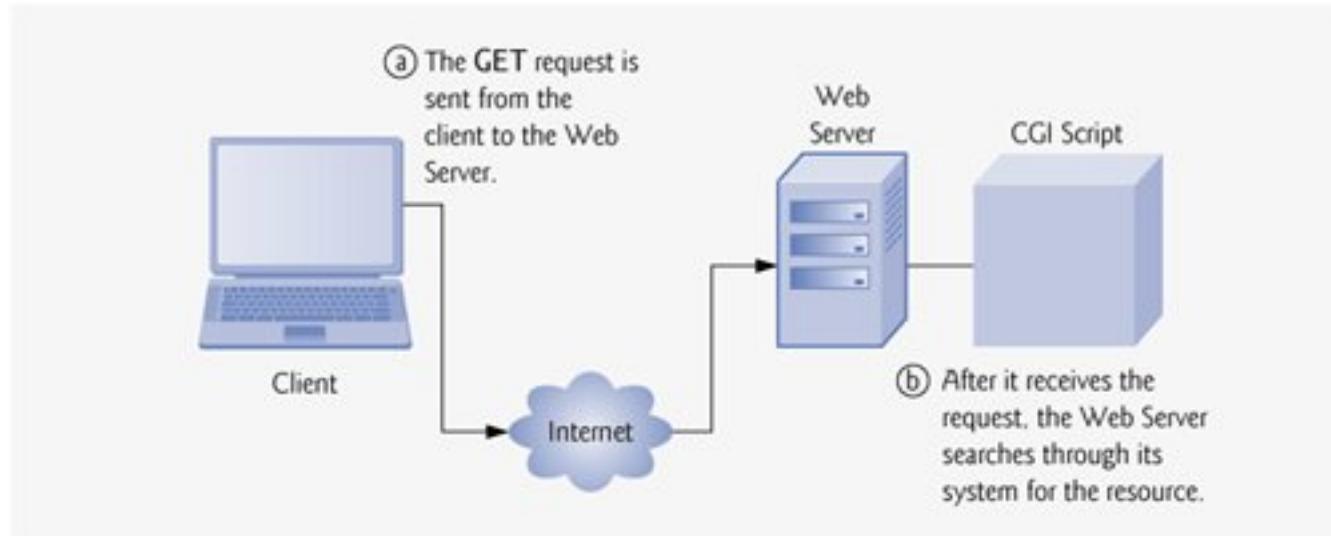
## How Web Server Redirects the Output

Figure 19.6 illustrates this process in more detail. In Step 1, the client requests the resource named `localtime.cgi` from the server, just as it requested `downloads.html` in the previous example (Fig. 19.4). If the server was not configured to handle CGI scripts, it might just return the contents of the C++ executable file to the client, as if it were any other document. However, based on the Web server configuration, the server executes `localtime.cgi` (implemented using C++) and sends the CGI script's output to the Web browser.

**Figure 19.6. Step 1: The get request, GET /cgi-bin/localtime.cgi HTTP/1.1.**

(This item is displayed on page 923 in the print version)

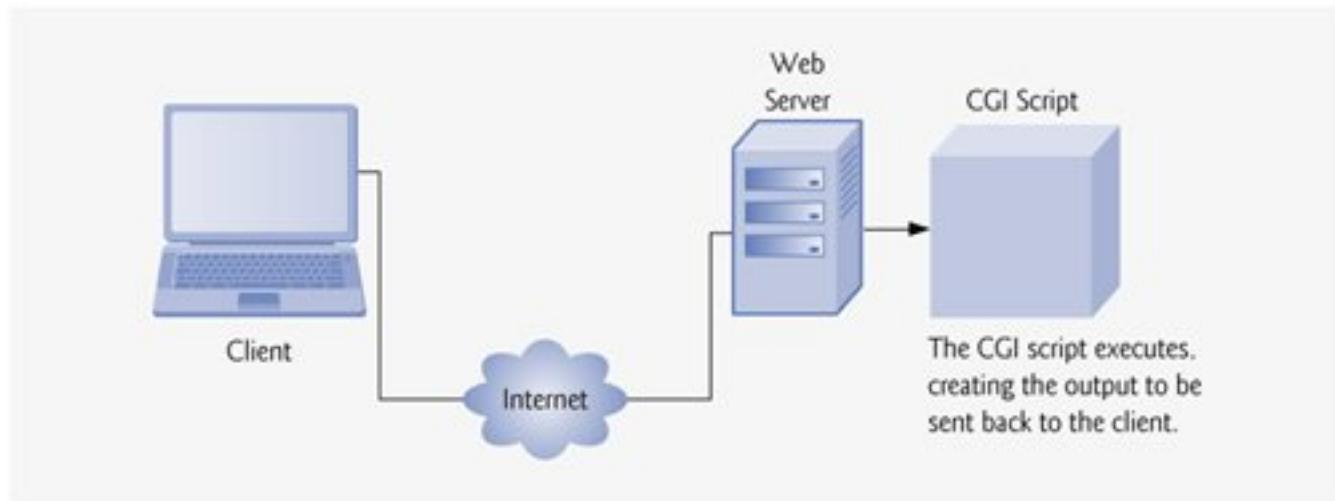
[View full size image]



**Figure 19.6. Step 2: The Web server starts the CGI script.**

(This item is displayed on page 923 in the print version)

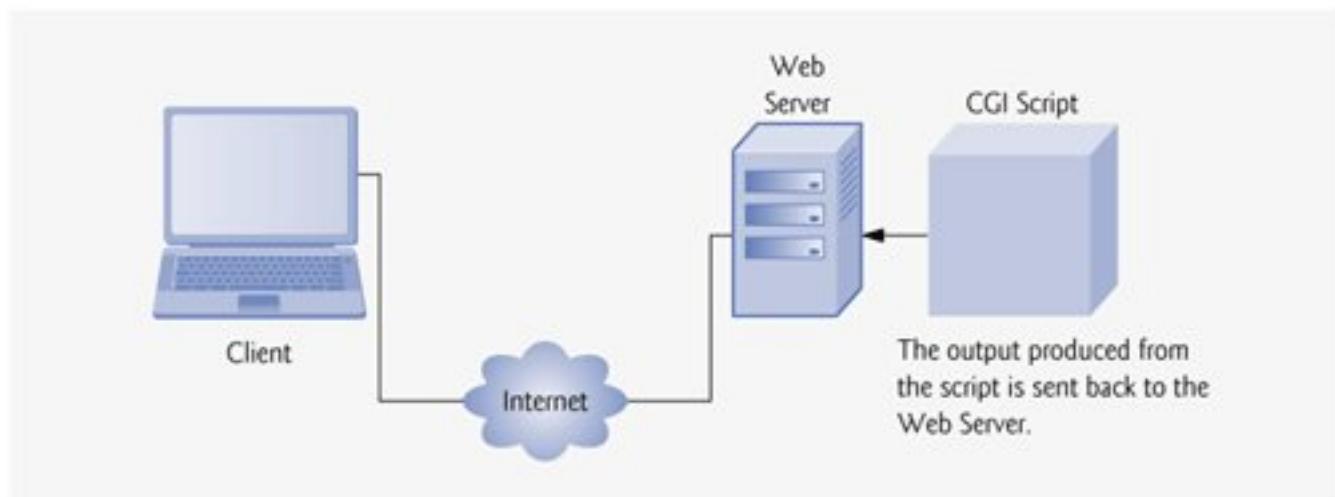
[View full size image]



**Figure 19.6. Step 3: The script output is sent to the Web server.**

(This item is displayed on page 923 in the print version)

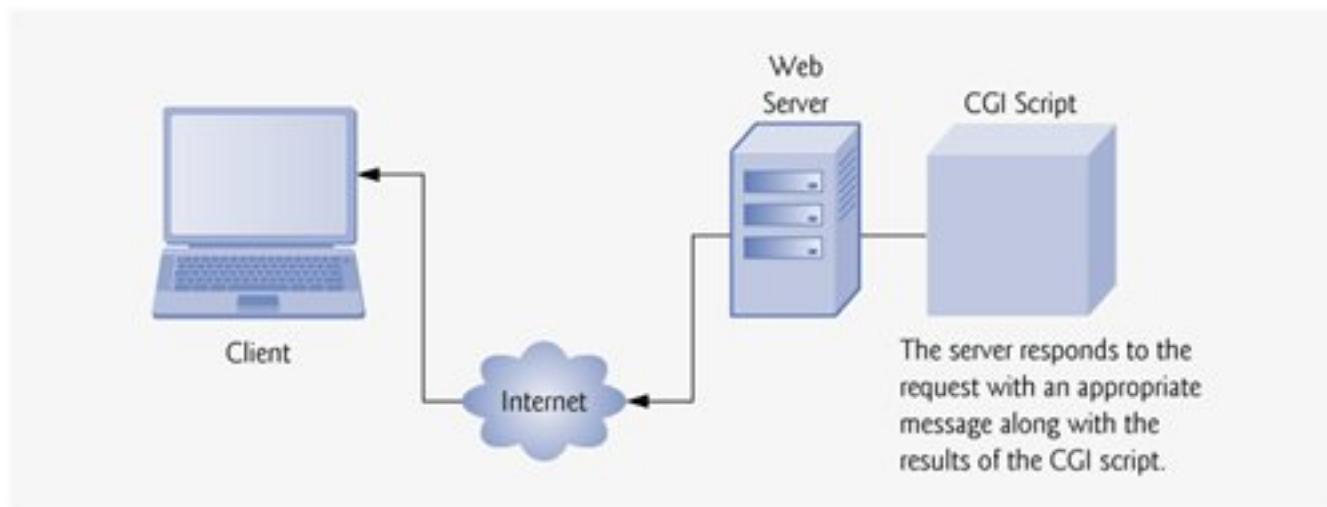
[View full size image]



**Figure 19.6. Step 4: The HTTP response, HTTP/1.1 200 OK.**

(This item is displayed on page 924 in the print version)

[View full size image]



A properly configured Web server, however, will recognize that different types of resources should be handled differently. For example, when the resource is a CGI script, the script must be executed by the server before it is sent. A CGI script is designated in one of two ways: either it has a special filename extension (such as `.cgi` or `.exe`) or it is located in a specific directory (often `cgi-bin`). In addition, the server administrator must give permission explicitly for remote clients to be able to access and execute CGI scripts. [5]

[5] If you are using the Apache HTTP Server and would like more information on configuration, consult the Apache home page at <http://httpd.apache.org/docs-2.0/>.

In Step 2 of Fig. 19.6, the server recognizes that the resource is a CGI script and executes the script. In Step 3, the output produced by the script's three cout statements (lines 16, 1921 and 2629 of Fig. 19.5) is sent to the standard output and is returned to the Web server. Finally, in Step 4, the Web server adds a message to the output that indicates the status of the HTTP transaction (such as `HTTP/1.1 200 OK`, for success) and sends the entire output from the CGI program to the client.

---

[Page 923]

The client-side browser then processes the XHTML document and displays the results. It is important to note that the browser is unaware of what has transpired on the server. In other words, as far as the browser is concerned, it requests a resource like any other and receives a response like any other. The browser receives and interprets the script's output just as if it were a simple, static XHTML document.

---

[Page 924]

In fact, you can view the content that the browser receives by executing `localtime.cgi` from the command line, as we normally would execute any of the programs from the previous chapters. [Note: The filename extension must be changed to `.exe` prior to executing it from the command line on a system

running Windows.] Figure 19.7 shows the output. For the purpose of this chapter, we formatted the output for human readability. Notice that, with the CGI script, we must output the Content-Type header, whereas, for an XHTML document, the Web server would include the header.

**Figure 19.7. Output of localtime.cgi when executed from the command line.**

```
Content-Type: text/html

<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns = "http://www.w3.org/1999/xhtml">
 <head>
 <title>Current date and time</title>
 </head>

 <body>
 <p>Wed Oct 13 10:22:18 2004</p>
 </body>
</html>
```

The CGI script prints the Content-Type header, a blank line and the data (XHTML, plain text, etc.) to standard output. When the CGI script is executed on the Web server, the Web server retrieves the script's output, inserts the HTTP response to the beginning and delivers the content to the client. Later we will see other content types that may be used in this manner, as well as other headers that may be used in addition to Content-Type.

### Common Programming Error 19.1



Forgetting to place a blank line after a header is a syntax error.

The program of Fig. 19.8 outputs the **environment variables** that the Apache HTTP Server sets for CGI scripts. These variables contain information about the client and server environment, such as the type of Web browser being used and the location of a document on the server. Lines 1423 initialize an array of string objects with the names of the CGI environment variables. [Note: Environment variables are server-specific. Servers other than Apache HTTP Server may not provide all of these environment variables.] Line 37 begins the XHTML table in which the data will be displayed.

**Figure 19.8. Retrieving environment variables via function getenv.**

(This item is displayed on pages 925 - 927 in the print version)

```

1 // Fig. 19.8: environment.cpp
2 // Program to display CGI environment variables.
3 #include <iostream>
4 using std::cout;
5
6 #include <string>
7 using std::string;
8
9 #include <cstdlib>
10 using std::getenv;
11
12 int main()
13 {
14 string environmentVariables[24] = {
15 "COMSPEC", "DOCUMENT_ROOT", "GATEWAY_INTERFACE",
16 "HTTP_ACCEPT", "HTTP_ACCEPT_ENCODING",
17 "HTTP_ACCEPT_LANGUAGE", "HTTP_CONNECTION",
18 "HTTP_HOST", "HTTP_USER_AGENT", "PATH",
19 "QUERY_STRING", "REMOTE_ADDR", "REMOTE_PORT",
20 "REQUEST_METHOD", "REQUEST_URI", "SCRIPT_FILENAME",
21 "SCRIPT_NAME", "SERVER_ADDR", "SERVER_ADMIN",
22 "SERVER_NAME", "SERVER_PORT", "SERVER_PROTOCOL",
23 "SERVER_SIGNATURE", "SERVER_SOFTWARE" };
24
25 cout << "Content-Type: text/html\n\n"; // output HTTP header
26
27 // output XML declaration and DOCTYPE
28 cout << "<?xml version = \"1.0\"?>"
29 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
30 << "\\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\\>" ;
31
32 // output html element and some of its contents
33 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\\>"
34 << "<head><title>Environment Variables</title></head><body>" ;
35

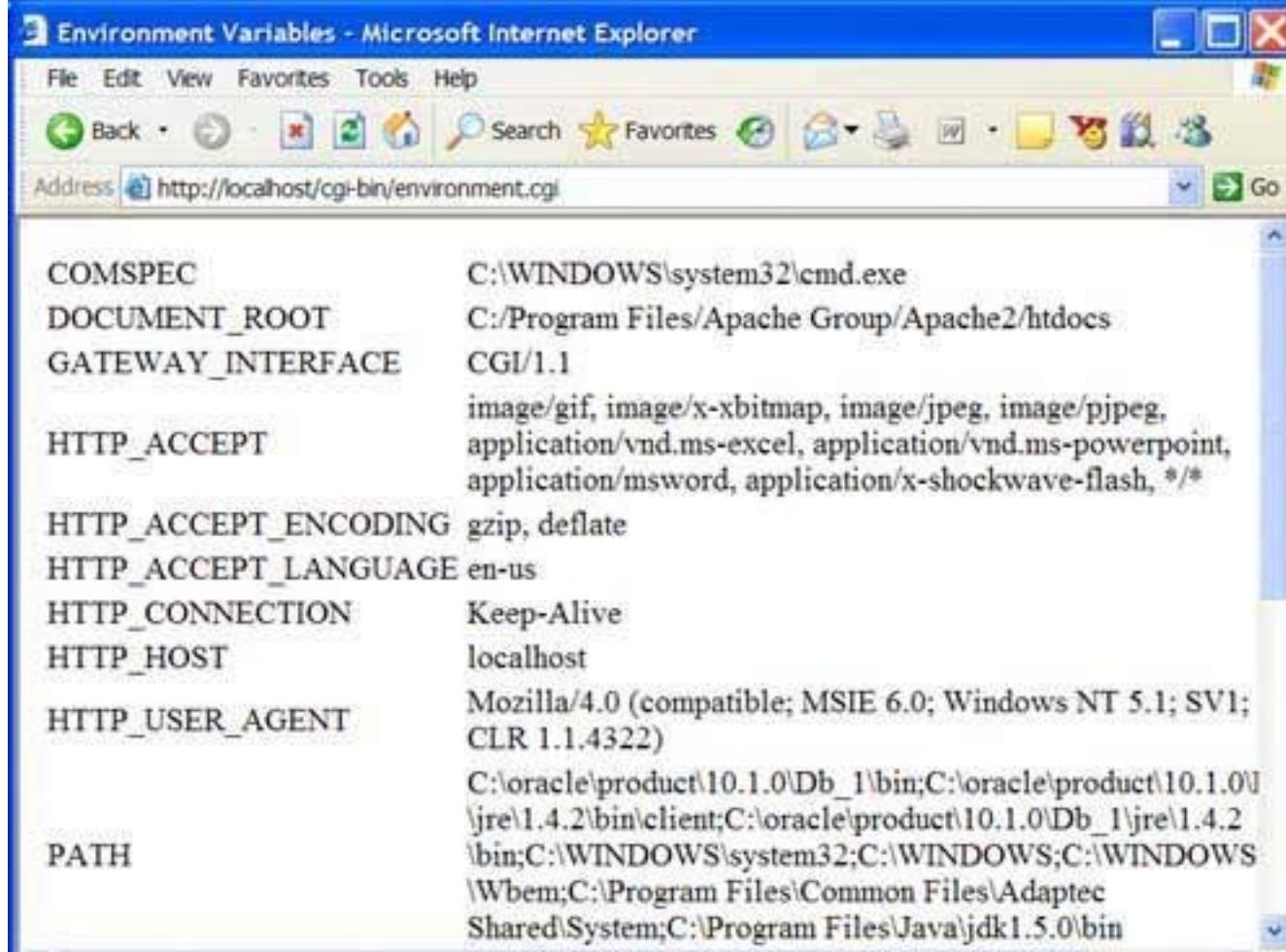
```

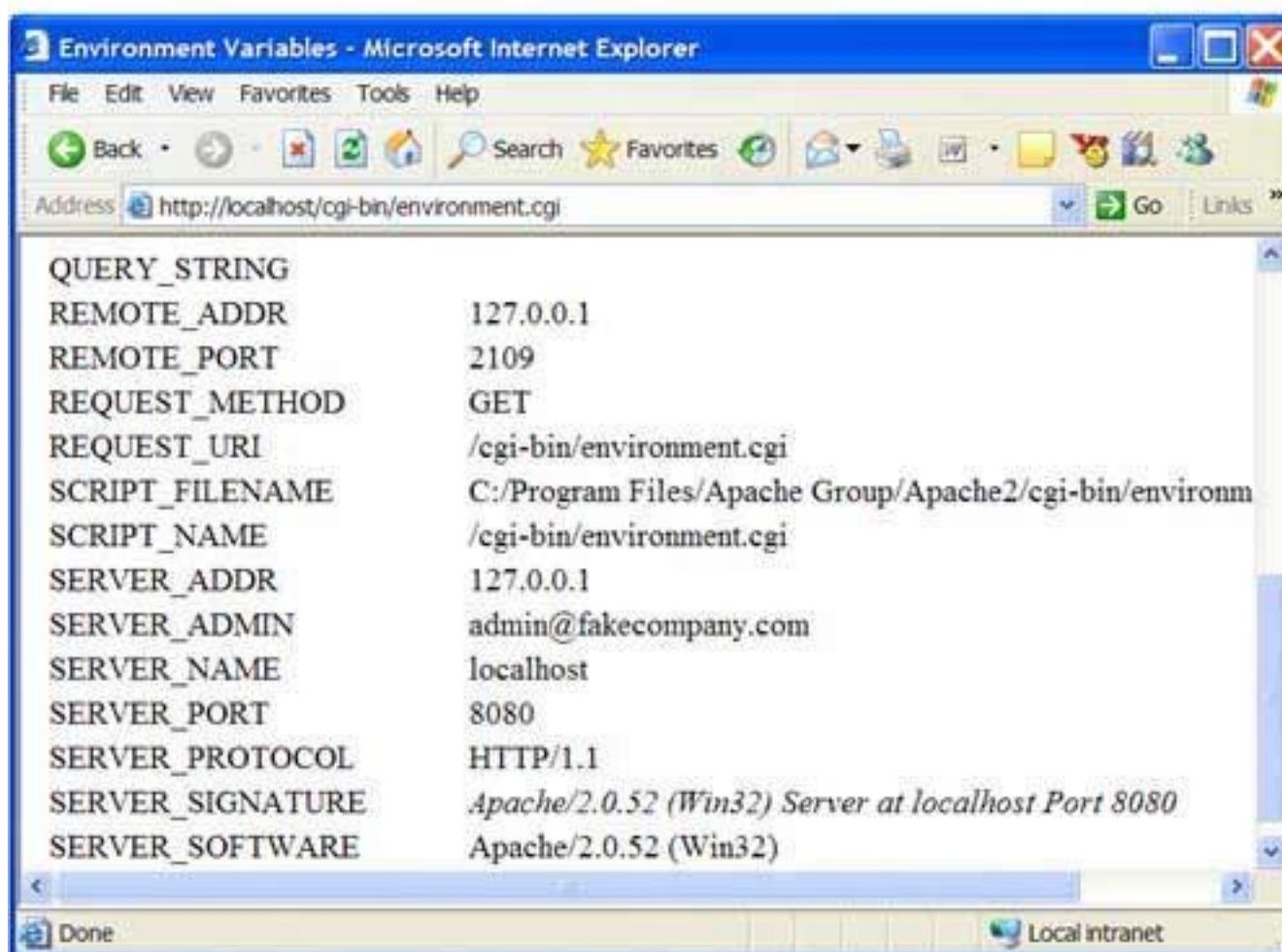
```

36 // begin outputting table
37 cout << "<table border = \"0\" cellspacing = \"2\">" ;
38
39 // iterate through environment variables
40 for (int i = 0; i < 24; i++)
41 {
42 cout << "<tr><td>" << environmentVariables[i] << "</td><td>" ;
43
44 // attempt to retrieve value of current environment variable
45 char *value = getenv(environmentVariables[i].c_str());
46
47 if (value != 0) // environment variable exists
48 cout << value;
49 else
50 cout << "Environment variable does not exist." ;
51
52 cout << "</td></tr>" ;
53 } // end for
54
55 cout << "</table></body></html>" ;
56 return 0;
57 } // end main

```

[View full size image]





[Page 927]

Lines 4252 output each row of the table. Let us examine each of these lines closely. Line 42 outputs an XHTML `<tr>` (table row) start tag, which indicates the beginning of a new table row. Line 52 outputs a corresponding `</tr>` end tag, which indicates the end of the row. Each row of the table contains two table cells—the name of an environment variable and the data associated with that variable (if the variable exists). The `<td>` start tag (line 42) begins a new table cell. The `for` loop (lines 4053) iterates through each of the 24 `string` objects. Each environment variable's name is output in the left table cell (line 42). Line 45 attempts to retrieve the value associated with the environment variable by calling function `getenv` of `<cstdlib>` and passing it the string value returned from the function call `environmentVariables[i].c_str()`. Function `c_str` returns a C-style `char *` string containing the contents of the `environmentVariables[i]` string. Function `getenv` returns a `char *` string containing the value of a specified environment variable or a null pointer if the environment variable does not exist.

Lines 4750 output the contents of the right table cell. If the current environment variable exists (i.e., `getenv` did not return a null pointer), line 48 outputs the value returned by function `getenv`. If the environment variable does not exist on the server executing the script, line 50 outputs an appropriate message. The sample execution shown in Fig. 19.8 was produced by running this example on Apache HTTP Server, so the output contains data associated with each of the environment variables. The results on other servers may vary. For example, if you were to run this example on Microsoft Internet Information Services (IIS), several of the table cells in the right column would display the message "Environment variable does not exist."

---

[Page 928]

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[PREV](#)

[NEXT](#)

[Page 928 (continued)]

## 19.10. Sending Input to a CGI Script

Though preset environment variables provide much information, we would like to be able to supply any type of data to our CGI scripts, such as a user's name or a search-engine query. The environment variable `QUERY_STRING` provides a mechanism to do just that. The `QUERY_STRING` variable contains information that is appended to a URL in a get request. For example, the URL

[www.somesite.com/cgi-bin/script.cgi?state=California](http://www.somesite.com/cgi-bin/script.cgi?state=California)

causes the Web browser to request a CGI script (`cgi-bin/script.cgi`) with query string (`state=California`) from [www.somesite.com](http://www.somesite.com). The Web server stores the query string following the `?` in the `QUERY_STRING` environment variable. The query string provides parameters that customize the request for a particular client. Note that the question mark `(?)` is not part of the resource requested, nor is it part of the query string. It serves as a delimiter (or separator) between the two.

[Figure 19.9](#) shows a simple example of a CGI script that reads data passed through the `QUERY_STRING`. The data in the query string can be formatted in a variety of ways. The CGI script reading the query string must know how to interpret the formatted data. In the example in [Fig. 19.9](#), the query string contains a series of name-value pairs delimited by ampersands `(&)`, as in `name=Jill&age=22`.

**Figure 19.9. Reading input from `QUERY_STRING`.**

(This item is displayed on pages 929 - 930 in the print version)

```

1 // Fig. 19.9: querystring.cpp
2 // Demonstrating QUERY_STRING.
3 #include <iostream>
4 using std::cout;
5
6 #include <string>
7 using std::string;
8
9 #include <cstdlib>
10 using::getenv;
11
12 int main()
13 {
14 string query = "";
15
16 if (getenv("QUERY_STRING")) // QUERY_STRING variable exists
17 query = getenv("QUERY_STRING"); // retrieve QUERY_STRING value
18
19 cout << "Content-Type: text/html\n\n"; // output HTTP header
20
21 // output XML declaration and DOCTYPE
22 cout << "<?xml version = \"1.0\"?"
23 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
24 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">" ;
25
26 // output html element and some of its contents
27 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
28 << "<head><title>Name/Value Pairs</title></head><body>" ;
29 cout << "<h2>Name/Value Pairs</h2>" ;
30
31 // if query contained no data
32 if (query == "")
33 cout << "Please add some name-value pairs to the URL above.
 Or"
34 << " try this." ;
35 else // user entered query string
36 cout << "<p>The query string is: " << query << "</p>" ;
37
38 cout << "</body></html>" ;
39 return 0;
40 } // end main

```







In line 16 of [Figure 19.9](#), we pass "QUERY\_STRING" to function `getenv`, which returns the query string or a null pointer if the server has not set a `QUERY_STRING` environment variable. [Note: The Apache HTTP Server sets `QUERY_STRING` even if a request does not contain a query string; in this case, the variable contains an empty string. However, some servers, such as Microsoft's IIS, set this variable only if a query string actually exists.] If the `QUERY_STRING` environment variable exists (i.e., `getenv` does not return a null pointer), line 17 invokes `getenv` again, this time assigning the returned query string to `string` variable `query`. After outputting a header, some XHTML start tags and the title (lines 1929), we test if `query` contains data (line 32). If not, we output a message instructing the user to add a query string to the URL. We also provide a link to a URL that includes a sample query string. Query string data may be specified as part of a hyperlink in a Web page when encoded in this manner. The contents of the query string are output by line 36.

This example simply demonstrated how to access data passed to a CGI script in the query string. Later chapter examples show how to break a query string into useful pieces of information that can be manipulated using separate variables.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 929]

## XHTML form Element

The **form element** encloses an XHTML form. The form element generally takes two attributes. The first is **action**, which specifies the server resource to execute when the user submits the form. For our purposes, the action usually will be a CGI script that processes the form's data. The second attribute used in the form element is **method**, which identifies the type of HTTP request (i.e., get or post) to use when the browser submits the form to the Web server. In this section, we show examples using both get and post requests to illustrate them in detail.

An XHTML form may contain any number of elements. [Figure 19.10](#) gives a brief description of several form elements.

[Page 931]

**Figure 19.10. XHTML form elements.**

Element name	type attribute value (for input elements)	Description
input	text	Provides a single-line text field for text input.
	password	Like text, but each character typed by the user appears as an asterisk (*).
	checkbox	Displays a checkbox that can be checked (TRUE) or unchecked (false).
	radio	Radio buttons are like checkboxes, except that only one radio button in a group of radio buttons can be selected at a time.
	button	A push button.
	submit	A push button that submits form data according to the form's action.

<code>image</code>	The same as submit, but displays an image rather than a push button.
<code>reset</code>	A push button that resets form fields to their default values.
<code>file</code>	Displays a text field and button that allow the user to specify a file to upload to a Web server. When clicked, the button opens a file dialog that allows the user to select a file.
<code>hidden</code>	Hidden form data that can be used by the form handler on the server. These inputs are not visible to the user.
<code>select</code>	Drop-down menu or selection box. This element is used with the option element to specify a series of selectable items.
<code>textarea</code>	This is a multiline text field in which text can be input or displayed.

## Using get Request

Figure 19.11 demonstrates a basic XHTML form using the HTTP get method. The form is output in lines 3436 with the `form` element. Notice that attribute `method` has the value "get" and attribute `action` has the value "getquery.cgi" (i.e., the script actually calls itself to handle the form data, once it is submitted).

### Figure 19.11. Using get method with an XHTML form.

(This item is displayed on pages 932 - 934 in the print version)

```

1 // Fig. 19.11: getquery.cpp
2 // Demonstrates GET method with XHTML form.
3 #include <iostream>
4 using std::cout;
5
6 #include <string>
7 using std::string;
8
9 #include <cstdlib>
10 using std::getenv;
11
12 int main()
13 {
14 string nameString = "";
15 string wordString = "";
16 string query = "";
17
18 if (getenv("QUERY_STRING")) // QUERY_STRING variable exists
19 query = getenv("QUERY_STRING"); // retrieve QUERY_STRING value
20
21 cout << "Content-Type: text/html\n\n"; // output HTTP header
22
23 // output XML declaration and DOCTYPE
24 cout << "<?xml version = \"1.0\"?>"
25 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
26 << "\\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\\>" ;
27
28 // output html element and some of its contents
29 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\\>"
30 << "<head><title>Using GET with Forms</title></head><body>" ;
31
32 // output xhtml form
33 cout << "<p>Enter one of your favorite words here:</p>"
34 << "<form method = \"get\" action = \"getquery.cgi\\>"
35 << "<input type = \"text\" name = \"word\\\"/>"
36 << "<input type = \"submit\" value = \"Submit Word\\\"/></form>" ;
37
38 if (query == "") // query is empty
39 cout << "<p>Please enter a word.</p>" ;
40 else // user entered query string
41 {
42 int wordLocation = query.find_first_of("word=") + 5;
43 wordString = query.substr(wordLocation);
44
45 if (wordString == "") // no word was entered
46 cout << "<p>Please enter a word.</p>" ;
47 else // word was entered
48 cout << "<p>Your word is: " << wordString << "</p>" ;

```

```

49 } // end else
50
51 cout << "</body></html>" ;
52 return 0;
53 } // end main

```





The form contains two input fields. The first (line 35) is a single-line text field (`type = "text"`) named `word`. The second (line 36) displays a button, labeled `Submit Word` (`value = "Submit Word"`), to submit the form data.

The first time the script is executed, there should be no value in `QUERY_STRING` (unless the user has appended the query string to the URL). [Note: Recall that on some servers `QUERY_STRING` may not even exist when the query string is empty.] Once the user enters a word into the word text field and clicks Submit Word, the script is requested again. This time, the name of the input field (word) and the value entered by the user are placed in the `QUERY_STRING` environment variable. That is, if the user enters the word "technology" and clicks Submit Word, `QUERY_STRING` is assigned the value `word=technology`. Note that the query string is also appended to the URL in the browser's Address field with a question mark (?) in front of it.

---

[Page 934]

During the second execution of the script, the query string is decoded. Lines 42 uses `string` method `find_first_of` to search `query` for the first occurrence of `word=`, which returns an integer value corresponding to the location in the `string` where the first match was found. Line 42 then adds 5 to the value returned by `find_first_of` to set `wordLocation` equal to the position in the `string` containing the first character of the user's favorite word. Function `substr` (line 43) returns the remainder of the `string` starting at `wordLocation`. Line 43 then assigns this `string` to `wordString`. Line 45 determines whether the user entered a word. If so, line 48 outputs the word entered by the user.

## Using post Request

The two preceding examples used `get` to pass data to the CGI scripts through an environment variable (i.e., `QUERY_STRING`). Web browsers typically interact with Web servers by submitting forms using HTTP `post`. CGI programs read the contents of `post` requests using standard input. For comparison purposes, let us now reimplement the application of Fig. 19.11, using the `post` method (as in Fig. 19.12). Notice that the code in the two figures is virtually identical. The XHTML form (lines 4345) indicates that we are now using the `post` method to submit the form data.

**Figure 19.12. Using post method with an XHTML form.**

(This item is displayed on pages 935 - 937 in the print version)

```

1 // Fig. 19.12: post.cpp
2 // Demonstrates POST method with XHTML form.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6
7 #include <string>
8 using std::string;
9
10 #include <cstdlib>
11 using std::getenv;
12 using std::atoi;
13
14 int main()
15 {
16 char postString[1024] = " " ; // variable to hold POST data
17 string dataString = " " ;
18 string nameString = " " ;
19 string wordString = " " ;
20 int contentLength = 0 ;
21
22 // content was submitted
23 if (getenv("CONTENT_LENGTH"))
24 {
25 contentLength = atoi(getenv("CONTENT_LENGTH"));
26 cin.read(postString, contentLength);
27 dataString = postString;
28 } // end if
29
30 cout << "Content-Type: text/html\n\n" ; // output header
31
32 // output XML declaration and DOCTYPE
33 cout << "<?xml version = \"1.0\"?>"
34 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
35 << "\\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\\>" ;
36
37 // output XHTML element and some of its contents
38 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
39 << "<head><title>Using POST with Forms</title></head><body>" ;
40
41 // output XHTML form
42 cout << "<p>Enter one of your favorite words here:</p>"
43 << "<form method = \"post\" action = \"post.cgi\">"
44 << "<input type = \"text\" name = \"word\" />"
45 << "<input type = \"submit\" value = \"Submit Word\" /></form>" ;
46
47 // data was sent using POST
48 if (contentLength > 0)

```

```

49 {
50 int nameLocation = dataString.find_first_of("word=") + 5;
51 int endLocation = dataString.find_first_of("&") - 1;
52
53 // retrieve entered word
54 wordString = dataString.substr(
55 nameLocation, endLocation - nameLocation);
56
57 if (wordString == "") // no data was entered in text field
58 cout << "<p>Please enter a word.</p>";
59 else // output word
60 cout << "<p>Your word is: " << wordString << "</p>";
61 } // end if
62 else // no data was sent
63 cout << "<p>Please enter a word.</p>";
64
65 cout << "</body></html>";
66 return 0;
67 } // end main

```

[View full size image]



The screenshot shows a Microsoft Internet Explorer window with the address bar containing `http://localhost/cgi-bin/getquery.cgi?word=technology`. The main content area displays the text "Enter one of your favorite words here:" followed by an empty input field and a "Submit Word" button. Below this, the message "Your word is: technology" is displayed. The status bar at the bottom shows "Done" and "Local intranet".

The screenshot shows a Microsoft Internet Explorer window with the address bar containing `http://localhost/cgi-bin/getquery.cgi`. The main content area displays the text "Enter one of your favorite words here:" followed by an input field containing "high-tech" and a "Submit Word" button. A validation error message "Please enter a word." is displayed below the form. The status bar at the bottom shows "Done" and "Local intranet".

The screenshot shows a Microsoft Internet Explorer window with the address bar containing `http://localhost/cgi-bin/getquery.cgi?word=high-tech`. The main content area displays the text "Enter one of your favorite words here:" followed by an empty input field and a "Submit Word" button. The status bar at the bottom shows "Done" and "Local intranet".

The screenshot shows a web browser window. At the top, there is a toolbar with a 'Done' button and a 'Local intranet' icon. The main content area displays the text 'Your word is: high-tech' in a large, bold, black font. The browser interface includes standard window controls like minimize, maximize, and close buttons.

---

[Page 937]

The Web server sends post data to a CGI script via standard input. The data is encoded (i.e., formatted) just as in QUERY\_STRING (that is, with name-value pairs connected by equals signs and ampersands), but the QUERY\_STRING environment variable is not set. Instead, the post method sets the environment variable **CONTENT\_LENGTH**, to indicate the number of characters of data that were sent in a post request.

The CGI script uses the value of the CONTENT\_LENGTH environment variable to process the correct amount of data. Line 23 determines whether CONTENT\_LENGTH contains a value. If so, line 25 reads in the value and converts it to an integer by calling `<cstdlib>` function `atoi`. Line 26 calls function `cin.read` to read characters from standard input and stores the characters in array `postString`. Line 27 converts `postString`'s data to a string by assigning it to `dataString`.

In earlier chapters, we read data from standard input using an expression such as

```
cin >> data;
```

---

[Page 938]

The same approach might work in our CGI script as a replacement for the `cin.read` statement. Recall that `cin` reads data from standard input up to and including the first newline character, space or tab, whichever comes first. The CGI specification (freely available at [cgi-spec.golux.com/cgi-120-00a.html](http://cgi-spec.golux.com/cgi-120-00a.html)) does not require a newline to be appended after the last name-value pair. Although some browsers append a newline or EOF, they are not required to do so. If `cin` is used with a browser that sends only the name-value pairs (as per the CGI specification), `cin` must wait for a newline that will never arrive. In this case, the server eventually "times out" and the CGI script terminates. Therefore, `cin.read` is preferred over `cin`, because the programmer can specify exactly how much data to read.

The CGI scripts in this section, while useful for explaining how `get` and `post` operate, do not include many of the features described in the CGI specification. For example, if we enter the words `didn't` translate into the text field and click the submit button, the script informs us that our word is `didn%27t+translate`.

What has happened here? Web browsers "URL encode" the XHTML form data they send. This means that spaces are replaced with plus signs, and other symbols (e.g., apostrophes) are translated into their ASCII value in hexadecimal format and preceded with a percent sign. URL encoding is necessary because URLs cannot contain certain characters, such as spaces and apostrophes.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 939]

header indicates that, although the request was successful, the client should not display a new page in the browser window. This header might be useful if you want to allow users to submit forms without relocating to a new page.

We have now covered the fundamentals of the CGI specification. To review, CGI allows scripts to interact with servers in three basic ways:

1.

through the output of headers and content to the server via standard output

2.

by the server's setting of environment variables (including the URL-encoded `QUERY_STRING`) whose values are available within the script (via `getenv`)

3.

through `POSTed`, URL-encoded data that the server sends to the script's standard input.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 940]

**Figure 19.14** contains the CGI script. First, let us examine how the user's name and password are retrieved from standard input and stored in strings. The string library `find` function searches `dataString` (line 30) for an occurrence of `namebox=`. Function `find` returns a location in the string where `namebox=` was found. To retrieve the value associated with `namebox=` the value entered by the user we move the position in the string forward 8 characters. The program now contains an integer "pointing" to the starting location. Recall that a query string contains name-value pairs separated by equals signs and ampersands. To find the ending location for the data we wish to retrieve, we search for the `&` character (line 31). The length of the entered word is determined by the calculation `endNameLocation - nameLocation`. We use a similar approach to determine the start and end locations of the password (lines 3233). Lines 3638 assign the form-field values to variables `nameString` and `passwordString`. We use `nameString` in line 52 to output a personalized greeting to the user. The current weekly specials are displayed in lines 5356.

[Page 942]

If the member password is correct, lines 5960 output additional specials. If the password is incorrect, the client is informed that the password was invalid and no additional specials are displayed.

Note that we use a static Web page and a separate CGI script here. We could have incorporated the opening XHTML form and the processing of the data into a single CGI script, as we did in previous examples in this chapter. We ask the reader to do this in [Exercise 19.8](#).

[Page 943]

### Performance Tip 19.1



It is always much more efficient for the server to provide static content rather than execute a CGI script, because it takes time for the server to load the script from hard disk into memory and execute the script (whereas an XHTML file needs to be sent only to the client). It is a good practice to use a mix of static XHTML (for content that generally remains unchanged) and CGI scripting (for dynamic content). This practice allows the Web server to respond to clients more efficiently than if only CGI scripting were used.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 946]

**Figure 19.15** is an XHTML page that contains a form in which values are to be input. The form posts its information to `writecookie.cgi` (Fig. 19.16). This CGI script retrieves the data contained in the `CONTENT_LENGTH` environment variable. Line 24 of Fig. 19.16 declares and initializes `string expires` to store the expiration date of the cookie, which determines how long the cookie resides on the client's machine. This value can be a string, like the one in this example, or it can be a relative value. For instance, `"+30d"` sets the cookie to expire after 30 days. For the purposes of this chapter the expiration date is deliberately set to expire in the year 2010 to ensure that the program will run properly well into the future. You may set the expiration date of this example to any future date as needed. The browser deletes cookies when they expire.

### Figure 19.16. Writing a cookie.

(This item is displayed on pages 944 - 946 in the print version)

```

1 // Fig. 19.16: writecookie.cpp
2 // Program to write a cookie to a client's machine.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6
7 #include <string>
8 using std::string;
9
10 #include <cstdlib>
11 using std::getenv;
12 using std::atoi;
13
14 int main()
15 {
16 char query[1024] = "";
17 string dataString = "";
18 string nameString = "";
19 string ageString = "";
20 string colorString = "";
21 int contentLength = 0;
22
23 // expiration date of cookie
24 string expires = "Friday, 14-MAY-10 16:00:00 GMT";
25
26 // data was entered

```

```

27 if (getenv("CONTENT_LENGTH"))
28 {
29 contentLength = atoi(getenv("CONTENT_LENGTH"));
30 cin.read(query, contentLength); // read data from standard input
31 dataString = query;
32
33 // search string for data and store locations
34 int nameLocation = dataString.find("name=") + 5;
35 int endName = dataString.find("&");
36 int ageLocation = dataString.find("age=") + 4;
37 int endAge = dataString.find("&color");
38 int colorLocation = dataString.find("color=") + 6;
39 int endColor = dataString.find("&button");
40
41 // get value for user's name
42 nameString = dataString.substr(
43 nameLocation, endName - nameLocation);
44
45 if (ageLocation > 0) // get value for user's age
46 ageString = dataString.substr(
47 ageLocation, endAge - ageLocation);
48
49 if (colorLocation > 0) // get value for user's favorite color
50 colorString = dataString.substr(
51 colorLocation, endColor - colorLocation);
52
53 // set cookie
54 cout << "Set-Cookie: Name=" << nameString << "age:"
55 << ageString << "color:" << colorString
56 << ";" << expires << ";" << path=\n";
57 } // end if
58
59 cout << "Content-Type: text/html\n\n"; // output HTTP header
60
61 // output XML declaration and DOCTYPE
62 cout << "<?xml version = \"1.0\"?>"
63 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
64 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\"> ";
65
66 // output html element and some of its contents
67 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
68 << "<head><title>Cookie Saved</title></head><body>";
69
70 // output user's information
71 cout << "<p>A cookie has been set with the following"
72 << " data:</p><p>Name: " << nameString << "
</p>"
73 << "<p>Age: " << ageString << "
</p>"
74 << "<p>Color: " << colorString << "
</p>"
```

```

75 << "<p>Click "

76 << "here to read saved cookie data.</p></body></html>" ;

77 return 0;

78 } // end main

```



After obtaining the data from the form, the program creates a cookie (lines 5456). In this example, we create a cookie that stores a line of text containing the name-value pairs of the posted data, delimited by a colon character (:). The line must be output before the header is written to the client. The line of text begins with the **Set-Cookie:** header, indicating that the browser should store the incoming data in a cookie. We set three attributes for the cookie: a name-value pair containing the data to be stored, a name-value pair containing the expiration date and a name-value pair containing the URL of the server domain (e.g., [www.deitel.com](http://www.deitel.com)) for which the cookie is valid. For this example, path is not set to any value, making the cookie readable from any server in the domain of the server that originally wrote the cookie. Note that these name-value pairs are separated by semicolons (;). We use only colon characters within our cookie data so as not to conflict with the format of the Set-Cookie: header. When we enter the same data as displayed in Fig. 19.15, lines 5456 store the data "Name=Zoeage:24color:Red" to the cookie. Lines 5976 send a Web page indicating that the cookie has been written to the client.

## Portability Tip 19.1



Web browsers store the cookie information in a vendor-specific manner. For example, Microsoft's Internet Explorer stores cookies as text files in the Temporary Internet Files directory on the client's machine. Netscape stores its cookies in a single file named `cookies.txt`

Figure 19.17 reads the cookie written in Fig. 19.16 and displays the stored information. When a client sends a request to a server, the client Web browser locates any cookies previously written by that server. These cookies are sent by the browser back to the server as part of the request. On the server, the environment variable `HTTP_COOKIE` stores the client's cookies. Line 20 calls function `getenv` with the `HTTP_COOKIE` environment variable as the parameter and stores the returned value in `dataString`. The name-value pairs are decoded and stored in strings (lines 2334) according to the encoding scheme used in Fig. 19.16. Lines 3655 output the contents of the cookie as a Web page.

**Figure 19.17. Program to read cookies sent from the client's computer.**

(This item is displayed on pages 947 - 948 in the print version)

```

1 // Fig. 19.17: readcookie.cpp
2 // Program to read cookie data.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6
7 #include <string>
8 using std::string;
9
10 #include <cstdlib>
11 using std::getenv;
12
13 int main()
14 {
15 string dataString = "";
16 string nameString = "";
17 string ageString = "";
18 string colorString = "";
19
20 dataString = getenv("HTTP_COOKIE"); // get cookie data
21
22 // search through cookie data string
23 int nameLocation = dataString.find("Name=") + 5;
24 int endName = dataString.find("age:");
25 int ageLocation = dataString.find("age:") + 4;
26 int endAge = dataString.find("color:");

```

```
27 int colorLocation = dataString.find("color:") + 6;
28
29 // store cookie data in strings
30 nameString = dataString.substr(
31 nameLocation, endName - nameLocation);
32 ageString = dataString.substr(
33 ageLocation, endAge - ageLocation);
34 colorString = dataString.substr(colorLocation);
35
36 cout << "Content-Type: text/html\n\n"; // output HTTP header
37
38 // output XML declaration and DOCTYPE
39 cout << "<?xml version = \"1.0\"?>"
40 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
41 << "\\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\\>" ;
42
43 // output html element and some of its contents
44 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\\>"
45 << "<head><title>Read Cookies</title></head><body>" ;
46
47 if (dataString != "") // data was found
48 cout << "<h3>The following data is saved in a cookie on"
49 << " your computer</h3><p>Name: " << nameString << "
</p>"
50 << "<p>Age: " << ageString << "
</p>"
51 << "<p>Color: " << colorString << "
</p>" ;
52 else // no data was found
53 cout << "<p>No cookie data.</p>" ;
54
55 cout << "</body></html>" ;
56 return 0;
57 } // end main
```

A screenshot of Microsoft Internet Explorer window titled "Read Cookies - Microsoft Internet Explorer". The address bar shows "http://localhost/cgi-bin/readcookie.cgi". The main content area displays the text: "The following data is saved in a cookie on your computer". Below this, three pieces of data are listed: "Name: Zoe", "Age: 24", and "Color: Red". The status bar at the bottom shows "Done" and "Local intranet".

[Page 949]

## Software Engineering Observation 19.2



Cookies present a security risk. If unauthorized users gain access to a computer, they can examine the local disk and view files, which include cookies. For this reason, sensitive data, such as passwords, social security numbers and credit card numbers, should never be stored in cookies.

[◀ PREV](#)

page footer

[NEXT ▶](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 953]

There are a few important points to make about this program. First, we do not perform any validation on the data before writing it to disk. Normally, the script would check for bad data, incomplete data, etc. Second, our file is located in the `cgi-bin` directory, which is publicly accessible. Someone who knew the filename would find it relatively easy to access someone else's contact information.

This script is not robust enough for deployment on the Internet, but it does provide an example of the use of server-side files to store information. Once the files are stored on the server, users cannot change them unless they are allowed to do so by the server administrator. Thus, storing these files on the server is safer than storing user data in cookies. [Note: Many systems store user information in password-protected databases for higher levels of security.]

---

[Page 954]

Note that, in this example, we show how to write data to a server-side file. In the next section we show how to retrieve data from a server-side file, using the techniques used for reading from a file in [Chapter 17](#), File Processing.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 959]

[Page 960]

If the user submitted data, program control continues into the else block that begins in line 79, where the script processes the data. Line 86 opens `userdata.txt`the file that contains all usernames and passwords for existing members. If the user checked the New checkbox on the Web page to create a new membership, the condition in line 95 evaluates to `TRue`, and the script attempts to record the user's information in the `userdata.txt` file on the server. Lines 98102 read through this file, comparing each username with the name entered. If the username already appears in the file, the loop in lines 98102 terminates before reaching the end of the file, and lines 107108 output an appropriate message to the user, and a hyperlink back to the form is provided. If the username entered does not already exist in `userdata.txt`, line 117 adds the new user information to the file in the format

```
Bernard
blue
```

Each username and password is separated by a newline character. Lines 119120 provide a hyperlink to the script of [Fig. 19.22](#), which allows users to purchase items.

**Figure 19.22. CGI script that allows users to buy a book.**

(This item is displayed on pages 961 - 963 in the print version)

```
1 // Fig. 19.22: shop.cpp
2 // Program to display available books.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::ios;
7
8 #include <fstream>
9 using std::ifstream;
10
11 #include <string>
12 using std::string;
13
14 #include <cstdlib>
15 using std::exit;
16
```

```
17 void header();
18
19 int main()
20 {
21 // variables to store product information
22 char book[50] = "";
23 char year[50] = "";
24 char isbn[50] = "";
25 char price[50] = "";
26
27 string bookString = "";
28 string yearString = "";
29 string isbnString = "";
30 string priceString = "";
31
32 ifstream userData("catalog.txt", ios::in); // open file for input
33
34 // file could not be opened
35 if (!userData)
36 {
37 cerr << "Could not open database." ;
38 exit(1);
39 } // end if
40
41 header(); // output header
42
43 // output available books
44 cout << "<center>
Books available for sale

" ;
45 << "<table border = \"1\" cellpadding = \"7\" >" ;
46
47 // file is open
48 while (userData)
49 {
50 // retrieve data from file
51 userData.getline(book, 50);
52 bookString = book;
53
54 userData.getline(year, 50);
55 yearString = year;
56
57 userData.getline(isbn, 50);
58 isbnString = isbn;
59
60 userData.getline(price, 50);
61 priceString = price;
62
63 cout << "<tr><td>" << bookString << "</td><td>" << yearString
64 << "</td><td>" << isbnString << "</td><td>" << priceString
```

```
65 << "</td>" ;
66
67 // file is still open after reads
68 if (userData)
69 {
70 // output form with buy button
71 cout << "<td><form method=\"post\" "
72 << "action=\"/cgi-bin/viewcart.cgi\">"
73 << "<input type=\"hidden\" name=\"add\" value=\"true\"/>"
74 << "<input type=\"hidden\" name=\"isbn\" value=\"\""
75 << isbnString << "\"/>" << "<input type=\"submit\" "
76 << "value=\"Add to Cart\"/>\n</form></td>\n";
77 } // end if
78
79 cout << "</tr>\n";
80 } // end while
81
82 cout << "</table></center>
"
83 << "check Out"
84 << "</body></html>";
85 return 0;
86 } // end main
87
88 // function to output header information
89 void header()
90 {
91 cout << "Content-Type: text/html\n\n"; // output header
92
93 // output XML declaration and DOCTYPE
94 cout << "<?xml version = \"1.0\"?>"
95 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
96 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
97
98 // output html element and some of its contents
99 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
100 << "<head><title>Shop Page</title></head><body>" ;
101 } // end function header
```

[View full size image]

The screenshot shows a Microsoft Internet Explorer window titled "Shop Page - Microsoft Internet Explorer". The address bar contains "http://localhost/cgi-bin/shop.cgi". The main content area displays a table titled "Books available for sale" with four rows of book information:

Book Title	Publish Year	ISBN	Price	Add to Cart
Visual Basic .NET How to Program	2002	0-13-029363-6	\$50.00	<input type="button" value="Add to Cart"/>
C# How to Program	2002	0-13-062221-4	\$49.95	<input type="button" value="Add to Cart"/>
C How to Program 4e	2004	0-13-142644-3	\$88.00	<input type="button" value="Add to Cart"/>
Java How to Program 6e	2005	0-13-148398-6	\$88.00	<input type="button" value="Add to Cart"/>

Below the table, there is a link labeled "Check Out". The status bar at the bottom right of the browser window shows "Local intranet".

The last possible scenario for this script is for returning users (lines 123162). This portion of the program executes when the user enters a name and password but does not select the New checkbox (i.e., the else of line 123 is evaluated). In this case, we assume that the user already has a username and password in `userdata.txt`. Lines 128139 read through `userdata.txt` in an attempt to locate the username entered. If the username is found (line 131), we determine whether the password entered matches the password stored in the file (line 137). If so, bool variable `authenticated` is set to `true`. Otherwise, `authenticated` remains `false`. If the user has been authenticated (line 142), line 144 calls function `writeCookie` to initialize a cookie named `CART` (line 188), which is used by other scripts to store data indicating which books the user has added to the shopping cart. Note that this cookie replaces any existing cookie of the same name, causing data from prior sessions to be deleted. After creating the cookie, the script outputs a message welcoming the user back to the Web site and providing a link to `shop.cgi`, where the user can purchase books (lines 147148).

If the user was not authenticated, the program determines why (lines 154160). If the user was found but not authenticated, a message is output indicating that the password is invalid (line 155157). A hyperlink is provided to the login page (`<a href=".../cgi-bin/login.cgi">`), where the user can attempt to login again. If neither the username nor the password was found, an unregistered user has attempted to login. Lines 159160 output a message indicating that the user does not have the proper authorization to

access the page and providing a link that allows the user to attempt another login.

**Figure 19.22** uses the values in `catalog.txt` ([Fig. 19.25](#)) to output in an XHTML table the items that the user can purchase (lines 4582). The last column for each row includes a button for adding the item to the shopping cart. Lines 6365 output the different values for each book, and lines 7176 output a form containing the `submit` button for adding each book to the shopping cart. Hidden form fields are specified for each book and its associated information. Note that the resulting XHTML document sent to the client contains several forms, one for each book. However, the user can submit only one form at a time. The name-value pairs of the hidden fields within the submitted form are posted to the `viewcart.cgi` script.

[Page 963]

When a user purchases a book, the `viewcart.cgi` script is requested, and the ISBN for the book to be purchased is sent to the script via a hidden form field. **Figure 19.23** begins by reading the value of the cookie stored on the user's system (line 35). Any existing cookie data is stored in `string cookieString` (line 36). The entered ISBN number from the form of [Fig. 19.22](#) is stored in `string isbnEntered` (line 52). The script then determines whether the cart already contains data (line 61). If not, the `cookieString` is given the value of the entered ISBN number (line 62). If the cookie already contains data, the entered ISBN is appended to the existing cookie data (line 64). The new book is stored in the `CART` cookie in lines 6768. Line 84 outputs the cart's contents in a table by calling function `displayShoppingCart`.

### Figure 19.23. CGI script that allows users to view their carts' contents.

(This item is displayed on pages 964 - 967 in the print version)

```

1 // Fig. 19.23: viewcart.cpp
2 // Program to view books in the shopping cart.
3 #include <iostream>
4 using std::cerr;
5 using std::cin;
6 using std::cout;
7 using std::ios;
8
9 #include <fstream>
10 using std::ifstream;
11
12 #include <string>
13 using std::string;
14
15 #include <cstdlib>
16 using std::getenv;
17 using std::atoi;
18 using std::exit;

```

```
19
20 void displayShoppingCart(const string &);
21
22 int main()
23 {
24 char query[1024] = ""; // variable to store query string
25 string cartData; // variable to hold contents of cart
26
27 string dataString = "";
28 string cookieString = "";
29 string isbnEntered = "";
30 int contentLength = 0;
31
32 // retrieve cookie data
33 if (getenv("HTTP_COOKIE"))
34 {
35 cartData = getenv("HTTP_COOKIE");
36 cookieString = cartData;
37 } // end if
38
39 // data was entered
40 if (getenv("CONTENT_LENGTH"))
41 {
42 contentLength = atoi(getenv("CONTENT_LENGTH"));
43 cin.read(query, contentLength);
44 dataString = query;
45
46 // find location of isbn value
47 int addLocation = dataString.find("add=") + 4;
48 int endAdd = dataString.find("&isbn");
49 int isbnLocation = dataString.find("isbn=") + 5;
50
51 // retrieve isbn number to add to cart
52 isbnEntered = dataString.substr(isbnLocation);
53
54 // write cookie
55 string expires = "Friday, 14-MAY-10 16:00:00 GMT";
56 int cartLocation = cookieString.find("CART=") + 5;
57
58 if (cartLocation > 4) // cookie exists
59 cookieString = cookieString.substr(cartLocation);
60
61 if (cookieString == "") // no cookie data exists
62 cookieString = isbnEntered;
63 else // cookie data exists
64 cookieString += "," + isbnEntered;
65
66 // set cookie
```

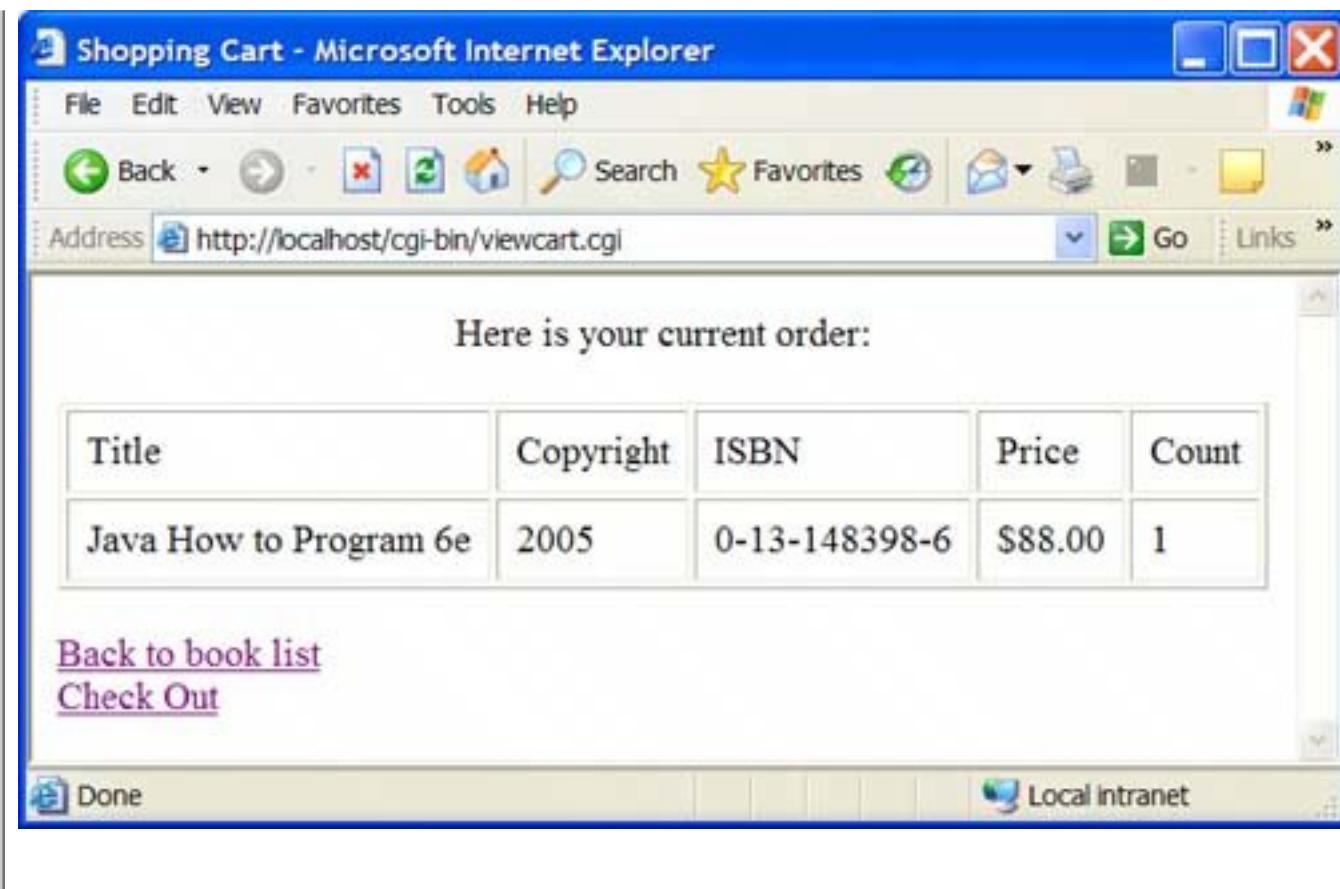
```

67 cout << "Set-Cookie: CART=" << cookieString << "; expires="
68 << expires << "; path=\n";
69 } // end if
70
71 cout << "Content-Type: text/html\n\n"; // output HTTP header
72
73 // output XML declaration and DOCTYPE
74 cout << "<?xml version = \"1.0\"?>"
75 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
76 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
77
78 // output html element and some of its contents
79 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
80 << "<head><title>Shopping Cart</title></head>"
81 << "<body><center><p>Here is your current order:</p>";
82
83 if (cookieString != "") // cookie data exists
84 displayShoppingCart(cookieString);
85 else
86 cout << "The shopping cart is empty.";
87
88 // output links back to book list and to check out
89 cout << "</center>
";
90 cout << "back to book list
";
91 cout << "check Out";
92 cout << "</body></html>\n";
93 return 0;
94 } // end main
95
96 // function to display items in shopping cart
97 void displayShoppingCart(const string &cookieRef)
98 {
99 char book[50] = "";
100 char year[50] = "";
101 char isbn[50] = "";
102 char price[50] = "";
103
104 string bookString = "";
105 string yearString = "";
106 string isbnString = "";
107 string priceString = "";
108
109 ifstream userData("catalog.txt", ios::in); // open file for input
110
111 if (!userData) // file could not be opened
112 {
113 cerr << "Could not open database.";
114 exit(1);

```

```
115 } // end if
116
117 cout << "<table border = 1 cellpadding = 7 >" ;
118 cout << "<tr><td>Title</td><td>Copyright</td><td>ISBN</td>"
119 << "<td>Price</td><td>Count</td></tr>" ;
120
121 // file is open
122 while (!userData.eof())
123 {
124 // retrieve book information
125 userData.getline(book, 50);
126 bookString = book;
127
128 // retrieve year information
129 userData.getline(year, 50);
130 yearString = year;
131
132 // retrieve isbn number
133 userData.getline(isbn, 50);
134 isbnString = isbn;
135
136 // retrieve price
137 userData.getline(price, 50);
138 priceString = price;
139
140 int match = cookieRef.find(isbnString, 0);
141 int count = 0;
142
143 // match has been made
144 while (match >= 0 && isbnString != "")
145 {
146 count++;
147 match = cookieRef.find(isbnString, match + 13);
148 } // end while
149
150 // output table row with book information
151 if (count != 0)
152 cout << "<tr><td>" << bookString << "</td><td>" << yearString
153 << "</td><td>" << isbnString << "</td><td>" << priceString
154 << "</td><td>" << count << "</td></tr>" ;
155 } // end while
156
157 cout << "</table>"; // end table
158 } // end function displayShoppingCart
```

[View full size image]



Function `displayShoppingCart` displays the items in the shopping cart in a table. Line 109 opens the server-side file `catalog.txt`. If the file opens successfully, lines 122155 get each book's information (including its title, copyright, ISBN and price) from the file. Lines 125138 store these pieces of data in `string` objects. Lines 140148 count how many times the current ISBN appears in the cookie (i.e., in the shopping cart). If the current book appears in the user's cart, lines 151154 display a table row containing the book's title, copyright, ISBN and price, as well as the number of copies of the book the user has chosen to purchase.

---

[Page 967]

Figure 19.24 is the page that is displayed when the user chooses to check out (i.e., purchase the books in the shopping cart). This script outputs a message to the user and calls `writeCookie` (line 13), which effectively erases the current information in the shopping cart.

#### **Figure 19.24. Check out program.**

(This item is displayed on pages 967 - 968 in the print version)

```
1 // Fig. 19.24: checkout.cpp
2 // Program to log out of the system.
3 #include <iostream>
4 using std::cout;
5
6 #include <string>
7 using std::string;
8
9 void writeCookie();
10
11 int main()
12 {
13 writeCookie(); // write the cookie
14 cout << "Content-Type: text/html\n\n"; // output header
15
16 // output XML declaration and DOCTYPE
17 cout << "<?xml version = \"1.0\"?>"
18 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
19 << "\\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\\> ";
20
21 // output html element and its contents
22 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\\>"
23 << "<head><title>Checked Out</title></head><body><center>"
24 << "<p>You have checked out
"
25 << "You will be billed accordingly
To login again, "
26 << "click here"
27 << "</center></body></html>\n";
28 return 0;
29 } // end main
30
31 // function to write cookie
32 void writeCookie()
33 {
34 // string containing expiration date
35 string expires = "Friday, 14-MAY-10 16:00:00 GMT";
36
37 // set cookie
38 cout << "Set-Cookie: CART=; expires=" << expires << "; path=\n";
39 } // end writeCookie
```

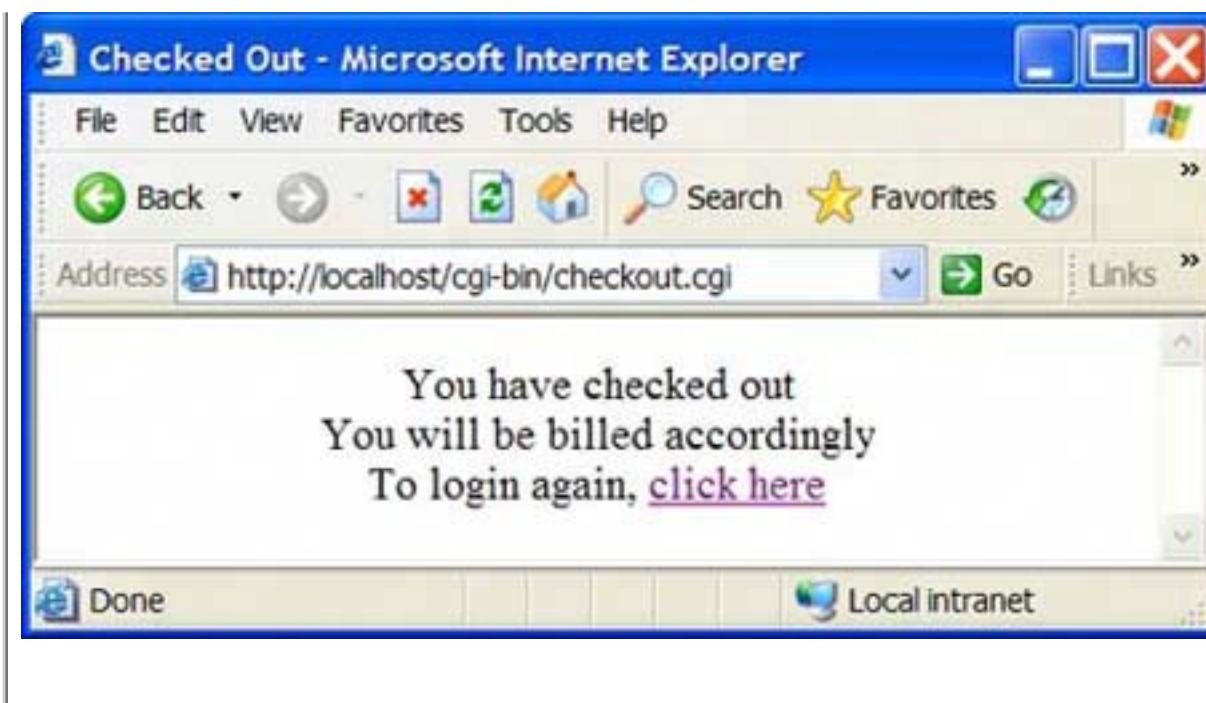


Figure 19.25 shows the contents of the catalog.txt file. This file must reside in the same directory as the CGI scripts for this shopping-cart application to work correctly.

[Page 968]

**Figure 19.25. Contents of catalog.txt.**

```
Visual Basic .NET How to Program
2002
0-13-029363-6
$50.00
C# How to Program
2002
0-13-062221-4
$49.95
C How to Program 4e
2004
0-13-142644-3
$88.00
Java How to Program 6e
2005
0-13-148398-6
$88.00
```

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 969]

## 19.17. Wrap-Up

In this chapter, we discussed the Hypertext Transfer Protocol (HTTP) and several HTTP request types that specify how a client makes requests from a server. You learned that a three-tier application contains an information tier (also called the data tier or the bottom tier), a middle tier (also called the business logic), and a client tier (also called the top tier). We also discussed how a Web browser is able to locate documents on a Web server in response to client requests. You then learned how the Common Gateway Interface (CGI) enables applications to interact with Web servers and clients (e.g., Web browsers). We demonstrated how to write CGI scripts in C++ and presented CGI scripts that display the local time, display system environment variables, read data passed through the query string, and read data passed through an XHTML form. We also presented a case study that provides an interactive portal for the fictional Bug2Bug Travel Web site. This case study contains both static XHTML documents and CGI scripts that generate dynamic Web content. You learned how to use CGI scripts to read and write cookies to maintain state information. We then discussed using server-side files as another way to maintain state information. Finally, we presented a case study that controls user access using server-side files and tracks user purchases with a cookie-based shopping cart.

In the next chapter, we discuss the binary search algorithm and the merge sort algorithm. We also use Big O notation to analyze and compare the efficiency of various searching and sorting algorithms.

[◀ PREV](#)[NEXT ▶](#)**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 969 (continued)]

## 19.18. Internet and Web Resources

### Apache

[httpd.apache.org](http://httpd.apache.org)

This is the product home page for the Apache HTTP server. Users may download Apache from this site.

[www.apacheweek.com](http://www.apacheweek.com)

This online magazine contains articles about Apache jobs, product reviews and other information concerning Apache software.

[linuxtoday.com/stories/18780.html](http://linuxtoday.com/stories/18780.html)

This site contains an article about the Apache HTTP server and the platforms that support it. It also contains links to other Apache articles.

### CGI

[www.gnu.org/software/cgicc/cgicc.html](http://www.gnu.org/software/cgicc/cgicc.html)

This site contains a free open-source CGI library for creating CGI scripts in C++.

[www.hotscripts.com](http://www.hotscripts.com)

This site contains a rich collection of scripts for performing image manipulation, server administration, networking, etc. using CGI.

[www.jmarshall.com/easy/cgi](http://www.jmarshall.com/easy/cgi)

This page contains a brief explanation of CGI for those with programming experience.

[www.w3.org/CGI](http://www.w3.org/CGI)

This World Wide Web Consortium page discusses CGI security issues.

[www.w3.org/Protocols](http://www.w3.org/Protocols)

This World Wide Web Consortium site contains information on the HTTP specification and links to news, mailing lists and published articles.

 PREV

NEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 971]

- Each FQDN corresponds to a numeric address called an IP (Internet Protocol) address, which is much like the street address of a house.
- A Domain Name System (DNS) server is a computer that maintains a database of FQDNs and their corresponding IP addresses. The process of translating FQDNs to IP addresses is called a DNS lookup.
- In the Apache HTTP server directory structure, XHTML documents must be saved in the `htdocs` directory.
- The Common Gateway Interface (CGI) is a standard protocol for enabling applications (commonly called CGI programs or CGI scripts) to interact with Web servers and (indirectly) with clients (e.g., Web browsers).
- CGI is often used to generate dynamic Web content using client input, databases and other information services.
- A Web page is dynamic if its content is generated programmatically when the page is requested, unlike static Web content, which is not generated programmatically when the page is requested (i.e., the page already exists before the request is made).
- HTTP describes a set of methods and headers that allows clients and servers to interact and exchange information in a uniform and predictable way.
- An XHTML document is a plain text file that contains markings (markup or elements) that describe the structure of the data the document contains.
- XHTML documents also can contain hypertext information (usually called hyperlinks), which are links to other Web pages or to other locations on the same page.
- A URL contains the protocol of the resource (such as `http`), the machine name or IP address for the resource and the name (including the path) of the resource.
- The HTTP get method indicates the client wishes to retrieve a resource.
- The information in the Content-Type header identifies the MIME (Multipurpose Internet Mail Extensions) type of the content.
- Each type of data sent from the server has a MIME type by which the browser determines how to process the data it receives.
- It is possible to redirect (or pipe) standard output to another destination.
- Function `time` gets the current time, which is represented as the number of seconds elapsed since midnight January 1, 1970, and stores the retrieved value to the location specified by the parameter.
- C++ library function `localtime`, when passed a `time_t` variable, returns a pointer to an object containing the "broken-down" local time (i.e., days, hours, etc. are placed in individual object members).
- Function `asctime`, which takes a pointer to an object containing "broken-down" time, returns a string such as `Wed Oct 31 13:10:37 2004`.
- Environment variables contain information about the client and server environment, such as the type of Web browser being used and the location of the document on the server.
- The value associated with an environment variable can be obtained by calling function `getenv` of `<cstdlib>` and passing it the name of the environment variable.
- The `QUERY_STRING` environment variable contains information that is appended to a URL in a get request.
- The `form` element encloses an XHTML form and generally takes two attributes. The first is `action`, which specifies the server resource to execute when the user submits the form. The second is `method`,

which identifies the type of HTTP request (i.e., get or post) to use when the browser submits the form to the Web server.

[Page 972]

- The `post` method sets the environment variable `CONTENT_LENGTH`, to indicate the number of characters of data that were sent in a post request.
- Function `atoi` of `<cstdlib>` can be used to convert the value contained in `CONTENT_LENGTH` to an integer.
- The `Refresh` header redirects the client to a new location after a specified amount of time.
- The `Location` header redirects the client to a new location.
- The `Status` header instructs the server to output a specified status header line (such as `HTTP/1.1 200 OK`).
- Cookies are essentially small text files that a Web server sends to your browser, which then saves them on your computer. Cookies are sent by the browser back to the server with each subsequent request to the same path until the cookies expire or are deleted.
- The `Set-Cookie:` header indicates that the browser should store the incoming data in a cookie.
- The environment variable `HTTP_COOKIE` stores the client's cookies.
- State information can be maintained using cookies or by creating server-side files (i.e., files that are located on the server or on the server's network).
- Only someone with access and permission to change files on the server can do so.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 973]

refresh header

relational database management system (RDBMS)

remote Web server

request method

server-side file

server-side form handler

Set-Cookie: HTTP header

start tag

static Web content

Status header

tier

time function

time\_t

title XHTML element (<title>...</title>)

top-level domain (TLD)

top tier

Universal Resource Locator (URL)

[virtual directory](#)

[Web address](#)

[Web server](#)

 PREV

NEXT 

**page footer**

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 973 (continued)]

## Self-Review Exercises

**19.1** Fill in the blanks in each of the following statements:

a.

The two most common HTTP request types are \_\_\_\_\_ and \_\_\_\_\_.

b.

Browsers often \_\_\_\_\_ Web pages for quick reloading.

c.

In a three-tier application, a Web server is typically part of the \_\_\_\_\_ tier.

d.

In the URL <http://www.deitel.com/books/downloads.html>, [www.deitel.com](http://www.deitel.com) is the \_\_\_\_\_ of the server, where a client can find the desired resource.

e.

A(n) \_\_\_\_\_ document is a text file containing markings that describe to a Web browser how to display and format the information in the document.

f.

The environment variable \_\_\_\_\_ provides a mechanism for supplying data to CGI scripts.

g.

A common way of reading input from the user is to implement the XHTML

\_\_\_\_\_ element.

**19.2** State whether each of the following is true or false. If false, explain why.

a.

Web servers and clients communicate with each other through the platform-independent HTTP.

b.

Web servers often cache Web pages for reloading.

c.

The information tier implements business logic to control the type of information that is presented to a particular client.

d.

A dynamic Web page is a Web page that is not created programmatically.

e.

We put data into a query string using a format that consists of a series of name-value pairs joined with exclamation points (!).

f.

Using a CGI script is more efficient than using an XHTML document.

g.

The post method of submitting form data is preferable to get when sending personal information to the Web server.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 973 (continued)]

## Answers to Self-Review Exercises

- 19.1** a) get and post. b) cache. c) middle. d) machine name. e) XHTML. f) QUERY\_STRING. g) form.
- 19.2** a) True. b) True. c) False. The middle tier implements business logic to control interactions between application clients and application data. d) False. A dynamic Web page is a Web page that is created programmatically. e) False. The pairs are joined with an ampersand (&). f) False. XHTML documents are more efficient than CGI scripts, because they do not need to be executed on the server side before they are output to the client. g) True.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 974]

## Exercises

**19.3** Define the following terms:

a.

HTTP.

b.

Multitier application.

c.

Request method.

**19.4** Explain the difference between the get request type and the post request type. When is it ideal to use the post request type?

**19.5** Write a CGI script that prints the squares of the integers from 1 to 10 on separate lines.

**19.6** Write a CGI script that receives as input three numbers from the client and displays a statement indicating whether the three numbers could represent an equilateral triangle (all three sides are the same length), an isosceles triangle (two sides are the same length) or a right triangle (the square of one side is equal to the sum of the squares of the other two sides).

**19.7** Write a soothsayer script that allows the user to submit a question. When the question is submitted, the script should choose a random response from a list of answers (such as "It could be", "Probably not", "Definitely", "Not looking too good" and "Yes") and display the answer to the client.

- 19.8** Modify the program of Figs. 19.1319.14 to incorporate the opening XHTML form and the processing of the data into a single CGI script (i.e., combine the XHTML of Fig. 19.13 into the CGI script of Fig. 19.14). When the CGI script is requested initially, the form should be displayed. When the form is submitted, the CGI script should display the result.
- 19.9** Modify the `viewcart.cgi` script (Fig. 19.23) to enable users to remove some items from the shopping cart.

 PREVNEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 976]

## Outline

### [20.1 Introduction](#)

### [20.2 Searching Algorithms](#)

#### [20.2.1 Efficiency of Linear Search](#)

#### [20.2.2 Binary Search](#)

### [20.3 Sorting Algorithms](#)

#### [20.3.1 Efficiency of Selection Sort](#)

#### [20.3.2 Efficiency of Insertion Sort](#)

#### [20.3.3 Merge Sort \(A Recursive Implementation\)](#)

### [20.4 Wrap-Up](#)

## Summary

## Terminology

## Self-Review Exercises

## Answers to Self-Review Exercises

## Exercises

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)
[NEXT ▶](#)

[Page 976 (continued)]

## 20.1. Introduction

**Searching** data involves determining whether a value (referred to as the **search key**) is present in the data and, if so, finding the value's location. Two popular search algorithms are the simple linear search (introduced in [Section 7.7](#)) and the faster but more complex binary search, which is introduced in this chapter.

**Sorting** places data in order, typically ascending or descending, based on one or more **sort keys**. A list of names could be sorted alphabetically, bank accounts could be sorted by account number, employee payroll records could be sorted by social security number, and so on. Previously, you learned about insertion sort ([Section 7.8](#)) and selection sort ([Section 8.6](#)). This chapter introduces the more efficient, but more complex merge sort. [Figure 20.1](#) summarizes the searching and sorting algorithms discussed in the examples and exercises of this book. This chapter also introduces **Big O notation**, which is used to estimate the worst-case runtime for an algorithm that is, how hard an algorithm may have to work to solve a problem.

**Figure 20.1. Searching and sorting algorithms in this text.**

(This item is displayed on page 977 in the print version)

Chapter	Algorithm	Location
Searching Algorithms		
7	Linear search	<a href="#">Section 7.7</a>
20	Binary search	<a href="#">Section 20.2.2</a>
	Recursive linear search	<a href="#">Exercise 20.8</a>
	Recursive binary search	<a href="#">Exercise 20.9</a>
21	Binary tree search	<a href="#">Section 21.7</a>

	Linear search of a linked list	Exercise 21.21
23	binary_search standard library function	Section 23.5.6
<b>Sorting Algorithms</b>		
7	Insertion sort	Section 7.8
8	Selection sort	Section 8.6
20	Recursive merge sort	Section 20.3.3
	Bubble sort	Exercises 20.5 and 20.6
	Bucket sort	Exercise 20.7
	Recursive quicksort	Exercise 20.10
21	Binary tree sort	Section 21.7
23	sort standard library function	Section 23.5.6
	Heap sort	Section 23.5.12

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 977]

An algorithm that tests whether the first element of a vector is equal to any of the other elements of the vector requires at most  $n - 1$  comparisons, where  $n$  is the number of elements in the vector. If the vector has 10 elements, this algorithm requires up to nine comparisons. If the vector has 1000 elements, this algorithm requires up to 999 comparisons. As  $n$  grows larger, the  $n$  part of the expression "dominates," and subtracting one becomes inconsequential. Big O is designed to highlight these dominant terms and ignore terms that become unimportant as  $n$  grows. For this reason, an algorithm that requires a total of  $n - 1$  comparisons (such as the one we described in this paragraph) is said to be  **$O(n)$** . An  $O(n)$  algorithm is referred to as having a **linear runtime**.  $O(n)$  is often pronounced "on the order of  $n$ " or more simply "**order  $n$** ."

Now suppose you have an algorithm that tests whether any element of a vector is duplicated elsewhere in the vector. The first element must be compared with every other element in the vector. The second element must be compared with every other element except the first (it was already compared to the first). The third element must be compared with every other element except the first two. In the end, this algorithm will end up making  $(n - 1) + (n - 2) + \dots + 2 + 1$  or  $n^2/2$  comparisons. As  $n$  increases, the  $n^2$  term dominates and the  $n$  term becomes inconsequential. Again, Big O notation highlights the  $n^2$  term, leaving  $n^2/2$ . As we will soon see, however, constant factors are omitted in Big O notation.

[Page 978]

Big O is concerned with how an algorithm's runtime grows in relation to the number of items processed. Suppose an algorithm requires  $n^2$  comparisons. With four elements, the algorithm will require 16 comparisons; with eight elements, 64 comparisons. With this algorithm, doubling the number of elements quadruples the number of comparisons. Consider a similar algorithm requiring  $n^2/2$  comparisons. With four elements, the algorithm will require eight comparisons; with eight elements, 32 comparisons. Again, doubling the number of elements quadruples the number of comparisons. Both of these algorithms grow as the square of  $n$ , so Big O ignores the constant, and both algorithms are considered to be  **$O(n^2)$** , which is referred to as **quadratic runtime** and pronounced "on the order of  $n$ -squared" or more simply "**order  $n$ -squared**."

When  $n$  is small,  $O(n^2)$  algorithms (running on today's billion-operation-per-second personal computers) will not noticeably affect performance. But as  $n$  grows, you will start to notice the performance degradation. An  $O(n^2)$  algorithm running on a million-element vector would require a trillion "operations" (where each could actually require several machine instructions to execute). This could require a few hours to execute. A billion-element vector would require a quintillion operations, a number so large that the algorithm could take decades!  $O(n^2)$  algorithms are easy to write, as you have seen in previous chapters. In this chapter, you will see algorithms with more favorable Big O measures. These efficient algorithms often take a bit more cleverness and effort to create, but their superior performance can be well worth the extra effort, especially as  $n$  gets large and as algorithms are compounded into

larger programs.

The linear search algorithm runs in  $O(n)$  time. The worst case in this algorithm is that every element must be checked to determine whether the search key exists in the vector. If the size of the vector is doubled, the number of comparisons that the algorithm must perform is also doubled. Note that linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the vector. But we seek algorithms that perform well, on average, across all searches, including those where the element matching the search key is near the end of the vector.

Linear search is the easiest search algorithm to implement, but it can be slow compared to other search algorithms. If a program needs to perform many searches on large vectors, it may be better to implement a different, more efficient algorithm, such as the binary search which we present in the next section.

Performance Tip 20.1



Sometimes the simplest algorithms perform poorly. Their virtue is that they are easy to program, test and debug. Sometimes more complex algorithms are required to realize maximum performance.

## 20.2.2. Binary Search

The **binary search algorithm** is more efficient than the linear search algorithm, but it requires that the vector first be sorted. This is only worthwhile when the vector, once sorted, will be searched a great many times or when the searching application has stringent performance requirements. The first iteration of this algorithm tests the middle element in the vector. If this matches the search key, the algorithm ends. Assuming the vector is sorted in ascending order, then if the search key is less than the middle element, the search key cannot match any element in the second half of the vector and the algorithm continues with only the first half of the vector (i.e., the first element up to, but not including, the middle element). If the search key is greater than the middle element, the search key cannot match any element in the first half of the vector and the algorithm continues with only the second half of the vector (i.e., the element after the middle element through the last element). Each iteration tests the middle value of the remaining portion of the vector. If the element does not match the search key, the algorithm eliminates half of the remaining elements. The algorithm ends either by finding an element that matches the search key or by reducing the subvector to zero size.

---

[Page 979]

As an example, consider the sorted 15-element vector

2    3    5    10    27    30    34    51    56    65    77    81    82    93    99

and a search key of 65. A program implementing the binary search algorithm would first check whether

51 is the search key (because 51 is the middle element of the vector). The search key (65) is larger than 51, so 51 is discarded along with the first half of the vector (all elements smaller than 51.) Next, the algorithm checks whether 81 (the middle element of the remainder of the vector) matches the search key. The search key (65) is smaller than 81, so 81 is discarded along with the elements larger than 81. After just two tests, the algorithm has narrowed the number of elements to check to three (56, 65 and 77). The algorithm then checks 65 (which indeed matches the search key), and returns the index (9) of the vector element containing 65. In this case, the algorithm required just three comparisons to determine whether an element of the vector matched the search key. Using a linear search algorithm would have required 10 comparisons. [Note: In this example, we have chosen to use a vector with 15 elements, so that there will always be an obvious middle element in the vector. With an even number of elements, the middle of the vector lies between two elements. We implement the algorithm to choose the larger of those two elements.]

Figures 20.220.3 define class `BinarySearch` and its member functions respectively. Class `BinarySearch` is similar to `LinearSearch` (Section 7.7) it has a constructor, a search function (`binarySearch`), a `displayElements` function, two private data members and a private utility function (`displaySubElements`). Lines 1828 of Fig. 20.3 define the constructor. After initializing the vector with random ints from 1099 (lines 2425), line 27 calls the Standard Library function `sort` on the vector data. Function `sort` takes two **random-access iterators** and sorts the elements in vector data in ascending order. A random-access iterator is an iterator that allows access to any data item in the vector at any time. In this case, we use `data.begin()` and `data.end()` to encompass the entire vector. Recall that the binary search algorithm will work only on a sorted vector.

## Figure 20.2. BinarySearch class definition.

(This item is displayed on page 980 in the print version)

```

1 // Fig 20.2: BinarySearch.h
2 // Class that contains a vector of random integers and a function
3 // that uses binary search to find an integer.
4 #include <vector>
5 using std::vector;
6
7 class BinarySearch
8 {
9 public:
10 BinarySearch(int); // constructor initializes vector
11 int binarySearch(int) const; // perform a binary search on vector
12 void displayElements() const; // display vector elements
13 private:
14 int size; // vector size
15 vector< int > data; // vector of ints
16 void displaySubElements(int, int) const; // display range of values
17 };// end class BinarySearch

```

**Figure 20.3. BinarySearch class member-function definition.**

(This item is displayed on pages 980 - 981 in the print version)

```

1 // Fig 20.3: BinarySearch.cpp
2 // BinarySearch class member-function definition.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // prototypes for functions srand and rand
8 using std::rand;
9 using std::srand;
10
11 #include <ctime> // prototype for function time
12 using std::time;
13
14 #include <algorithm> // prototype for sort function
15 #include "BinarySearch.h" // class BinarySearch definition
16
17 // constructor initializes vector with random ints and sorts the vector
18 BinarySearch::BinarySearch(int vectorSize)
19 {
20 size = (vectorSize > 0 ? vectorSize : 10); // validate vectorSize
21 srand(time(0)); // seed using current time
22
23 // fill vector with random ints in range 10-99
24 for (int i = 0; i < size; i++)
25 data.push_back(10 + rand() % 90); // 10-99
26
27 std::sort(data.begin(), data.end()); // sort the data
28 } // end BinarySearch constructor
29
30 // perform a binary search on the data
31 int BinarySearch::binarySearch(int searchElement) const
32 {
33 int low = 0; // low end of the search area
34 int high = size - 1; // high end of the search area
35 int middle = (low + high + 1) / 2; // middle element
36 int location = -1; // return value; -1 if not found
37
38 do // loop to search for element
39 {
40 // print remaining elements of vector to be searched
41 displaySubElements(low, high);
42

```

```

43 // output spaces for alignment
44 for (int i = 0; i < middle; i++)
45 cout << " ";
46
47 cout << " * " << endl; // indicate current middle
48
49 // if the element is found at the middle
50 if (searchElement == data[middle])
51 location = middle; // location is the current middle
52 else if (searchElement < data[middle]) // middle is too high
53 high = middle - 1; // eliminate the higher half
54 else // middle element is too low
55 low = middle + 1; // eliminate the lower half
56
57 middle = (low + high + 1) / 2; // recalculate the middle
58 } while ((low <= high) && (location == -1));
59
60 return location; // return location of search key
61 } // end function binarySearch
62
63 // display values in vector
64 void BinarySearch::displayElements() const
65 {
66 displaySubElements(0, size - 1);
67 } // end function displayElements
68
69 // display certain values in vector
70 void BinarySearch::displaySubElements(int low, int high) const
71 {
72 for (int i = 0; i < low; i++) // output spaces for alignment
73 cout << " ";
74
75 for (int i = low; i <= high; i++) // output elements left in vector
76 cout << data[i] << " ";
77
78 cout << endl;
79 } // end function displaySubElements

```

Lines 3161 define function `binarySearch`. The search key is passed into parameter `searchElement` (line 31). Lines 3335 calculate the `low` end index, `high` end index and `middle` index of the portion of the vector that the program is currently searching. At the beginning of the function, the `low` end is 0, the `high` end is the size of the vector minus 1 and the `middle` is the average of these two values. Line 36 initializes the `location` of the found element to `-1`the value that will be returned if the search key is not found. Lines 3858 loop until `low` is greater than `high` (this occurs when the element is not found) or `location` does not equal `-1` (indicating that the search key was found). Line 50 tests whether the value

in the middle element is equal to searchElement. If this is TRUE, line 51 assigns middle to location. Then the loop terminates and location is returned to the caller. Each iteration of the loop tests a single value (line 50) and eliminates half of the remaining values in the vector (line 53 or 55).

[Page 981]

[Page 982]

Lines 2541 of Fig. 20.4 loop until the user enters the value -1. For each other number the user enters, the program performs a binary search on the data to determine whether it matches an element in the vector. The first line of output from this program is the vector of ints, in increasing order. When the user instructs the program to search for 38, the program first tests the middle element, which is 67 (as indicated by \*). The search key is less than 67, so the program eliminates the second half of the vector and tests the middle element from the first half of the vector. The search key equals 38, so the program returns the index 3.

**Figure 20.4. BinarySearch test program.**

(This item is displayed on pages 983 - 984 in the print version)

```

1 // Fig 20.4: Fig20_04.cpp
2 // BinarySearch test program.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include "BinarySearch.h" // class BinarySearch definition
9
10 int main()
11 {
12 int searchInt; // search key
13 int position; // location of search key in vector
14
15 // create vector and output it
16 BinarySearch searchVector (15);
17 searchVector.displayElements();
18
19 // get input from user
20 cout << "\nPlease enter an integer value (-1 to quit): ";
21 cin >> searchInt; // read an int from user
22 cout << endl;
23
24 // repeatedly input an integer; -1 terminates the program
25 while (searchInt != -1)

```

```

26 {
27 // use binary search to try to find integer
28 position = searchVector.binarySearch(searchInt);
29
30 // return value of -1 indicates integer was not found
31 if (position == -1)
32 cout << "The integer " << searchInt << " was not found.\n";
33 else
34 cout << "The integer " << searchInt
35 << " was found in position " << position << ".\n";
36
37 // get input from user
38 cout << "\n\nPlease enter an integer value (-1 to quit): ";
39 cin >> searchInt; // read an int from user
40 cout << endl;
41 } // end while
42
43 return 0;
44 } // end main

```

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95

Please enter an integer value (-1 to quit): 38

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95  
\*

26 31 33 38 47 49 49  
\*

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95

Please enter an integer value (-1 to quit): 38

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95  
\*

26 31 33 38 47 49 49  
\*

The integer 38 was found in position 3.

Please enter an integer value (-1 to quit): 91

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95  
\*

73 74 82 89 90 91 95

\*

90 91 95

\*

```
The integer 91 was found in position 13.
```

```
Please enter an integer value (-1 to quit): 25
```

```
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
*
```

```
26 31 33 38 47 49 49
*
```

```
26 31 33
*
```

```
26
*
```

```
The integer 25 was not found.
```

```
Please enter an integer value (-1 to quit): -1
```

## Efficiency of Binary Search

In the worst-case scenario, searching a sorted vector of 1023 elements will take only 10 comparisons when using a binary search. Repeatedly dividing 1023 by 2 (because, after each comparison, we are able to eliminate half of the vector) and rounding down (because we also remove the middle element) yields the values 511, 255, 127, 63, 31, 15, 7, 3, 1 and 0. The number 1023 ( $2^{10} - 1$ ) is divided by 2 only 10 times to get the value 0, which indicates that there are no more elements to test. Dividing by 2 is equivalent to one comparison in the binary search algorithm. Thus, a vector of 1,048,575 ( $2^{20} - 1$ ) elements takes a maximum of 20 comparisons to find the key, and a vector of over one billion elements takes a maximum of 30 comparisons to find the key. This is a tremendous improvement in performance over the linear search. For a one-billion-element vector, this is a difference between an average of 500 million comparisons for the linear search and a maximum of only 30 comparisons for the binary search! The maximum number of comparisons needed for the binary search of any sorted vector is the exponent of the first power of 2 greater than the number of elements in the vector, which is represented as  $\log_2 n$ .

All logarithms grow at roughly the same rate, so in Big O notation the base can be omitted. This results in a Big O of **O(log n)** for a binary search, which is also known as **logarithmic runtime** and pronounced "on the order of log n" or more simply "**order log n**."

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 984]

### 20.3.1. Efficiency of Selection Sort

Selection sort is an easy-to-implement, but inefficient, sorting algorithm. The first iteration of the algorithm selects the smallest element in the vector and swaps it with the first element. The second iteration selects the second-smallest element (which is the smallest element of the remaining elements) and swaps it with the second element. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last element, leaving the largest element in the last index. After the  $i$ th iteration, the smallest  $i$  elements of the vector will be sorted into increasing order in the first  $i$  elements of the vector.

The selection sort algorithm iterates  $n - 1$  times, each time swapping the smallest remaining element into its sorted position. Locating the smallest remaining element requires  $n - 1$  comparisons during the first iteration,  $n - 2$  during the second iteration, then  $n - 3, \dots, 3, 2, 1$ . This results in a total of  $n(n - 1)/2$  or  $(n^2 - n)/2$  comparisons. In Big O notation, smaller terms drop out and constants are ignored, leaving a final Big O of  $O(n^2)$ .

[Page 985]

### 20.3.2. Efficiency of Insertion Sort

Insertion sort is another simple, but inefficient, sorting algorithm. The first iteration of this algorithm takes the second element in the vector and, if it is less than the first element, swaps it with the first element. The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order. At the  $i$ th iteration of this algorithm, the first  $i$  elements in the original vector will be sorted.

Insertion sort iterates  $n - 1$  times, inserting an element into the appropriate position in the elements sorted so far. For each iteration, determining where to insert the element can require comparing the element to each of the preceding elements in the vector. In the worst case, this will require  $n - 1$  comparisons. Each individual repetition statement runs in  $O(n)$  time. For determining Big O notation, nested statements mean that you must multiply the number of comparisons. For each iteration of an outer loop, there will be a certain number of iterations of the inner loop. In this algorithm, for each  $O(n)$  iteration of the outer loop, there will be  $O(n)$  iterations of the inner loop, resulting in a Big O of  $O(n^* n)$  or  $O(n^2)$ .

### 20.3.3. Merge Sort (A Recursive Implementation)

**Merge sort** is an efficient sorting algorithm, but is conceptually more complex than selection sort and insertion sort. The merge sort algorithm sorts a vector by splitting it into two equal-sized subvectors,

sorting each subvector and then merging them into one larger vector. With an odd number of elements, the algorithm creates the two subvectors such that one has one more element than the other.

The implementation of merge sort in this example is recursive. The base case is a vector with one element. A one-element vector is, of course, sorted, so merge sort immediately returns when it is called with a one-element vector. The recursion step splits a vector of two or more elements into two equal-sized subvectors, recursively sorts each subvector, then merges them into one larger, sorted vector. [Again, if there is an odd number of elements, one subvector is one element larger than the other.]

Suppose the algorithm has already merged smaller vectors to create sorted vectors A:

4      10      34      56      77

and B:

5      30      51      52      93

Merge sort combines these two vectors into one larger, sorted vector. The smallest element in A is 4 (located in the zeroth index of A). The smallest element in B is 5 (located in the zeroth index of B). In order to determine the smallest element in the larger vector, the algorithm compares 4 and 5. The value from A is smaller, so 4 becomes the first element in the merged vector. The algorithm continues by comparing 10 (the second element in A) to 5 (the first element in B). The value from B is smaller, so 5 becomes the second element in the larger vector. The algorithm continues by comparing 10 to 30, with 10 becoming the third element in the vector, and so on.

[Figure 20.5](#) defines class `MergeSort` and lines 3134 of [Fig. 20.6](#) define the `sort` function. Line 33 calls function `sortSubVector` with 0 and `size - 1` as the arguments. The arguments correspond to the beginning and ending indices of the vector to be sorted, causing `sortSubVector` to operate on the entire vector. Function `sortSubVector` is defined in lines 3761. Line 40 tests the base case. If the size of the vector is 1, the vector is already sorted, so the function simply returns immediately. If the size of the vector is greater than 1, the function splits the vector in two, recursively calls function `sortSubVector` to sort the two subvectors, then merges them. Line 55 recursively calls function `sortSubVector` on the first half of the vector, and line 56 recursively calls function `sortSubVector` on the second half of the vector. When these two function calls return, each half of the vector has been sorted. Line 59 calls function `merge` (lines 64108) on the two halves of the vector to combine the two sorted vectors into one larger sorted vector.

[Page 988]

[Page 989]

**Figure 20.5. MergeSort class definition.**

(This item is displayed on page 986 in the print version)

```

1 // Fig 20.5: MergeSort.h
2 // Class that creates a vector filled with random integers.
3 // Provides a function to sort the vector with merge sort.
4 #include <vector>
5 using std::vector;
6
7 // MergeSort class definition
8 class MergeSort
9 {
10 public:
11 MergeSort(int); // constructor initializes vector
12 void sort(); // sort vector using merge sort
13 void displayElements() const; // display vector elements
14 private:
15 int size; // vector size
16 vector< int > data; // vector of ints
17 void sortSubVector(int, int); // sort subvector
18 void merge(int, int, int, int); // merge two sorted vectors
19 void displaySubVector(int, int) const; // display subvector
20 } // end class SelectionSort

```

**Figure 20.6. MergeSort class member-function definition.**

(This item is displayed on pages 986 - 988 in the print version)

```

1 // Fig 20.6: MergeSort.cpp
2 // Class MergeSort member-function definition.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <vector>
8 using std::vector;
9
10 #include <cstdlib> // prototypes for functions srand and rand
11 using std::rand;
12 using std::srand;
13
14 #include <ctime> // prototype for function time
15 using std::time;
16
17 #include "MergeSort.h" // class MergeSort definition

```

```

18
19 // constructor fill vector with random integers
20 MergeSort::MergeSort(int vectorSize)
21 {
22 size = (vectorSize > 0 ? vectorSize : 10); // validate vectorSize
23 srand(time(0)); // seed random number generator using current time
24
25 // fill vector with random ints in range 10-99
26 for (int i = 0; i < size; i++)
27 data.push_back(10 + rand() % 90);
28 } // end MergeSort constructor
29
30 // split vector, sort subvectors and merge subvectors into sorted vector
31 void MergeSort::sort()
32 {
33 sortSubVector(0, size - 1); // recursively sort entire vector
34 } // end function sort
35
36 // recursive function to sort subvectors
37 void MergeSort::sortSubVector(int low, int high)
38 {
39 // test base case; size of vector equals 1
40 if ((high - low) >= 1) // if not base case
41 {
42 int middle1 = (low + high) / 2; // calculate middle of vector
43 int middle2 = middle1 + 1; // calculate next element over
44
45 // output split step
46 cout << "split: ";
47 displaySubVector(low, high);
48 cout << endl << " ";
49 displaySubVector(low, middle1);
50 cout << endl << " ";
51 displaySubVector(middle2, high);
52 cout << endl << endl;
53
54 // split vector in half; sort each half (recursive calls)
55 sortSubVector(low, middle1); // first half of vector
56 sortSubVector(middle2, high); // second half of vector
57
58 // merge two sorted vectors after split calls return
59 merge(low, middle1, middle2, high);
60 } // end if
61 } // end function sortSubVector
62
63 // merge two sorted subvectors into one sorted subvector
64 void MergeSort::merge(int left, int middle1, int middle2, int right)
65 {
66 int leftIndex = left; // index into left subvector

```

```

67 int rightIndex = middle2; // index into right subvector
68 int combinedIndex = left; // index into temporary working vector
69 vector< int > combined(size); // working vector
70
71 // output two subvectors before merging
72 cout << "merge: ";
73 displaySubVector(left, middle1);
74 cout << endl << " ";
75 displaySubVector(middle2, right);
76 cout << endl;
77
78 // merge vectors until reaching end of either
79 while (leftIndex <= middle1 && rightIndex <= right)
80 {
81 // place smaller of two current elements into result
82 // and move to next space in vector
83 if (data[leftIndex] <= data[rightIndex])
84 combined[combinedIndex++] = data[leftIndex++];
85 else
86 combined[combinedIndex++] = data[rightIndex++];
87 } // end while
88
89 if (leftIndex == middle2) // if at end of left vector
90 {
91 while (rightIndex <= right) // copy in rest of right vector
92 combined[combinedIndex++] = data[rightIndex++];
93 } // end if
94 else // at end of right vector
95 {
96 while (leftIndex <= middle1) // copy in rest of left vector
97 combined[combinedIndex++] = data[leftIndex++];
98 } // end else
99
100 // copy values back into original vector
101 for (int i = left; i <= right; i++)
102 data[i] = combined[i];
103
104 // output merged vector
105 cout << " ";
106 displaySubVector(left, right);
107 cout << endl << endl;
108 } // end function merge
109
110 // display elements in vector
111 void MergeSort::displayElements() const
112 {
113 displaySubVector(0, size - 1);
114 } // end function displayElements
115

```

```

116 // display certain values in vector
117 void MergeSort::displaySubVector(int low, int high) const
118 {
119 // output spaces for alignment
120 for (int i = 0; i < low; i++)
121 cout << " ";
122
123 // output elements left in vector
124 for (int i = low; i <= high; i++)
125 cout << " " << data[i];
126 } // end function displaySubVector

```

Lines 7987 in function `merge` loop until the program reaches the end of either subvector. Line 83 tests which element at the beginning of the vectors is smaller. If the element in the left vector is smaller, line 84 places it in position in the combined vector. If the element in the right vector is smaller, line 86 places it in position in the combined vector. When the while loop has completed (line 87), one entire subvector is placed in the combined vector, but the other subvector still contains data. Line 89 tests whether the left vector has reached the end. If so, lines 9192 fill the combined vector with the elements of the right vector. If the left vector has not reached the end, then the right vector must have reached the end, and lines 9697 fill the combined vector with the elements of the left vector. Finally, lines 101102 copy the combined vector into the original vector. [Figure 20.7](#) creates and uses a `MergeSort` object. The output from this program displays the splits and merges performed by merge sort, showing the progress of the sort at each step of the algorithm.

**Figure 20.7. MergeSort test program.**

(This item is displayed on pages 989 - 991 in the print version)

```

1 // Fig 20.7: Fig20_07.cpp
2 // MergeSort test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "MergeSort.h" // class MergeSort definition
8
9 int main()
10 {
11 // create object to perform merge sort
12 MergeSort sortVector(10);
13
14 cout << "Unsorted vector:" << endl;
15 sortVector.displayElements(); // print unsorted vector
16 cout << endl << endl;
17

```

```
18 sortVector.sort(); // sort vector
19
20 cout << "Sorted vector:" << endl;
21 sortVector.displayElements(); // print sorted vector
22 cout << endl;
23 return 0;
24 } // end main
```

Unsorted vector:

30 47 22 67 79 18 60 78 26 54

split:    30 47 22 67 79 18 60 78 26 54  
            30 47 22 67 79  
                                        18 60 78 26 54

split:    30 47 22 67 79  
            30 47 22  
                                        67 79

split:    30 47 22  
            30 47  
                                        22

split:    30 47  
            30  
                                        47

merge:    30  
            47  
            30 47

merge:    30 47  
            22  
            22 30 47

split:        67 79  
            67  
                                        79

merge:        67  
            79  
            67 79

merge:    22 30 47  
            67 79  
            22 30 47 67 79

```

split: 18 60 78 26 54
 18 60 78
 26 54

split: 18 60 78
 18 60
 78

split: 18 60
 18
 60

merge: 18
 60
 18 60

merge: 18 60
 78
 18 60 78

split: 26 54
 26
 54

merge: 26
 54
 26 54

merge: 18 60 78
 26 54
 18 26 54 60 78

merge: 22 30 47 67 79
 18 26 54 60 78
 18 22 26 30 47 54 60 67 78 79

Sorted vector:
 18 22 26 30 47 54 60 67 78 79

```

## Efficiency of Merge Sort

Merge sort is a far more efficient algorithm than either insertion sort or selection sort (although that may be difficult to believe when looking at the rather busy Fig. 20.7). Consider the first (nonrecursive) call to

function `sortSubVector`. This results in two recursive calls to function `sortSubVector` with subvectors each approximately half the size of the original vector, and a single call to function `merge`. This call to function `merge` requires, at worst,  $n - 1$  comparisons to fill the original vector, which is  $O(n)$ . (Recall that each element in the vector is chosen by comparing one element from each of the subvectors.) The two calls to function `sortSubVector` result in four more recursive calls to function `sortSubVector`, each with a subvector approximately one quarter the size of the original vector, along with two calls to function `merge`. These two calls to function `merge` each require, at worst,  $n/2 - 1$  comparisons, for a total number of comparisons of  $O(n)$ . This process continues, each call to `sortSubVector` generating two additional calls to `sortSubVector` and a call to `merge`, until the algorithm has split the vector into one-element subvectors. At each level,  $O(n)$  comparisons are required to merge the subvectors. Each level splits the size of the vectors in half, so doubling the size of the vector requires one more level. Quadrupling the size of the vector requires two more levels. This pattern is logarithmic and results in  $\log_2 n$  levels. This results in a total efficiency of  **$O(n \log n)$** . Figure 20.8 summarizes many of the searching and sorting algorithms covered in this book and lists the Big O for each of them. Figure 20.9 lists the Big O values we have covered in this chapter along with a number of values for  $n$  to highlight the differences in the growth rates.

[Page 991]

**Figure 20.8. Searching and sorting algorithms with Big O values.**

[Page 992]

Algorithm	Location	Big O
Searching Algorithms		
Linear search	<a href="#">Section 7.7</a>	$O(n)$
Binary search	<a href="#">Section 20.2.2</a>	$O(\log n)$
Recursive linear search	<a href="#">Exercise 20.8</a>	$O(n)$
Recursive binary search	<a href="#">Exercise 20.9</a>	$O(\log n)$
Sorting Algorithms		
Insertion sort	<a href="#">Section 7.8</a>	$O(n^2)$
Selection sort	<a href="#">Section 8.6</a>	$O(n^2)$
Merge sort	<a href="#">Section 20.3.3</a>	$O(n \log n)$
Bubble sort	<a href="#">Exercises 16.3 and 16.4</a>	$O(n^2)$

Quick sort	Exercise 20.10	Worst case: $O(n^2)$
		Average case: $O(n \log n)$

---

[Page 992]

**Figure 20.9. Approximate number of comparisons for common Big O notations.**

n	Approximate decimal value	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
$2^{10}$	1000	10	$2^{10}$	$10 \cdot 2^{10}$	$2^{20}$
$2^{20}$	1,000,000	20	$2^{20}$	$20 \cdot 2^{20}$	$2^{40}$
$2^{30}$	1,000,000,000	30	$2^{30}$	$30 \cdot 2^{30}$	$2^{60}$

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 992 (continued)]

## 20.4. Wrap-Up

This chapter discussed searching and sorting data. We discussed the binary search algorithm faster, but more complex searching algorithm than linear search ([Section 7.7](#)). The binary search algorithm will only work on a sorted array, but each iteration of binary search eliminates half of the elements in the array. You also learned the merge sort algorithm which is a more efficient sorting algorithm than either insertion sort ([Section 7.8](#)) or selection sort ([Section 8.6](#)). We also introduced Big O notation, which helps you express the efficiency of an algorithm. Big O notation measures the worst case runtime for an algorithm. The Big O value of an algorithm is useful for comparing algorithms to choose the most efficient one. In the next chapter, you will learn about dynamic data structures that can grow or shrink at execution time.

[◀ PREV](#)[NEXT ▶](#)

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 993]

- A major difference among searching algorithms is the amount of effort they require in order to return a result.
- One way to describe the efficiency of an algorithm is with Big O notation ( $O$ ), which indicates how hard an algorithm may have to work to solve a problem.
- For searching and sorting algorithms, Big O describes how the amount of effort of a particular algorithm varies depending on how many elements are in the data.
- An algorithm that is  $O(1)$  is said to have a constant runtime. This does not mean that the algorithm requires only one comparison. It just means that the number of comparisons does not grow as the size of the vector increases.
- An  $O(n)$  algorithm is referred to as having a linear runtime.
- Big O is designed to highlight dominant factors and ignore terms that become unimportant with high values of  $n$ .
- Big O notation is concerned with the growth rate of algorithm runtimes, so constants are ignored.
- The linear search algorithm runs in  $O(n)$  time.
- The worst case in linear search is that every element must be checked to determine whether the search element exists. This occurs if the search key is the last element in the vector or is not present.
- The binary search algorithm is more efficient than the linear search algorithm, but it requires that the vector first be sorted. This is only worthwhile when the vector, once sorted, will be searched a great many times or when the searching application has stringent performance requirements.
- The first iteration of binary search tests the middle element in the vector. If this is the search key, the algorithm returns its location. If the search key is less than the middle element, binary search continues with the first half of the vector. If the search key is greater than the middle element, binary search continues with the second half of the vector. Each iteration of binary search tests the middle value of the remaining vector and, if the element is not found, eliminates half of the remaining elements.
- Binary search is more efficient than linear search because, with each comparison it eliminates from consideration half of the elements in the vector.
- Binary search runs in  $O(\log n)$  time because each step removes half of the remaining elements from consideration.
- If the size of the vector is doubled, binary search requires only one extra comparison to complete successfully.
- Selection sort is a simple, but inefficient, sorting algorithm.
- The first iteration of selection sort selects the smallest element in the vector and swaps it with the first element. The second iteration of selection sort selects the second-smallest element (which is the smallest remaining element) and swaps it with the second element. Selection sort continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index. At the  $i$ th iteration of selection sort, the smallest  $i$  elements of the whole vector are sorted into the first  $i$  indices.
- The selection sort algorithm runs in  $O(n^2)$  time.
- The first iteration of insertion sort takes the second element in the vector and, if it is less than the first element, swaps it with the first element. The second iteration of insertion sort looks at the third element and inserts it in the correct position with respect to the first two elements. After the  $i$ th iteration of insertion sort, the first  $i$  elements in the original vector are sorted. Only  $n - 1$  iterations are required.

- The insertion sort algorithm runs in  $O(n^2)$  time.
- Merge sort is a sorting algorithm that is faster, but more complex to implement, than selection sort and insertion sort.
- The merge sort algorithm sorts a vector by splitting the vector into two equal-sized subvectors, sorting each subvector and merging the subvectors into one larger vector.
- Merge sort's base case is a vector with one element. A one-element vector is already sorted, so merge sort immediately returns when it is called with a one-element vector. The merge part of merge sort takes two sorted vectors (these could be one-element vectors) and combines them into one larger sorted vector.
- Merge sort performs the merge by looking at the first element in each vector, which is also the smallest element in the vector. Merge sort takes the smallest of these and places it in the first element of the larger, sorted vector. If there are still elements in the subvector, merge sort looks at the second element in that subvector (which is now the smallest element remaining) and compares it to the first element in the other subvector. Merge sort continues this process until the larger vector is filled.
- In the worst case, the first call to merge sort has to make  $O(n)$  comparisons to fill the  $n$  slots in the final vector.
- The merging portion of the merge sort algorithm is performed on two subvectors, each of approximately size  $n/2$ . Creating each of these subvectors requires  $n/2 - 1$  comparisons for each subvector, or  $O(n)$  comparisons total. This pattern continues, as each level works on twice as many vectors, but each is half the size of the previous vector.
- Similar to binary search, this halving results in  $\log n$  levels, each level requiring  $O(n)$  comparisons, for a total efficiency of  $O(n \log n)$ .

 PREVNEXT 

page footer

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 994 (continued)]

## Terminology

Big O notation

binary search

constant runtime

linear runtime

logarithmic runtime

merge sort

merge two vectors

$O(1)$

$O(\log n)$

$O(n \log n)$

$O(n)$

$O(n^2)$

order 1

order  $\log n$

order  $n$

order  $n$ -squared

quadratic runtime

random-access iterator

search key

searching data

sort key

sort Standard Library function

sorting data

split the vector in merge sort

worst-case runtime for an algorithm

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 995]

### 20.3

In what sense is the insertion sort superior to the merge sort? In what sense is the merge sort superior to the insertion sort?

### 20.4

In the text, we say that after the merge sort splits the vector into two subvectors, it then sorts these two subvectors and merges them. Why might someone be puzzled by our statement that "it then sorts these two subvectors"?

 PREV

[page footer](#)

NEXT 

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 995 (continued)]

## Answers to Self-Review Exercises

- 20.1** a) 16, because an  $O(n^2)$  algorithm takes 16 times as long to sort four times as much information. b)  $O(n \log n)$ .
- 20.2** Both of these algorithms incorporate "halving" somehow reducing something by half. The binary search eliminates from consideration one-half of the vector after each comparison. The merge sort splits the vector in half each time it is called.
- 20.3** The insertion sort is easier to understand and to implement than the merge sort. The merge sort is far more efficient ( $O(n \log n)$ ) than the insertion sort ( $O(n^2)$ ).
- 20.4** In a sense, it does not really sort these two subvectors. It simply keeps splitting the original vector in half until it provides a one-element subvector, which is, of course, sorted. It then builds up the original two subvectors by merging these one-element vectors to form larger subvectors, which are then merged, and so on.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 996]

## 20.7

(Bucket Sort) A bucket sort begins with a one-dimensional vector of positive integers to be sorted and a two-dimensional vector of integers with rows indexed from 0 to 9 and columns indexed from 0 to  $n - 1$ , where  $n$  is the number of values to be sorted. Each row of the two-dimensional vector is referred to as a bucket. Write a class named `BucketSort` containing a function called `sort` that operates as follows:

a.

Place each value of the one-dimensional vector into a row of the bucket vector, based on the value's "ones" (rightmost) digit. For example, 97 is placed in row 7, 3 is placed in row 3 and 100 is placed in row 0. This procedure is called a distribution pass.

b.

Loop through the bucket vector row by row, and copy the values back to the original vector. This procedure is called a gathering pass. The new order of the preceding values in the one-dimensional vector is 100, 3 and 97.

c.

Repeat this process for each subsequent digit position (tens, hundreds, thousands, etc.).

On the second (tens digit) pass, 100 is placed in row 0, 3 is placed in row 0 (because 3 has no tens digit) and 97 is placed in row 9. After the gathering pass, the order of the values in the one-dimensional vector is 100, 3 and 97. On the third (hundreds digit) pass, 100 is placed in row 1, 3 is placed in row 0 and 97 is placed in row 0 (after the 3). After this last gathering pass, the original vector is in sorted order.

Note that the two-dimensional vector of buckets is 10 times the length of the integer vector being sorted. This sorting technique provides better performance than a bubble sort, but requires much more memory—the bubble sort requires space for only one additional element of data. This comparison is an example of the spacetime trade-off: The bucket sort uses more memory than the bubble sort, but performs better. This version of the bucket sort requires copying all the data back to the original vector on each pass. Another possibility is to create a second two-dimensional bucket vector and repeatedly swap the data between the two bucket vectors.

## 20.8

(Recursive Linear Search) Modify [Exercise 7.33](#) to use recursive function `recursiveLinearSearch` to

perform a linear search of the vector. The function should receive the search key and starting index as arguments. If the search key is found, return its index in the vector; otherwise, return 1. Each call to the recursive function should check one index in the vector.

## 20.9

(Recursive Binary Search) Modify Fig. 20.3 to use recursive function `recursiveBinarySearch` to perform a binary search of the vector. The function should receive the search key, starting index and ending index as arguments. If the search key is found, return its index in the vector. If the search key is not found, return -1.

## 20.10

(Quicksort) The recursive sorting technique called quicksort uses the following basic algorithm for a one-dimensional vector of values:

a.

Partitioning Step : Take the first element of the unsorted vector and determine its final location in the sorted vector (i.e., all values to the left of the element in the vector are less than the element, and all values to the right of the element in the vector are greater than the element we show how to do this below). We now have one element in its proper location and two unsorted subvectors.

b.

Recursion Step : Perform Step 1 on each unsorted subvector. Each time Step 1 is performed on a subvector, another element is placed in its final location of the sorted vector, and two unsorted subvectors are created. When a subvector consists of one element, that element is in its final location (because a one-element vector is already sorted).

The basic algorithm seems simple enough, but how do we determine the final position of the first element of each subvector? As an example, consider the following set of values (the element in bold is the partitioning element it will be placed in its final location in the sorted vector):

37    2    6    4    89    8    10    12    68    45

[Page 997]

Starting from the rightmost element of the vector, compare each element with 37 until an element less than 37 is found; then swap 37 and that element. The first element less than 37 is 12, so 37 and 12 are swapped. The new vector is

12   2   6   4   89   8   10   37   68   45

Element 12 is in italics to indicate that it was just swapped with 37.

Starting from the left of the vector, but beginning with the element after 12, compare each element with 37 until an element greater than 37 is found then swap 37 and that element. The first element greater than 37 is 89, so 37 and 89 are swapped.

The new vector is

12 2 6 4 37 8 10 89 68 45

Starting from the right, but beginning with the element before 89, compare each element with 37 until an element less than 37 is found then swap 37 and that element.

The first element less than 37 is 10, so 37 and 10 are swapped. The new vector is

12 2 6 4 10 8 37 89 68 45

Starting from the left, but beginning with the element after 10, compare each element with 37 until an element greater than 37 is found then swap 37 and that element. There are no more elements greater than 37, so when we compare 37 with itself, we know that 37 has been placed in its final location of the sorted vector. Every value to the left of 37 is smaller than it, and every value to the right of 37 is larger than it.

Once the partition has been applied on the previous vector, there are two unsorted subvectors. The subvector with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subvector with values greater than 37 contains 89, 68 and 45. The sort continues recursively, with both subvectors being partitioned in the same manner as the original vector.

Based on the preceding discussion, write recursive function `quickSortHelper` to sort a one-dimensional integer vector. The function should receive as arguments a starting index and an ending index on the original vector being sorted.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 999]

## Outline

[21.1 Introduction](#)

[21.2 Self-Referential Classes](#)

[21.3 Dynamic Memory Allocation and Data Structures](#)

[21.4 Linked Lists](#)

[21.5 Stacks](#)

[21.6 Queues](#)

[21.7 Trees](#)

[21.8 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

[Exercises](#)

[Special Section](#)[Building Your Own Compiler](#)

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1000]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1001]

## Common Programming Error 21.1



Not setting the link in the last node of a linked data structure to null (0) is a (possibly fatal) logic error.

[◀ PREV](#)

page footer

[NEXT ▶](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1001 (continued)]

## 21.3. Dynamic Memory Allocation and Data Structures

Creating and maintaining dynamic data structures requires dynamic memory allocation, which enables a program to obtain more memory at execution time to hold new nodes. When that memory is no longer needed by the program, the memory can be released so that it can be reused to allocate other objects in the future. The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available virtual memory in a virtual memory system. Often, the limits are much smaller, because available memory must be shared among many programs.

The `new` operator takes as an argument the type of the object being dynamically allocated and returns a pointer to an object of that type. For example, the statement

```
Node *newPtr = new Node(10); // create Node with data 10
```

allocates `sizeof( Node )` bytes, runs the `Node` constructor and assigns the address of the new `Node` object to `newPtr`. If no memory is available, `new` throws a `bad_alloc` exception. The value 10 is passed to the `Node` constructor which initializes the `Node`'s `data` member to 10.

The `delete` operator runs the `Node` destructor and deallocates memory allocated with `new`. The memory is returned to the system so that the memory can be reallocated in the future. To free memory dynamically allocated by the preceding `new`, use the statement

```
delete newPtr;
```

Note that `newPtr` itself is not deleted; rather the space `newPtr` points to is deleted. If pointer `newPtr` has the null pointer value 0, the preceding statement has no effect. It is not an error to delete a null pointer.

The following sections discuss lists, stacks, queues and trees. The data structures presented in this chapter are created and maintained with dynamic memory allocation, self-referential classes, class templates and function templates.

 PREV

NEXT 

page footer



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1002]

### Performance Tip 21.1



An array can be declared to contain more elements than the number of items expected, but this can waste memory. Linked lists can provide better memory utilization in these situations. Linked lists allow the program to adapt at runtime. Note that class template `vector` (introduced in [Section 7.11](#)) implements a dynamically resizable array-based data structure.

Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list. Existing list elements do not need to be moved.

### Performance Tip 21.2



Insertion and deletion in a sorted array can be time consuming all the elements following the inserted or deleted element must be shifted appropriately. A linked list allows efficient insertion operations anywhere in the list.

### Performance Tip 21.3

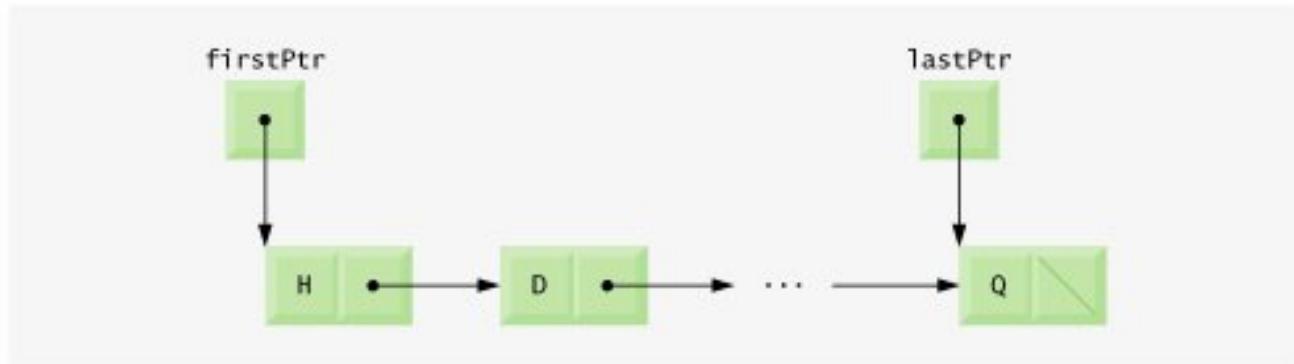


The elements of an array are stored contiguously in memory. This allows immediate access to any array element, because the address of any element can be calculated directly based on its position relative to the beginning of the array. Linked lists do not afford such immediate "direct access" to their elements. So accessing individual elements in a linked list can be considerably more expensive than accessing individual elements in an array. The selection of a data structure is typically based on the performance of specific operations used by a program and the order in which the data items are maintained in the data structure. For example, it is typically more efficient to insert an item in a sorted linked list than a sorted array.

Linked list nodes are normally not stored contiguously in memory. Logically, however, the nodes of a linked list appear to be contiguous. [Figure 21.2](#) illustrates a linked list with several nodes.

**Figure 21.2. A graphical representation of a list.**

[\[View full size image\]](#)



### Performance Tip 21.4



Using dynamic memory allocation (instead of fixed-size arrays) for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that pointers occupy space and that dynamic memory allocation incurs the overhead of function calls.

---

[Page 1003]

## Linked List Implementation

The program of Figs. 21.321.5 uses a `List` class template (see Chapter 14 for information on class templates) to manipulate a list of integer values and a list of floating-point values. The driver program (Fig. 21.5) provides five options: 1) Insert a value at the beginning of the list, 2) insert a value at the end of the list, 3) delete a value from the beginning of the list, 4) delete a value from the end of the list and 5) end the list processing. A detailed discussion of the program follows. Exercise 21.20 asks you to implement a recursive function that prints a linked list backward, and Exercise 21.21 asks you to implement a recursive function that searches a linked list for a particular data item.

**Figure 21.3. ListNode class-template definition.**

(This item is displayed on pages 1003 - 1004 in the print version)

```

1 // Fig. 21.3: Listnode.h
2 // Template ListNode class definition.
3 #ifndef LISTNODE_H
4 #define LISTNODE_H
5
6 // forward declaration of class List required to announce that class
7 // List exists so it can be used in the friend declaration at line 13
8 template< typename NODETYPE > class List;
9
10 template< typename NODETYPE>
11 class ListNode
12 {
13 friend class List< NODETYPE >; // make List a friend
14
15 public:
16 ListNode(const NODETYPE &); // constructor
17 NODETYPE getData() const; // return data in node
18 private:
19 NODETYPE data; // data
20 ListNode< NODETYPE > *nextPtr; // next node in list
21 }; // end class ListNode
22
23 // constructor
24 template< typename NODETYPE>
25 ListNode< NODETYPE >::ListNode(const NODETYPE &info)
26 : data(info), nextPtr(0)
27 {
28 // empty body
29 } // end ListNode constructor
30
31 // return copy of data in node
32 template< typename NODETYPE >
33 NODETYPE ListNode< NODETYPE >::getData() const
34 {
35 return data;
36 } // end function getData
37
38 #endif

```

The program uses class templates `ListNode` (Fig. 21.3) and `List` (Fig. 21.4). Encapsulated in each `List` object is a linked list of `ListNode` objects. Class template `ListNode` (Fig. 21.3) contains `private` members `data` and `nextPtr` (lines 1920), a constructor to initialize these members and function `getdata` to return the data in a node. Member `data` stores a value of type `NODETYPE`, the type parameter passed to the class template. Member `nextPtr` stores a pointer to the next `ListNode` object in the linked list. Note that line 13 of the `ListNode` class template definition declares class `List< NODETYPE >` as a

friend. This makes all member functions of a given specialization of class template `List` friends of the corresponding specialization of class template `ListNode`, so they can access the private members of `ListNode` objects of that type. Because the `ListNode` template parameter `NODETYPE` is used as the template argument for `List` in the friend declaration, `ListNodes` specialized with a particular type can be processed only by a `List` specialized with the same type (e.g., a `List` of `int` values manages `ListNode` objects that store `int` values).

[Page 1011]

**Figure 21.4. List class-template definition.**

(This item is displayed on pages 1004 - 1007 in the print version)

```

1 // Fig. 21.4: List.h
2 // Template List class definition.
3 #ifndef LIST_H
4 #define LIST_H
5
6 #include <iostream>
7 using std::cout;
8
9 #include "listnode.h" // ListNode class definition
10
11 template< typename NODETYPE >
12 class List
13 {
14 public:
15 List(); // constructor
16 ~List(); // destructor
17 void insertAtFront(const NODETYPE &);
18 void insertAtBack(const NODETYPE &);
19 bool removeFromFront(NODETYPE &);
20 bool removeFromBack(NODETYPE &);
21 bool isEmpty() const;
22 void print() const;
23 private:
24 ListNode< NODETYPE > *firstPtr; // pointer to first node
25 ListNode< NODETYPE > *lastPtr; // pointer to last node
26
27 // utility function to allocate new node
28 ListNode< NODETYPE > *getNewNode(const NODETYPE &);
29 }; // end class List
30
31 // default constructor
32 template< typename NODETYPE >
33 List< NODETYPE >::List()

```

```

34 : firstPtr(0), lastPtr(0)
35 {
36 // empty body
37 } // end List constructor
38
39 // destructor
40 template< typename NODETYPE >
41 List< NODETYPE >::~List()
42 {
43 if (!isEmpty()) // List is not empty
44 {
45 cout << "Destroying nodes ...\\n";
46
47 ListNode< NODETYPE > *currentPtr = firstPtr;
48 ListNode< NODETYPE > *tempPtr;
49
50 while (currentPtr != 0) // delete remaining nodes
51 {
52 tempPtr = currentPtr;
53 cout << tempPtr->data << '\\n';
54 currentPtr = currentPtr->nextPtr;
55 delete tempPtr;
56 } // end while
57 } // end if
58
59 cout << "All nodes destroyed\\n\\n";
60 } // end List destructor
61
62 // insert node at front of list
63 template< typename NODETYPE >
64 void List< NODETYPE >::insertAtFront(const NODETYPE &value)
65 {
66 ListNode< NODETYPE > *newPtr = getNewNode(value); // new node
67
68 if (isEmpty()) // List is empty
69 firstPtr = lastPtr = newPtr; // new list has only one node
70 else // List is not empty
71 {
72 newPtr->nextPtr = firstPtr; // point new node to previous 1st node
73 firstPtr = newPtr; // aim firstPtr at new node
74 } // end else
75 } // end function insertAtFront
76
77 // insert node at back of list
78 template< typename NODETYPE >
79 void List< NODETYPE >::insertAtBack(const NODETYPE &value)
80 {
81 ListNode< NODETYPE > *newPtr = getNewNode(value); // new node
82

```

```

83 if (isEmpty()) // List is empty
84 firstPtr = lastPtr = newPtr; // new list has only one node
85 else // List is not empty
86 {
87 lastPtr->nextPtr = newPtr; // update previous last node
88 lastPtr = newPtr; // new last node
89 } // end else
90 } // end function insertAtBack
91
92 // delete node from front of list
93 template< typename NODETYPE >
94 bool List< NODETYPE >::removeFromFront(NODETYPE &value)
95 {
96 if (isEmpty()) // List is empty
97 return false; // delete unsuccessful
98 else
99 {
100 ListNode< NODETYPE > *tempPtr = firstPtr; // hold tempPtr to delete
101
102 if (firstPtr == lastPtr)
103 firstPtr = lastPtr = 0; // no nodes remain after removal
104 else
105 firstPtr = firstPtr->nextPtr; // point to previous 2nd node
106
107 value = tempPtr->data; // return data being removed
108 delete tempPtr; // reclaim previous front node
109 return true; // delete successful
110 } // end else
111 } // end function removeFromFront
112
113 // delete node from back of list
114 template< typename NODETYPE >
115 bool List< NODETYPE >::removeFromBack(NODETYPE &value)
116 {
117 if (isEmpty()) // List is empty
118 return false; // delete unsuccessful
119 else
120 {
121 ListNode< NODETYPE > *tempPtr = lastPtr; // hold tempPtr to delete
122
123 if (firstPtr == lastPtr) // List has one element
124 firstPtr = lastPtr = 0; // no nodes remain after removal
125 else
126 {
127 ListNode< NODETYPE > *currentPtr = firstPtr;
128
129 // locate second-to-last element
130 while (currentPtr->nextPtr != lastPtr)
131 currentPtr = currentPtr->nextPtr; // move to next node

```

```

132
133 lastPtr = currentPtr; // remove last node
134 currentPtr->nextPtr = 0; // this is now the last node
135 } // end else
136
137 value = tempPtr->data; // return value from old last node
138 delete tempPtr; // reclaim former last node
139 return true; // delete successful
140 } // end else
141 } // end function removeFromBack
142
143 // is List empty?
144 template< typename NODETYPE >
145 bool List< NODETYPE >::isEmpty() const
146 {
147 return firstPtr == 0;
148 } // end function isEmpty
149
150 // return pointer to newly allocated node
151 template< typename NODETYPE >
152 ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
153 const NODETYPE &value)
154 {
155 return new ListNode< NODETYPE >(value);
156 } // end function getNewNode
157
158 // display contents of List
159 template< typename NODETYPE >
160 void List< NODETYPE >::print() const
161 {
162 if (isEmpty()) // List is empty
163 {
164 cout << "The list is empty\n\n";
165 return;
166 } // end if
167
168 ListNode< NODETYPE > *currentPtr = firstPtr;
169
170 cout << "The list is: ";
171
172 while (currentPtr != 0) // get element data
173 {
174 cout << currentPtr->data << ' ';
175 currentPtr = currentPtr->nextPtr;
176 } // end while
177
178 cout << "\n\n";
179 } // end function print
180

```

181 #endif

**Figure 21.5. Manipulating a linked list.**

(This item is displayed on pages 1007 - 1010 in the print version)

```

1 // Fig. 21.5: Fig21_05.cpp
2 // List class test program.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9 using std::string;
10
11 #include "List.h" // List class definition
12
13 // function to test a List
14 template< typename T >
15 void testList(List< T > &listObject, const string &typeName)
16 {
17 cout << "Testing a List of " << typeName << " values\n";
18 instructions(); // display instructions
19
20 int choice; // store user choice
21 T value; // store input value
22
23 do // perform user-selected actions
24 {
25 cout << "? ";
26 cin >> choice;
27
28 switch (choice)
29 {
30 case 1: // insert at beginning
31 cout << "Enter " << typeName << ": ";
32 cin >> value;
33 listObject.insertAtFront(value);
34 listObject.print();
35 break;
36 case 2: // insert at end
37 cout << "Enter " << typeName << ": ";
38 cin >> value;
39 listObject.insertAtBack(value);

```

```
40 listObject.print();
41 break;
42 case 3: // remove from beginning
43 if (listObject.removeFromFront(value))
44 cout << value << " removed from list\n";
45
46 listObject.print();
47 break;
48 case 4: // remove from end
49 if (listObject.removeFromBack(value))
50 cout << value << " removed from list\n";
51
52 listObject.print();
53 break;
54 } // end switch
55 } while (choice != 5); // end do...while
56
57 cout << "End list test\n\n";
58 } // end function testList
59
60 // display program instructions to user
61 void instructions()
62 {
63 cout << "Enter one of the following:\n"
64 << " 1 to insert at beginning of list\n"
65 << " 2 to insert at end of list\n"
66 << " 3 to delete from beginning of list\n"
67 << " 4 to delete from end of list\n"
68 << " 5 to end list processing\n";
69 } // end function instructions
70
71 int main()
72 {
73 // test List of int values
74 List< int > integerList;
75 testList(integerList, "integer");
76
77 // test List of double values
78 List< double > doubleList;
79 testList(doubleList, "double");
80 return 0;
81 } // end main
```

```
Testing a List of integer values
Enter one of the following:
 1 to insert at beginning of list
 2 to insert at end of list
 3 to delete from beginning of list
 4 to delete from end of list
 5 to end list processing
? 1
Enter integer: 1
The list is: 1

? 1
Enter integer: 2
The list is: 2 1

? 2
Enter integer: 3
The list is: 2 1 3

? 2
Enter integer: 4
The list is: 2 1 3 4

? 3
2 removed from list
The list is: 1 3 4

? 3
1 removed from list
The list is: 3 4

? 4
4 removed from list
The list is: 3

? 4
3 removed from list
The list is empty

? 5
End list test

Testing a List of double values
Enter one of the following:
 1 to insert at beginning of list
 2 to insert at end of list
 3 to delete from beginning of list
 4 to delete from end of list
 5 to end list processing
```

```
? 1
Enter double: 1.1
The list is: 1.1

? 1
Enter double: 2.2
The list is: 2.2 1.1

? 2
Enter double: 3.3
The list is: 2.2 1.1 3.3

? 2
Enter double: 4.4
The list is: 2.2 1.1 3.3 4.4

? 3
2.2 removed from list
The list is: 1.1 3.3 4.4

? 3
1.1 removed from list
The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3

? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed
```

Lines 2425 of the `List` class template (Fig. 21.4) declare private data members `firstPtr` (a pointer to the first `ListNode` in a `List`) and `lastPtr` (a pointer to the last `ListNode` in a `List`). The default constructor (lines 3237) initializes both pointers to 0 (null). The destructor (lines 4060) ensures that all `ListNode` objects in a `List` object are destroyed when that `List` object is destroyed. The primary

List functions are `insertAtFront` (lines 6375), `insertAtBack` (lines 7890), `removeFromFront` (lines 93111) and `removeFromBack` (lines 114141).

Function `isEmpty` (lines 144148) is called a predicate function it does not alter the List; rather, it determines whether the List is empty (i.e., the pointer to the first node of the List is null). If the List is empty, `true` is returned; otherwise, `false` is returned. Function `print` (lines 159179) displays the List's contents. Utility function `getListNode` (lines 151156) returns a dynamically allocated `ListNode` object. This function is called from functions `insertAtFront` and `insertAtBack`.

## Error-Prevention Tip 21.1



Assign null (0) to the link member of a new node. Pointers should be initialized before they are used.

The driver program ([Fig. 21.5](#)) uses function template `testList` to enable the user to manipulate objects of class `List`. Lines 74 and 78 create `List` objects for types `int` and `double`, respectively. Lines 75 and 79 invoke the `testList` function template with these `List` objects.

## Member Function `insertAtFront`

Over the next several pages, we discuss each of the member functions of class `List` in detail. Function `insertAtFront` ([Fig. 21.4](#), lines 6375) places a new node at the front of the list. The function consists of several steps:

**1.**

Call function `getListNode` (line 66), passing it `value`, which is a constant reference to the node value to be inserted.

**2.**

Function `getListNode` (lines 151156) uses operator `new` to create a new list node and return a pointer to this newly allocated node, which is assigned to `newPtr` in `insertAtFront` (line 66).

**3.**

If the list is empty (line 68), then both `firstPtr` and `lastPtr` are set to `newPtr` (line 69).

**4.**

If the list is not empty (line 70), then the node pointed to by `newPtr` is threaded into the list by

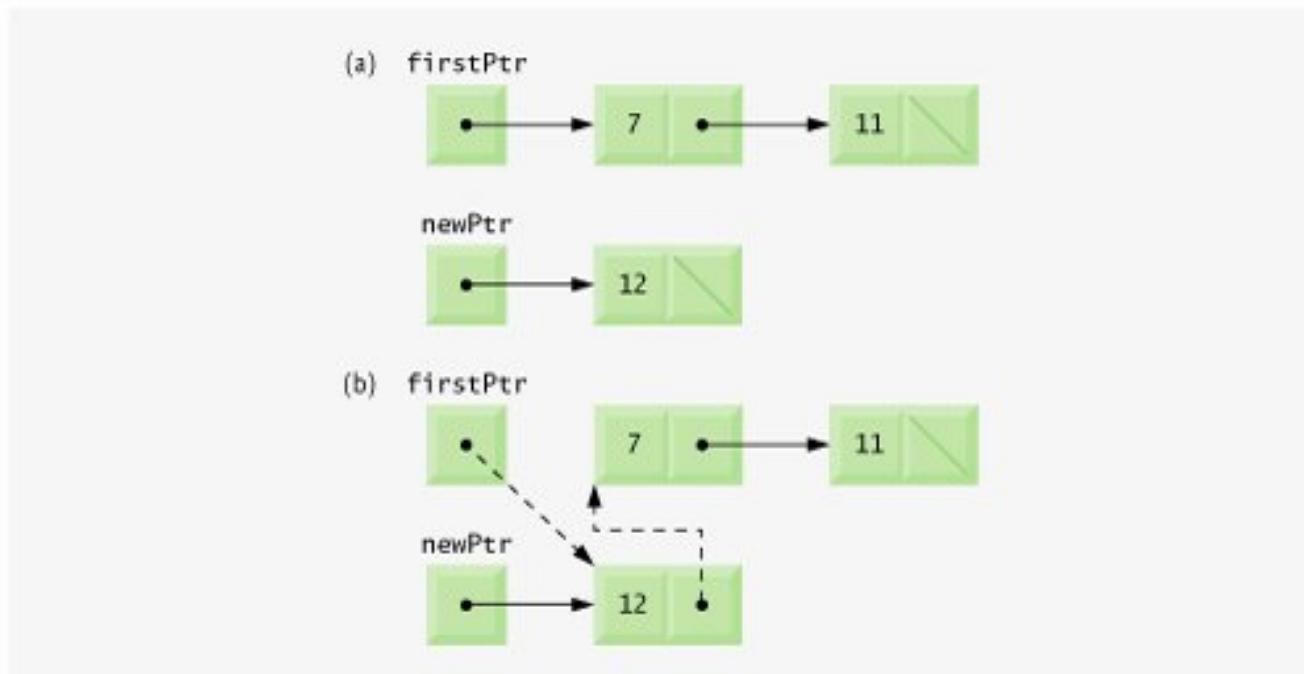
copying `firstPtr` to `newPtr->nextPtr` (line 72), so that the new node points to what used to be the first node of the list, and copying `newPtr` to `firstPtr` (line 73), so that `firstPtr` now points to the new first node of the list.

[Figure 21.6](#) illustrates function `insertAtFront`. Part (a) of the figure shows the list and the new node before the `insertAtFront` operation. The dashed arrows in part (b) illustrate Step 4 of the `insertAtFront` operation that enables the node containing 12 to become the new list front.

[Page 1012]

**Figure 21.6. Operation `insertAtFront` represented graphically.**

[\[View full size image\]](#)



## Member Function `insertAtBack`

Function `insertAtBack` (Fig. 21.4, lines 7890) places a new node at the back of the list. The function consists of several steps:

1.

Call function `getNewNode` (line 81), passing it `value`, which is a constant reference to the node value to be inserted.

2.

Function `getNewNode` (lines 151156) uses operator `new` to create a new list node and return a pointer to this newly allocated node, which is assigned to `newPtr` in `insertAtBack` (line 81).

3.

If the list is empty (line 83), then both `firstPtr` and `lastPtr` are set to `newPtr` (line 84).

4.

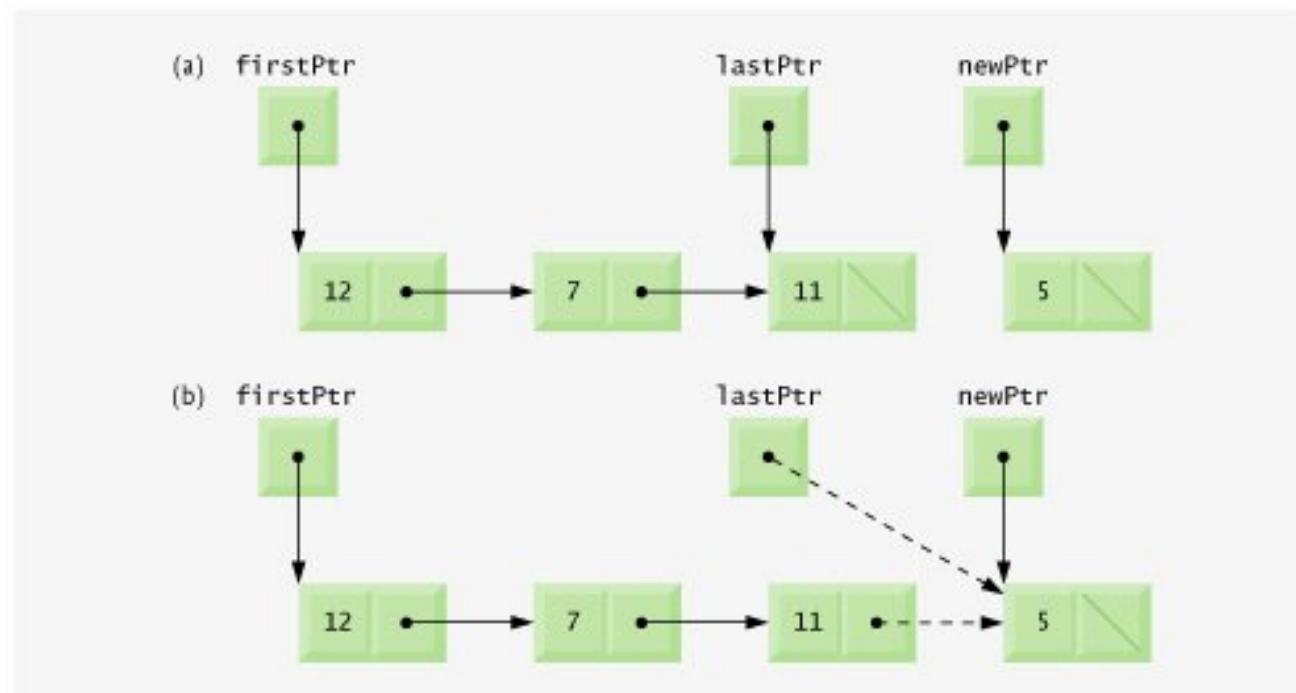
If the list is not empty (line 85), then the node pointed to by `newPtr` is threaded into the list by copying `newPtr` into `lastPtr->nextPtr` (line 87), so that the new node is pointed to by what used to be the last node of the list, and copying `newPtr` to `lastPtr` (line 88), so that `lastPtr` now points to the new last node of the list.

[Figure 21.7](#) illustrates an `insertAtBack` operation. Part (a) of the figure shows the list and the new node before the operation. The dashed arrows in part (b) illustrate Step 4 of function `insertAtBack` that enables a new node to be added to the end of a list that is not empty.

**Figure 21.7. Operation `insertAtBack` represented graphically.**

(This item is displayed on page 1013 in the print version)

[View full size image]



## Member Function `removeFromFront`

Function `removeFromFront` ([Fig. 21.4](#), lines 93111) removes the front node of the list and copies the

node value to the reference parameter. The function returns `false` if an attempt is made to remove a node from an empty list (lines 9697) and returns `TRUE` if the removal is successful. The function consists of several steps:

1.

Assign `tempPtr` the address to which `firstPtr` points (line 100). Eventually, `tempPtr` will be used to delete the node being removed.

---

[Page 1013]

2.

If `firstPtr` is equal to `lastPtr` (line 102), i.e., if the list has only one element prior to the removal attempt, then set `firstPtr` and `lastPtr` to zero (line 103) to dethread that node from the list (leaving the list empty).

3.

If the list has more than one node prior to removal, then leave `lastPtr` as is and set `firstPtr` to `firstPtr->nextPtr` (line 105); i.e., modify `firstPtr` to point to what was the second node prior to removal (and is now the new first node).

4.

After all these pointer manipulations are complete, copy to reference parameter `value` the data member of the node being removed (line 107).

5.

Now delete the node pointed to by `tempPtr` (line 108).

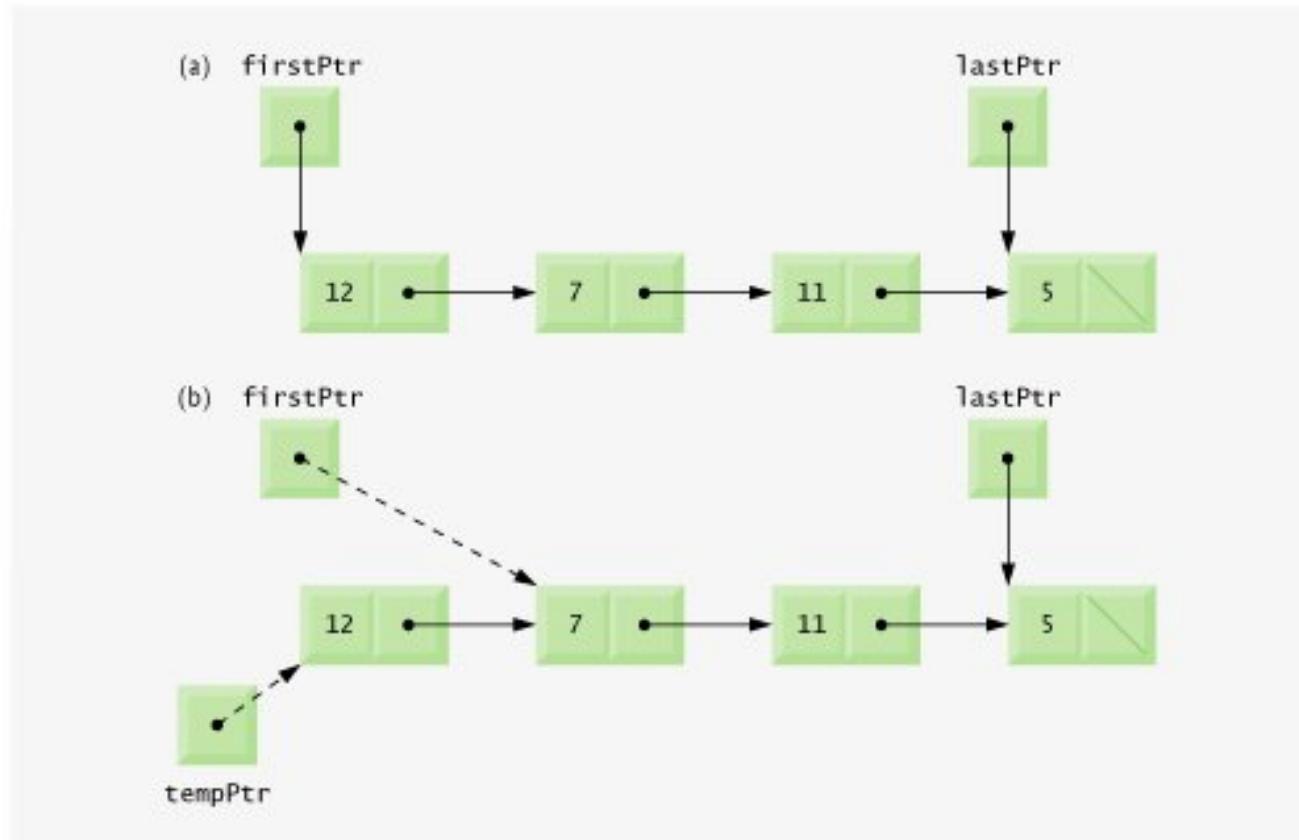
6.

Return `true`, indicating successful removal (line 109).

Figure 21.8 illustrates function `removeFromFront`. Part (a) illustrates the list before the removal operation. Part (b) shows the actual pointer manipulations for removing the front node from a nonempty list.

**Figure 21.8. Operation `removeFromFront` represented graphically.**

(This item is displayed on page 1014 in the print version)

[\[View full size image\]](#)

## Member Function `removeFromBack`

Function `removeFromBack` (Fig. 21.4, lines 114141) removes the back node of the list and copies the node value to the reference parameter. The function returns `false` if an attempt is made to remove a node from an empty list (lines 117118) and returns `true` if the removal is successful. The function consists of several steps:

1.

Assign to `tempPtr` the address to which `lastPtr` points (line 121). Eventually, `tempPtr` will be used to delete the node being removed.

2.

If `firstPtr` is equal to `lastPtr` (line 123), i.e., if the list has only one element prior to the removal attempt, then set `firstPtr` and `lastPtr` to zero (line 124) to dethread that node from the list (leaving the list empty).

3.

If the list has more than one node prior to removal, then assign `currentPtr` the address to which `firstPtr` points (line 127) to prepare to "walk the list."

---

[Page 1014]

4.

Now "walk the list" with `currentPtr` until it points to the node before the last node. This node will become the last node after the remove operation completes. This is done with a `while` loop (lines 130131) that keeps replacing `currentPtr` by `currentPtr->nextPtr`, while `currentPtr->nextPtr` is not `lastPtr`.

5.

Assign `lastPtr` to the address to which `currentPtr` points (line 133) to dethread the back node from the list.

6.

Set `currentPtr->nextPtr` to zero (line 134) in the new last node of the list.

7.

After all the pointer manipulations are complete, copy to reference parameter `value` the data member of the node being removed (line 137).

8.

Now delete the node pointed to by `tempPtr` (line 138).

9.

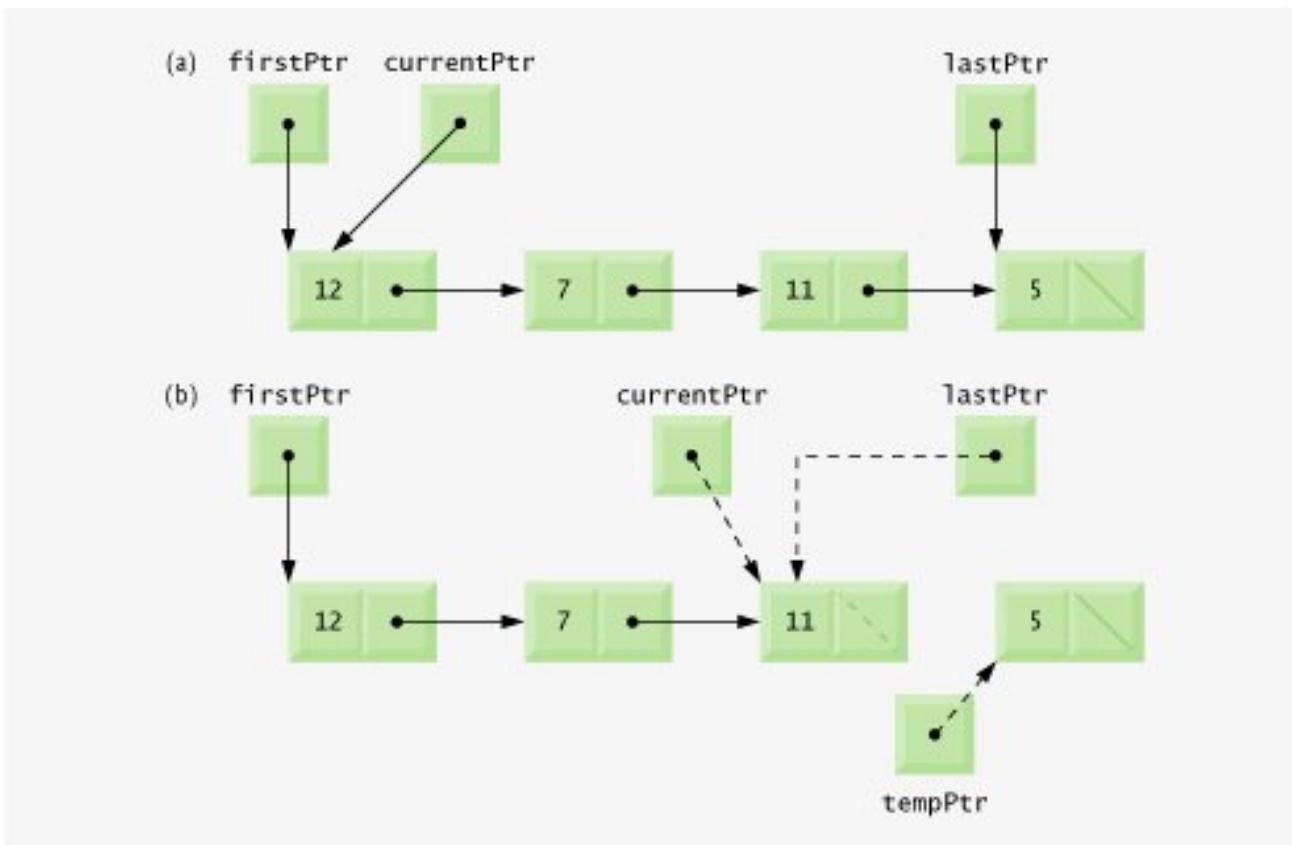
Return `true` (line 139), indicating successful removal.

[Figure 21.9](#) illustrates `removeFromBack`. Part (a) of the figure illustrates the list before the removal operation. Part (b) of the figure shows the actual pointer manipulations.

**Figure 21.9. Operation `removeFromBack` represented graphically.**

(This item is displayed on page 1015 in the print version)

[\[View full size image\]](#)



## Member Function `print`

Function `print` (lines 159179) first determines whether the list is empty (line 162). If so, it prints "The list is empty" and returns (lines 164165). Otherwise, it iterates through the list and outputs the value in each node. The function initializes `currentPtr` as a copy of `firstPtr` (line 168), then prints the string "The list is: " (line 170). While `currentPtr` is not null (line 172), `currentPtr->data` is printed (line 174) and `currentPtr` is assigned the value of `currentPtr->nextPtr` (line 175). Note that if the link in the last node of the list is not null, the printing algorithm will erroneously print past the end of the list. The printing algorithm is identical for linked lists, stacks and queues (because we base each of these data structures on the same linked list infrastructure).

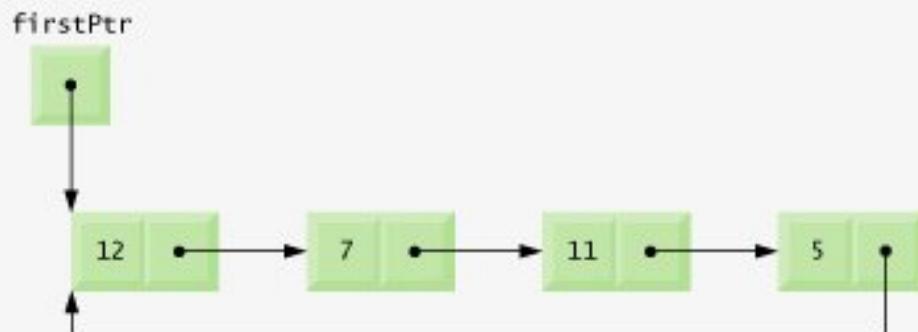
---

[Page 1015]

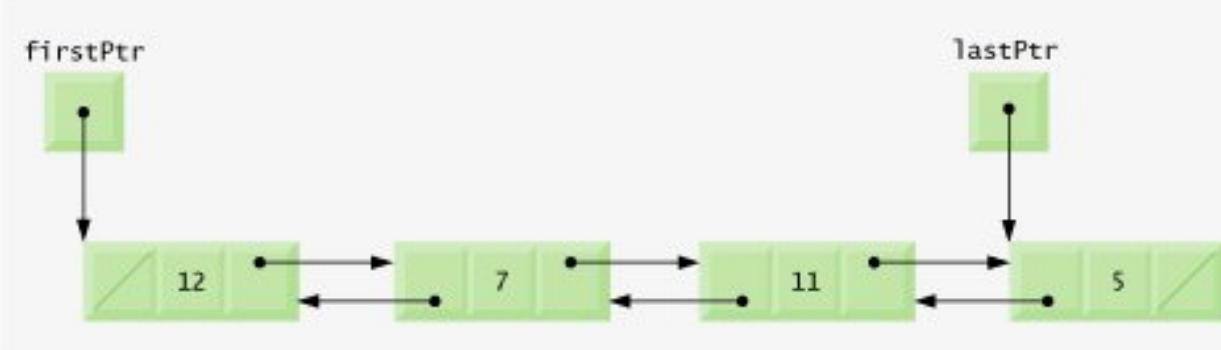
## Linear and Circular Singly Linked and Doubly Linked Lists

The kind of linked list we have been discussing is a **singly linked list**—the list begins with a pointer to the first node, and each node contains a pointer to the next node "in sequence." This list terminates with a node whose pointer member has the value 0. A singly linked list may be traversed in only one direction.

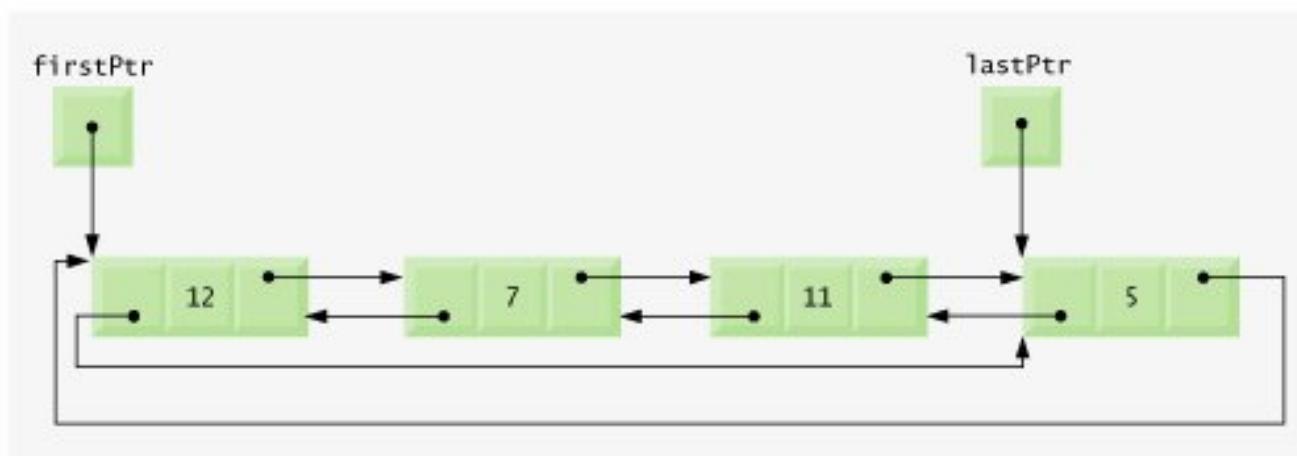
A **circular, singly linked list** (Fig. 21.10) begins with a pointer to the first node, and each node contains a pointer to the next node. The "last node" does not contain a 0 pointer; rather, the pointer in the last node points back to the first node, thus closing the "circle."

**Figure 21.10. Circular, singly linked list.**[\[View full size image\]](#)[\[Page 1016\]](#)

A **doubly linked list** (Fig. 21.11) allows traversals both forward and backward. Such a list is often implemented with two "start pointers" one that points to the first element of the list to allow front-to-back traversal of the list and one that points to the last element to allow back-to-front traversal. Each node has both a forward pointer to the next node in the list in the forward direction and a backward pointer to the next node in the list in the backward direction. If your list contains an alphabetized telephone directory, for example, a search for someone whose name begins with a letter near the front of the alphabet might begin from the front of the list. Searching for someone whose name begins with a letter near the end of the alphabet might begin from the back of the list.

**Figure 21.11. Doubly linked list.**[\[View full size image\]](#)

In a **circular, doubly linked list** (Fig. 21.12), the forward pointer of the last node points to the first node, and the backward pointer of the first node points to the last node, thus closing the "circle."

**Figure 21.12. Circular, doubly linked list.**[\[View full size image\]](#)[◀ PREV](#)[NEXT ▶](#)

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1017]

The primary member functions used to manipulate a stack are `push` and `pop`. Function `push` inserts a new node at the top of the stack. Function `pop` removes a node from the top of the stack, stores the popped value in a reference variable that is passed to the calling function and returns `true` if the `pop` operation was successful (`false` otherwise).

Stacks have many interesting applications. For example, when a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack. If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order, so that each function can return to its caller. Stacks support recursive function calls in the same manner as conventional nonrecursive calls. [Section 6.11](#) discusses the function call stack in detail.

Stacks provide the memory for, and store the values of, automatic variables on each invocation of a function. When the function returns to its caller or throws an exception, the destructor (if any) for each local object is called, the space for that function's automatic variables is popped off the stack and those variables are no longer known to the program.

Stacks are used by compilers in the process of evaluating expressions and generating machine-language code. The exercises explore several applications of stacks, including using them to develop your own complete working compiler.

We will take advantage of the close relationship between lists and stacks to implement a stack class primarily by reusing a list class. First, we implement the stack class through `private` inheritance of the list class. Then we implement an identically performing stack class through composition by including a list object as a `private` member of a stack class. Of course, all of the data structures in this chapter, including these two stack classes, are implemented as templates to encourage further reusability.

The program of [Figs. 21.13](#)[21.14](#) creates a Stack class template ([Fig. 21.13](#)) primarily through `private` inheritance (line 9) of the List class template of [Fig. 21.4](#). We want the Stack to have member functions `push` (lines 1316), `pop` (lines 1922), `isStackEmpty` (lines 2528) and `printStack` (lines 3134). Note that these are essentially the `insertAtFront`, `removeFromFront`, `isEmpty` and `print` functions of the List class template. Of course, the List class template contains other member functions (i.e., `insertAtBack` and `removeFromBack`) that we would not want to make accessible through the public interface to the Stack class. So when we indicate that the Stack class template is to inherit from the List class template, we specify `private` inheritance. This makes all the List class template's member functions `private` in the Stack class template. When we implement the Stack's member functions, we then have each of these call the appropriate member function of the List class: `push` calls `insertAtFront` (line 15), `pop` calls `removeFromFront` (line 21), `isStackEmpty` calls `isEmpty` (line 27) and `printStack` calls `print` (line 33); this is referred to as **delegation**.

**Figure 21.13. Stack class-template definition.**

(This item is displayed on pages 1017 - 1018 in the print version)

```
1 // Fig. 21.13: Stack.h
2 // Template Stack class definition derived from class List.
3 #ifndef STACK_H
4 #define STACK_H
5
6 #include "List.h" // List class definition
7
8 template< typename STACKTYPE >
9 class Stack : private List< STACKTYPE >
10 {
11 public:
12 // push calls the List function insertAtFront
13 void push(const STACKTYPE &data)
14 {
15 insertAtFront(data);
16 } // end function push
17
18 // pop calls the List function removeFromFront
19 bool pop(STACKTYPE &data)
20 {
21 return removeFromFront(data);
22 } // end function pop
23
24 // isStackEmpty calls the List function isEmpty
25 bool isStackEmpty() const
26 {
27 return isEmpty();
28 } // end function isStackEmpty
29
30 // printStack calls the List function print
31 void printStack() const
32 {
33 print();
34 } // end function print
35 }; // end class Stack
36
37 #endif
```

The stack class template is used in main (Fig. 21.14) to instantiate integer stack intStack of type Stack< int > (line 11). Integers 0 through 2 are pushed onto intStack (lines 1620), then popped off intStack (lines 2530). The program uses the Stack class template to create doubleStack of type Stack< double > (line 32). Values 1.1, 2.2 and 3.3 are pushed onto doubleStack (lines 3843), then popped off doubleStack (lines 4853).

**Figure 21.14. A simple stack program.**

(This item is displayed on pages 1018 - 1020 in the print version)

```

1 // Fig. 21.14: Fig21_14.cpp
2 // Template Stack class test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Stack.h" // Stack class definition
8
9 int main()
10 {
11 Stack< int > intStack; // create Stack of ints
12
13 cout << "processing an integer Stack" << endl;
14
15 // push integers onto intStack
16 for (int i = 0; i < 3; i++)
17 {
18 intStack.push(i);
19 intStack.printStack();
20 } // end for
21
22 int popInteger; // store int popped from stack
23
24 // pop integers from intStack
25 while (!intStack.isEmpty())
26 {
27 intStack.pop(popInteger);
28 cout << popInteger << " popped from stack" << endl;
29 intStack.printStack();
30 } // end while
31
32 Stack< double > doubleStack; // create Stack of doubles
33 double value = 1.1;
34
35 cout << "processing a double Stack" << endl;
36

```

```
37 // push floating-point values onto doubleStack
38 for (int j = 0; j < 3; j++)
39 {
40 doubleStack.push(value);
41 doubleStack.printStack();
42 value += 1.1;
43 } // end for
44
45 double popDouble; // store double popped from stack
46
47 // pop floating-point values from doubleStack
48 while (!doubleStack.isEmpty())
49 {
50 doubleStack.pop(popDouble);
51 cout << popDouble << " popped from stack" << endl;
52 doubleStack.printStack();
53 } // end while
54
55 return 0;
56 } // end main
```

```
processing an integer Stack
The list is: 0

The list is: 1 0

The list is: 2 1 0

2 popped from stack
The list is: 1 0

1 popped from stack
The list is: 0

0 popped from stack
The list is empty
```

```
processing a double Stack
The list is: 1.1

The list is: 2.2 1.1

The list is: 3.3 2.2 1.1

3.3 popped from stack
The list is: 2.2 1.1
```

```
2.2 popped from stack
The list is: 1.1
```

```
1.1 popped from stack
The list is empty
```

```
All nodes destroyed
```

```
All nodes destroyed
```

[Page 1020]

Another way to implement a Stack class template is by reusing the List class template through composition. [Figure 21.15](#) is a new implementation of the Stack class template that contains a `List< STACKTYPE >` object called `stackList` (line 38). This version of the Stack class template uses class List from [Fig. 21.4](#). To test this class, use the driver program in [Fig. 21.14](#), but include the new header file `Stackcomposition.h` in line 6 of that file. The output of the program is identical for both versions of class Stack.

**Figure 21.15. Stack class template with a composed List object.**

(This item is displayed on pages 1020 - 1021 in the print version)

```

1 // Fig. 21.15: Stackcomposition.h
2 // Template Stack class definition with composed List object.
3 #ifndef STACKCOMPOSITION_H
4 #define STACKCOMPOSITION_H
5
6 #include "List.h" // List class definition
7
8 template< typename STACKTYPE >
9 class Stack
10 {
11 public:
12 // no constructor; List constructor does initialization
13
14 // push calls stackList object's insertAtFront member function
15 void push(const STACKTYPE &data)
16 {
17 stackList.insertAtFront(data);

```

```
18 } // end function push
19
20 // pop calls stackList object's removeFromFront member function
21 bool pop(STACKTYPE &data)
22 {
23 return stackList.removeFromFront(data);
24 } // end function pop
25
26 // isStackEmpty calls stackList object's isEmpty member function
27 bool isStackEmpty() const
28 {
29 return stackList.isEmpty();
30 } // end function isStackEmpty
31
32 // printStack calls stackList object's print member function
33 void printStack() const
34 {
35 stackList.print();
36 } // end function printStack
37 private:
38 List< STACKTYPE > stackList; // composed List object
39 }; // end class Stack
40
41 #endif
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1022]

A file server in a computer network handles file access requests from many clients throughout the network. Servers have a limited capacity to service requests from clients. When that capacity is exceeded, client requests wait in queues.

The program of [Figs. 21.16](#)[21.17](#) creates a Queue class template (Fig. 21.16) through private inheritance (line 9) of the List class template of [Fig. 21.4](#). We want the Queue to have member functions enqueue (lines 1316), dequeue (lines 1922), isQueueEmpty (lines 2528) and printQueue (lines 3134). Note that these are essentially the insertAtBack, removeFromFront, isEmpty and print functions of the List class template. Of course, the List class template contains other member functions (i.e., insertAtFront and removeFromBack) that we would not want to make accessible through the public interface to the Queue class. So when we indicate that the Queue class template is to inherit the List class template, we specify private inheritance. This makes all the List class template's member functions private in the Queue class template. When we implement the Queue's member functions, we have each of these call the appropriate member function of the list class: enqueue calls insertAtBack (line 15), dequeue calls removeFromFront (line 21), isQueueEmpty calls isEmpty (line 27) and printQueue calls print (line 33). Again, this is called delegation.

[Page 1023]

**Figure 21.16. Queue class-template definition.**

(This item is displayed on page 1022 in the print version)

```

1 // Fig. 21.16: Queue.h
2 // Template Queue class definition derived from class List.
3 #ifndef QUEUE_H
4 #define QUEUE_H
5
6 #include "List.h" // List class definition
7
8 template< typename QUEUETYPE >
9 class Queue : private List< QUEUETYPE >
10 {
11 public:
12 // enqueue calls List member function insertAtBack
13 void enqueue(const QUEUETYPE &data)
14 {
15 insertAtBack(data);

```

```

16 } // end function enqueue
17
18 // dequeue calls List member function removeFromFront
19 bool dequeue(QUEUETYPE &data)
20 {
21 return removeFromFront(data);
22 } // end function dequeue
23
24 // isQueueEmpty calls List member function isEmpty
25 bool isQueueEmpty() const
26 {
27 return isEmpty();
28 } // end function isQueueEmpty
29
30 // printQueue calls List member function print
31 void printQueue() const
32 {
33 print();
34 } // end function printQueue
35 } // end class Queue
36
37 #endif

```

**Figure 21.17. Queue-processing program.**

(This item is displayed on pages 1023 - 1024 in the print version)

```

1 // Fig. 21.17: Fig21_17.cpp
2 // Template Queue class test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Queue.h" // Queue class definition
8
9 int main()
10 {
11 Queue< int > intQueue; // create Queue of integers
12
13 cout << "processing an integer Queue" << endl;
14
15 // enqueue integers onto intQueue
16 for (int i = 0; i < 3; i++)
17 {

```

```
18 intQueue.enqueue(i);
19 intQueue.printQueue();
20 } // end for
21
22 int dequeueInteger; // store dequeued integer
23
24 // dequeue integers from intQueue
25 while (!intQueue.isEmpty())
26 {
27 intQueue.dequeue(dequeueInteger);
28 cout << dequeueInteger << " dequeued" << endl;
29 intQueue.printQueue();
30 } // end while
31
32 Queue< double > doubleQueue; // create Queue of doubles
33 double value = 1.1;
34
35 cout << "processing a double Queue" << endl;
36
37 // enqueue floating-point values onto doubleQueue
38 for (int j = 0; j < 3; j++)
39 {
40 doubleQueue.enqueue(value);
41 doubleQueue.printQueue();
42 value += 1.1;
43 } // end for
44
45 double dequeueDouble; // store dequeued double
46
47 // dequeue floating-point values from doubleQueue
48 while (!doubleQueue.isEmpty())
49 {
50 doubleQueue.dequeue(dequeueDouble);
51 cout << dequeueDouble << " dequeued" << endl;
52 doubleQueue.printQueue();
53 } // end while
54
55 return 0;
56 } // end main
```

```
processing an integer Queue
The list is: 0

The list is: 0 1

The list is: 0 1 2

0 dequeued
The list is: 1 2

1 dequeued
The list is: 2

2 dequeued
The list is empty

processing a double Queue
The list is: 1.1

The list is: 1.1 2.2

The list is: 1.1 2.2 3.3

1.1 dequeued
The list is: 2.2 3.3

2.2 dequeued
The list is: 3.3

3.3 dequeued
The list is empty

All nodes destroyed

All nodes destroyed
```

Figure 21.17 uses the Queue class template to instantiate integer queue `intQueue` of type `Queue< int >` (line 11). Integers 0 through 2 are enqueued to `intQueue` (lines 1620), then dequeued from `intQueue` in first-in, first-out order (lines 2530). Next, the program instantiates queue `doubleQueue` of type `Queue< double >` (line 32). Values 1.1, 2.2 and 3.3 are enqueued to `doubleQueue` (lines 3843), then dequeued from `doubleQueue` in first-in, first-out order (lines 4853).

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1026]

We begin our discussion with the driver program ([Fig. 21.22](#)), then continue with the implementations of classes TReeNode ([Fig. 21.20](#)) and TRee ([Fig. 21.21](#)). Function `main` ([Fig. 21.22](#)) begins by instantiating integer tree `intTree` of type `TRee< int >` (line 15). The program prompts for 10 integers, each of which is inserted in the binary tree by calling `insertNode` (line 24). The program then performs preorder, inorder and postorder traversals (these are explained shortly) of `intTree` (lines 28, 31 and 34, respectively). The program then instantiates floating-point tree `doubleTree` of type `TRee< double >` (line 36). The program prompts for 10 `double` values, each of which is inserted in the binary tree by calling `insertNode` (line 46). The program then performs preorder, inorder and postorder traversals of `doubleTree` (lines 50, 53 and 56, respectively).

[Page 1027]

**Figure 21.21. TRee class-template definition.**

(This item is displayed on pages 1027 - 1029 in the print version)

```

1 // Fig. 21.21: Tree.h
2 // Template Tree class definition.
3 #ifndef TREE_H
4 #define TREE_H
5
6 #include <iostream>
7 using std::cout;
8 using std::endl;
9
10 #include "Treenode.h"
11
12 // Tree class-template definition
13 template< typename NODETYPE > class Tree
14 {
15 public:
16 Tree(); // constructor
17 void insertNode(const NODETYPE &);
18 void preOrderTraversal() const;
19 void inOrderTraversal() const;
20 void postOrderTraversal() const;
21 private:
22 TreeNode< NODETYPE > *rootPtr;

```

```

23
24 // utility functions
25 void insertNodeHelper(TreeNode< NODETYPE > **, const NODETYPE &);
26 void preOrderHelper(TreeNode< NODETYPE > *) const;
27 void inOrderHelper(TreeNode< NODETYPE > *) const;
28 void postOrderHelper(TreeNode< NODETYPE > *) const;
29 } // end class Tree
30
31 // constructor
32 template< typename NODETYPE >
33 Tree< NODETYPE >::Tree()
34 {
35 rootPtr = 0; // indicate tree is initially empty
36 } // end Tree constructor
37
38 // insert node in Tree
39 template< typename NODETYPE >
40 void Tree< NODETYPE >::insertNode(const NODETYPE &value)
41 {
42 insertNodeHelper(&rootPtr, value);
43 } // end function insertNode
44
45 // utility function called by insertNode; receives a pointer
46 // to a pointer so that the function can modify pointer's value
47 template< typename NODETYPE >
48 void Tree< NODETYPE >::insertNodeHelper(
49 TreeNode< NODETYPE > **ptr, const NODETYPE &value)
50 {
51 // subtree is empty; create new TreeNode containing value
52 if (*ptr == 0)
53 *ptr = new TreeNode< NODETYPE >(value);
54 else // subtree is not empty
55 {
56 // data to insert is less than data in current node
57 if (value < (*ptr)->data)
58 insertNodeHelper(&((*ptr)->leftPtr), value);
59 else
60 {
61 // data to insert is greater than data in current node
62 if (value > (*ptr)->data)
63 insertNodeHelper(&((*ptr)->rightPtr), value);
64 else // duplicate data value ignored
65 cout << value << " dup" << endl;
66 } // end else
67 } // end else
68 } // end function insertNodeHelper
69
70 // begin preorder traversal of Tree

```

```
71 template< typename NODETYPE >
72 void Tree< NODETYPE >::preOrderTraversal() const
73 {
74 preOrderHelper(rootPtr);
75 } // end function preOrderTraversal
76
77 // utility function to perform preorder traversal of Tree
78 template< typename NODETYPE >
79 void Tree< NODETYPE >::preOrderHelper(TreeNode< NODETYPE > *ptr) const
80 {
81 if (ptr != 0)
82 {
83 cout << ptr->data << ' ' ; // process node
84 preOrderHelper(ptr->leftPtr); // traverse left subtree
85 preOrderHelper(ptr->rightPtr); // traverse right subtree
86 } // end if
87 } // end function preOrderHelper
88
89 // begin inorder traversal of Tree
90 template< typename NODETYPE >
91 void Tree< NODETYPE >::inOrderTraversal() const
92 {
93 inOrderHelper(rootPtr);
94 } // end function inOrderTraversal
95
96 // utility function to perform inorder traversal of Tree
97 template< typename NODETYPE >
98 void Tree< NODETYPE >::inOrderHelper(TreeNode< NODETYPE > *ptr) const
99 {
100 if (ptr != 0)
101 {
102 inOrderHelper(ptr->leftPtr); // traverse left subtree
103 cout << ptr->data << ' ' ; // process node
104 inOrderHelper(ptr->rightPtr); // traverse right subtree
105 } // end if
106 } // end function inOrderHelper
107
108 // begin postorder traversal of Tree
109 template< typename NODETYPE >
110 void Tree< NODETYPE >::postOrderTraversal() const
111 {
112 postOrderHelper(rootPtr);
113 } // end function postOrderTraversal
114
115 // utility function to perform postorder traversal of Tree
116 template< typename NODETYPE >
117 void Tree< NODETYPE >::postOrderHelper(
118 TreeNode< NODETYPE > *ptr) const
```

```

119 {
120 if (ptr != 0)
121 {
122 postOrderHelper(ptr->leftPtr); // traverse left subtree
123 postOrderHelper(ptr->rightPtr); // traverse right subtree
124 cout << ptr->data << ' '; // process node
125 } // end if
126 } // end function postOrderHelper
127
128 #endif

```

**Figure 21.22. Creating and traversing a binary tree.**

(This item is displayed on pages 1030 - 1031 in the print version)

```

1 // Fig. 21.22: Fig21_22.cpp
2 // Tree class test program.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include "Tree.h" // Tree class definition
12
13 int main()
14 {
15 Tree< int > intTree; // create Tree of int values
16 int intValue;
17
18 cout << "Enter 10 integer values:\n";
19
20 // insert 10 integers to intTree
21 for (int i = 0; i < 10; i++)
22 {
23 cin >> intValue;
24 intTree.insertNode(intValue);
25 } // end for
26
27 cout << "\nPreorder traversal\n";
28 intTree.preOrderTraversal();
29
30 cout << "\nInorder traversal\n";

```

```

31 intTree.inOrderTraversal();
32
33 cout << "\nPostorder traversal\n";
34 intTree.postOrderTraversal();
35
36 Tree< double > doubleTree; // create Tree of double values
37 double doubleValue;
38
39 cout << fixed << setprecision(1)
40 << "\n\n\nEnter 10 double values:\n";
41
42 // insert 10 doubles to doubleTree
43 for (int j = 0; j < 10; j++)
44 {
45 cin >> doubleValue;
46 doubleTree.insertNode(doubleValue);
47 } // end for
48
49 cout << "\nPreorder traversal\n";
50 doubleTree.preOrderTraversal();
51
52 cout << "\nInorder traversal\n";
53 doubleTree.inOrderTraversal();
54
55 cout << "\nPostorder traversal\n";
56 doubleTree.postOrderTraversal();
57
58 cout << endl;
59 return 0;
60 } // end main

```

Enter 10 integer values:  
 50 25 75 12 33 67 88 6 13 68

Preorder traversal  
 50 25 12 6 13 33 75 67 68 88  
 Inorder traversal  
 6 12 13 25 33 50 67 68 75 88  
 Postorder traversal  
 6 13 12 33 25 68 67 88 75 50

Enter 10 double values:  
 39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5

Preorder traversal  
 39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5  
 Inorder traversal

```

1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5
Postorder traversal
1.1 4.4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2

```

[Page 1029]

Now we discuss the class-template definitions. We begin with the `treeNode` class template (Fig. 21.20) definition that declares `tree< NODETYPE >` as its friend (line 13). This makes all member functions of a given specialization of class template `tree` (Fig. 21.21) friends of the corresponding specialization of class template `treeNode`, so they can access the `private` members of `treeNode` objects of that type. Because the `TReeNode` template parameter `NODETYPE` is used as the template argument for `tree` in the friend declaration, `treeNodes` specialized with a particular type can be processed only by a `tree` specialized with the same type (e.g., a `TRee` of `int` values manages `TReeNode` objects that store `int` values).

Lines 3032 declare a `TReeNode`'s `private` data—the node's data value, and pointers `leftPtr` (to the node's left subtree) and `rightPtr` (to the node's right subtree). The constructor (lines 1622) sets `data` to the value supplied as a constructor argument and sets pointers `leftPtr` and `rightPtr` to zero (thus initializing this node to be a leaf node). Member function `getdata` (lines 2528) returns the data value.

[Page 1031]

The `TRee` class template (Fig. 21.21) has as `private` data `rootPtr` (line 22), a pointer to the root node of the tree. Lines 1720 of the class template declare the `public` member functions `insertNode` (that inserts a new node in the tree) and `preOrderTraversal`, `inOrderTraversal` and `postOrderTraversal`, each of which walks the tree in the designated manner. Each of these member functions calls its own separate recursive utility function to perform the appropriate operations on the internal representation of the tree, so the program is not required to access the underlying `private` data to perform these functions. Remember that the recursion requires us to pass in a pointer that represents the next subtree to process. The `tree` constructor initializes `rootPtr` to zero to indicate that the tree is initially empty.

The `tree` class's utility function `insertNodeHelper` (lines 4768) is called by `insertNode` (lines 3943) to recursively insert a node into the tree. A node can only be inserted as a leaf node in a binary search tree. If the tree is empty, a new `TReeNode` is created, initialized and inserted in the tree (lines 5354).

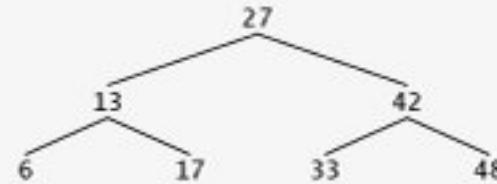
If the tree is not empty, the program compares the value to be inserted with the `data` value in the root node. If the insert value is smaller (line 57), the program recursively calls `insertNodeHelper` (line 58)

to insert the value in the left subtree. If the insert value is larger (line 62), the program recursively calls `insertNodeHelper` (line 64) to insert the value in the right subtree. If the value to be inserted is identical to the data value in the root node, the program prints the message " dup" (line 65) and returns without inserting the duplicate value into the tree. Note that `insertNode` passes the address of `rootPtr` to `insertNodeHelper` (line 42) so it can modify the value stored in `rootPtr` (i.e., the address of the root node). To receive a pointer to `rootPtr` (which is also a pointer), `insertNodeHelper`'s first argument is declared as a pointer to a pointer to a `treeNode`.

[Page 1032]

Each of the member functions `inOrderTraversal` (lines 9094), `preOrderTraversal` (lines 7175) and `postOrderTraversal` (lines 109113) traverses the tree and prints the node values. For the purpose of the following discussion, we use the binary search tree in Fig. 21.23.

**Figure 21.23. A binary search tree.**



## Inorder Traversal Algorithm

Function `inOrderTraversal` invokes utility function `inOrderHelper` to perform the inorder traversal of the binary tree. The steps for an inorder traversal are:

**1.**

Traverse the left subtree with an inorder traversal. (This is performed by the call to `inOrderHelper` at line 102.)

**2.**

Process the value in the node i.e., print the node value (line 103).

**3.**

Traverse the right subtree with an inorder traversal. (This is performed by the call to `inOrderHelper` at line 104.)

The value in a node is not processed until the values in its left subtree are processed, because each call to `inOrderHelper` immediately calls `inOrderHelper` again with the pointer to the left subtree. The inorder traversal of the tree in Fig. 21.23 is

6 13 17 27 33 42 48

Note that the inorder traversal of a binary search tree prints the node values in ascending order. The process of creating a binary search tree actually sorts the data thus, this process is called the **binary tree sort**.

## Preorder Traversal Algorithm

Function `preOrderTraversal` invokes utility function `preOrderHelper` to perform the preorder traversal of the binary tree. The steps for an preorder traversal are:

**1.**

Process the value in the node (line 83).

**2.**

Traverse the left subtree with a preorder traversal. (This is performed by the call to `preOrderHelper` at line 84.)

**3.**

Traverse the right subtree with a preorder traversal. (This is performed by the call to `preOrderHelper` at line 85.)

The value in each node is processed as the node is visited. After the value in a given node is processed, the values in the left subtree are processed. Then the values in the right subtree are processed. The preorder traversal of the tree in Fig. 21.23 is

27 13 6 17 42 33 48

## Postorder Traversal Algorithm

Function `postOrderTraversal` invokes utility function `postOrderHelper` to perform the postorder traversal of the binary tree. The steps for an postorder traversal are:

**1.**

Traverse the left subtree with a postorder traversal. (This is performed by the call to `postOrderHelper` at line 122.)

**2.**

Traverse the right subtree with a postorder traversal. (This is performed by the call to `postOrderHelper` at line 123.)

**3.**

Process the value in the node (line 124).

The value in each node is not printed until the values of its children are printed. The `postOrderTraversal` of the tree in Fig. 21.23 is

6 17 13 33 48 42 27

## Duplicate Elimination

The binary search tree facilitates **duplicate elimination**. As the tree is being created, an attempt to insert a duplicate value will be recognized, because a duplicate will follow the same "go left" or "go right" decisions on each comparison as the original value did when it was inserted in the tree. Thus, the duplicate will eventually be compared with a node containing the same value. The duplicate value may be discarded at this point.

Searching a binary tree for a value that matches a key value is also fast. If the tree is balanced, then each branch contains about half the number of nodes in the tree. Each comparison of a node to the search key eliminates half the nodes. This is called an O ( $\log n$ ) algorithm (Big O notation is discussed in Chapter 20). So a binary search tree with  $n$  elements would require a maximum of  $\log_2 n$  comparisons either to find a match or to determine that no match exists. This means, for example, that when searching a (balanced) 1000-element binary search tree, no more than 10 comparisons need to be made, because  $2^{10} > 1000$ . When searching a (balanced) 1,000,000-element binary search tree, no more than 20 comparisons need to be made, because  $2^{20} > 1,000,000$ .

## Overview of the Binary Tree Exercises

In the exercises, algorithms are presented for several other binary tree operations such as deleting an item from a binary tree, printing a binary tree in a two-dimensional tree format and performing a level-order traversal of a binary tree. The level-order traversal of a binary tree visits the nodes of the tree row by row, starting at the root node level. On each level of the tree, the nodes are visited from left to right.

Other binary tree exercises include allowing a binary search tree to contain duplicate values, inserting string values in a binary tree and determining how many levels are contained in a binary tree.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1034]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1035]

- A stack is referred to as a last-in, first-out (LIFO) data structure.
- The primary member functions used to manipulate a stack are `push` and `pop`. Function `push` inserts a new node at the top of the stack. Function `pop` removes a node from the top of the stack.
- A queue is similar to a supermarket checkout line—the first person in line is serviced first, and other customers enter the line at the end and wait to be serviced.
- Queue nodes are removed only from the head of the queue and are inserted only at the tail of the queue.
- A queue is referred to as a first-in, first-out (FIFO) data structure. The insert and remove operations are known as `enqueue` and `dequeue`.
- Binary trees are trees whose nodes all contain two links (none, one or both of which may be null).
- The root node is the first node in a tree.
- Each link in the root node refers to a child. The left child is the root node of the left subtree, and the right child is the root node of the right subtree.
- The children of a single node are called *siblings*. A node with no children is called a *leaf node*.
- A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its parent node, and the values in any right subtree are greater than the value in its parent node.
- A node can only be inserted as a leaf node in a binary search tree.
- An inorder traversal of a binary tree traverses the left subtree inorder, processes the value in the root node and then traverses the right subtree inorder. The value in a node is not processed until the values in its left subtree are processed.
- A preorder traversal processes the value in the root node, traverses the left subtree preorder, then traverses the right subtree preorder. The value in each node is processed as the node is encountered.
- A postorder traversal traverses the left subtree postorder, traverses the right subtree postorder, then processes the value in the root node. The value in each node is not processed until the values in both its subtrees are processed.
- The binary search tree facilitates duplicate elimination. As the tree is being created, an attempt to insert a duplicate value will be recognized and the duplicate value may be discarded.
- The level-order traversal of a binary tree visits the nodes of the tree row by row, starting at the root node level. On each level of the tree, the nodes are visited from left to right.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1036]

pointer link

pop

postorder traversal of a binary tree

preorder traversal of a binary tree

print spooling

push

queue

right child

right subtree

root node

self-referential structure

siblings

singly linked list

spooler

stack

tail of a queue

top of a stack



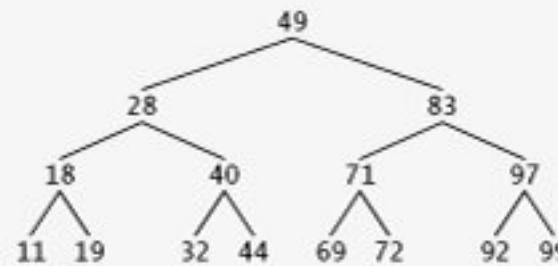
Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1037]

## 21.5

Manually provide the inorder, preorder and postorder traversals of the binary search tree of [Fig. 21.24](#).

**Figure 21.24. A 15-node binary search tree.**



[◀ PREV](#)

[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1037 (continued)]

## Answers to Self-Review Exercises

- 21.1** a) referential. b) new. c) stack. d) predicate. e) first-in, first-out (FIFO). f) link. g) delete. h) queue. i) tree. j) last-in, first-out (LIFO). k) binary. l) root. m) child or subtree. n) leaf. o) inorder, preorder, postorder and level order.
- 21.2** It is possible to insert a node anywhere in a linked list and remove a node from anywhere in a linked list. Nodes in a stack may only be inserted at the top of the stack and removed from the top of a stack.
- 21.3** A queue data structure allows nodes to be removed only from the head of the queue and inserted only at the tail of the queue. A queue is referred to as a first-in, first-out (FIFO) data structure. A stack data structure allows nodes to be added to the stack and removed from the stack only at the top. A stack is referred to as a last-in, first-out (LIFO) data structure.
- 21.4**
- Classes allow us to instantiate as many data structure objects of a certain type (i.e., class) as we wish.
  - Class templates enable us to instantiate related classes, each based on different type parameters we can then generate as many objects of each template class as we like.
  - Inheritance enables us to reuse code from a base class in a derived class, so that the derived-class data structure is also a base-class data structure (with public inheritance, that is).

Private inheritance enables us to reuse portions of the code from a base class to form a derived-class data structure; because the inheritance is `private`, all `public` base-class member functions become `private` in the derived class. This enables us to prevent clients of the derived-class data structure from accessing base-class member functions that do not apply to the derived class.

e.

Composition enables us to reuse code by making a class object data structure a member of a composed class; if we make the class object a `private` member of the composed class, then the class object's `public` member functions are not available through the composed object's interface.

## 21.5 The inorder traversal is

11 18 19 28 32 40 44 49 69 71 72 83 92 97 99

The preorder traversal is

49 28 18 11 19 40 32 44 83 71 69 72 97 92 99

The postorder traversal is

11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1039]

•

While the stack is not empty, read `infix` from left to right and do the following:

If the current character in `infix` is a digit, copy it to the next element of `postfix`.

If the current character in `infix` is a left parenthesis, push it onto the stack.

If the current character in `infix` is an operator,

Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and insert the popped operators in `postfix`.

Push the current character in `infix` onto the stack.

If the current character in `infix` is a right parenthesis

Pop operators from the top of the stack and insert them in `postfix` until a left parenthesis is at the top of the stack.

Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:

+ addition

- subtraction

\* multiplication

/ division

^ exponentiation

% modulus

[Note: We assume left to right associativity for all operators for the purpose of this exercise.] The stack should be maintained with stack nodes, each containing a data member and a pointer to the next stack node.

Some of the functional capabilities you may want to provide are:

a.

function `convertToPostfix` that converts the infix expression to postfix notation

b.

function `isOperator` that determines whether `c` is an operator

c.

function `precedence` that determines whether the precedence of `operator1` is less than, equal to or greater than the precedence of `operator2` (the function returns -1, 0 and 1, respectively)

d.

function `push` that pushes a value onto the stack

e.

function `pop` that pops a value off the stack

f.

function `stackTop` that returns the top value of the stack without popping the stack

g.

function `isEmpty` that determines if the stack is empty

h.

function `printStack` that prints the stack

## 21.13

Write a program that evaluates a postfix expression (assume it is valid) such as

6 2 + 5 \* 8 4 / -

The program should read a postfix expression consisting of digits and operators into a character array. Using modified versions of the stack functions implemented earlier in this chapter, the program should scan the expression and evaluate it. The algorithm is as follows:

**1.**

Append the null character ('\0') to the end of the postfix expression. When the null character is encountered, no further processing is necessary.

**2.**

While '\0' has not been encountered, read the expression from left to right.

If the current character is a digit,

Push its integer value onto the stack (the integer value of a digit character is its value in the computer's character set minus the value of '0' in the computer's character set).

Otherwise, if the current character is an operator,

Pop the two top elements of the stack into variables  $x$  and  $y$ .

Calculate  $y$  operator  $x$ .

Push the result of the calculation onto the stack.

---

[Page 1040]

**3.**

When the null character is encountered in the expression, pop the top value of the stack. This is the result of the postfix expression.

[Note: In Step 2 above, if the operator is '/', the top of the stack is 2 and the next element in the stack is 8, then pop 2 into  $x$ , pop 8 into  $y$ , evaluate  $8 / 2$  and push the result, 4, back onto the stack. This note also applies to operator '-'.] The arithmetic operations allowed in an expression are

+ addition

subtraction

\* multiplication

/ division

^ exponentiation

% modulus

[Note: We assume left to right associativity for all operators for the purpose of this exercise.] The stack should be maintained with stack nodes that contain an `int` data member and a pointer to the next stack node. You may want to provide the following functional capabilities:

a.

function `evaluatePostfixExpression` that evaluates the postfix expression

b.

function `calculate` that evaluates the expression `op1 operator op2`

c.

function `push` that pushes a value onto the stack

d.

function `pop` that pops a value off the stack

e.

function `isEmpty` that determines if the stack is empty

f.

function `printStack` that prints the stack

## 21.14

Modify the postfix evaluator program of [Exercise 21.13](#) so that it can process integer operands larger than 9.

## 21.15

(Supermarket Simulation) Write a program that simulates a checkout line at a supermarket. The line is a queue object. Customers (i.e., customer objects) arrive in random integer intervals of 14 minutes. Also, each customer is served in random integer intervals of 14 minutes. Obviously, the rates need to be balanced. If the average arrival rate is larger than the average service rate, the queue will grow infinitely. Even with "balanced" rates, randomness can still cause long lines. Run the supermarket simulation for a 12-hour day (720 minutes) using the following algorithm:

1.

Choose a random integer between 1 and 4 to determine the minute at which the first customer arrives.

2.

At the first customer's arrival time:

Determine customer's service time (random integer from 1 to 4);

Begin servicing the customer;

Schedule arrival time of next customer (random integer 1 to 4 added to the current time).

3.

For each minute of the day:

If the next customer arrives,

Say so,

Enqueue the customer;

Schedule the arrival time of the next customer;

If service was completed for the last customer;

Say so

Dequeue next customer to be serviced

Determine customer's service completion time (random integer from

1 to 4 added to the current time).

Now run your simulation for 720 minutes, and answer each of the following:

a.

What is the maximum number of customers in the queue at any time?

b.

What is the longest wait any one customer experiences?

c.

What happens if the arrival interval is changed from 14 minutes to 13 minutes?

---

[Page 1041]

## 21.16

Modify the program of [Figs. 21.20 21.22](#) to allow the binary tree object to contain duplicates.

## 21.17

Write a program based on [Figs. 21.20 21.22](#) that inputs a line of text, tokenizes the sentence into separate words (you may want to use the `strtok` library function), inserts the words in a binary search tree and prints the inorder, preorder and postorder traversals of the tree. Use an OOP approach.

## 21.18

In this chapter, we saw that duplicate elimination is straightforward when creating a binary search tree. Describe how you would perform duplicate elimination using only a one-dimensional array. Compare the performance of array-based duplicate elimination with the performance of binary-search-tree-based duplicate elimination.

## 21.19

Write a function `depth` that receives a binary tree and determines how many levels it has.

## 21.20

(Recursively Print a List Backward) Write a member function `printListBackward` that recursively outputs the items in a linked list object in reverse order. Write a test program that creates a sorted list of

integers and prints the list in reverse order.

## 21.21

(Recursively Search a List) Write a member function `searchList` that recursively searches a linked list object for a specified value. The function should return a pointer to the value if it is found; otherwise, `null` should be returned. Use your function in a test program that creates a list of integers. The program should prompt the user for a value to locate in the list.

## 21.22

(Binary Tree Delete) In this exercise, we discuss deleting items from binary search trees. The deletion algorithm is not as straightforward as the insertion algorithm. There are three cases that are encountered when deleting an item: the item is contained in a leaf node (i.e., it has no children), the item is contained in a node that has one child or the item is contained in a node that has two children.

If the item to be deleted is contained in a leaf node, the node is deleted and the pointer in the parent node is set to `null`.

If the item to be deleted is contained in a node with one child, the pointer in the parent node is set to point to the child node and the node containing the data item is deleted. This causes the child node to take the place of the deleted node in the tree.

The last case is the most difficult. When a node with two children is deleted, another node in the tree must take its place. However, the pointer in the parent node cannot be assigned to point to one of the children of the node to be deleted. In most cases, the resulting binary search tree would not adhere to the following characteristic of binary search trees (with no duplicate values): The values in any left subtree are less than the value in the parent node, and the values in any right subtree are greater than the value in the parent node.

Which node is used as a replacement node to maintain this characteristic? Either the node containing the largest value in the tree less than the value in the node being deleted, or the node containing the smallest value in the tree greater than the value in the node being deleted. Let us consider the node with the smaller value. In a binary search tree, the largest value less than a parent's value is located in the left subtree of the parent node and is guaranteed to be contained in the rightmost node of the subtree. This node is located by walking down the left subtree to the right until the pointer to the right child of the current node is `null`. We are now pointing to the replacement node, which is either a leaf node or a node with one child to its left. If the replacement node is a leaf node, the steps to perform the deletion are as follows:

1.

Store the pointer to the node to be deleted in a temporary pointer variable (this pointer is used to delete the dynamically allocated memory).

Set the pointer in the parent of the node being deleted to point to the replacement node.

---

[Page 1042]

3.

Set the pointer in the parent of the replacement node to null.

4.

Set the pointer to the right subtree in the replacement node to point to the right subtree of the node to be deleted.

5.

Delete the node to which the temporary pointer variable points.

The deletion steps for a replacement node with a left child are similar to those for a replacement node with no children, but the algorithm also must move the child into the replacement node's position in the tree. If the replacement node is a node with a left child, the steps to perform the deletion are as follows:

1.

Store the pointer to the node to be deleted in a temporary pointer variable.

2.

Set the pointer in the parent of the node being deleted to point to the replacement node.

3.

Set the pointer in the parent of the replacement node to point to the left child of the replacement node.

4.

Set the pointer to the right subtree in the replacement node to point to the right subtree of the node to be deleted.

5.

Delete the node to which the temporary pointer variable points.

Write member function `deleteNode`, which takes as its arguments a pointer to the root node of the tree object and the value to be deleted. The function should locate in the tree the node containing the value to be deleted and use the algorithms discussed here to delete the node. The function should print a message that indicates whether the value is deleted. Modify the program of Figs. 21.2021.22 to use this function. After deleting an item, call the `inOrder`, `preOrder` and `postOrder` TRaversals functions to confirm that the delete operation was performed correctly.

## 21.23

(Binary Tree Search) Write member function `binaryTreeSearch`, which attempts to locate a specified value in a binary search tree object. The function should take as arguments a pointer to the root node of the binary tree and a search key to be located. If the node containing the search key is found, the function should return a pointer to that node; otherwise, the function should return a null pointer.

## 21.24

(Level-Order Binary Tree Traversal) The program of Figs. 21.2021.22 illustrated three recursive methods of traversing a binary tree: `inorder`, `preorder` and `postorder` traversals. This exercise presents the level-order traversal of a binary tree, in which the node values are printed level by level, starting at the root node level. The nodes on each level are printed from left to right. The `levelorder` traversal is not a recursive algorithm. It uses a queue object to control the output of the nodes. The algorithm is as follows:

1.

Insert the root node in the queue

2.

While there are nodes left in the queue,

    Get the next node in the queue

    Print the node's value

    If the pointer to the left child of the node is not null

        Insert the left child node in the queue

    If the pointer to the right child of the node is not null

        Insert the right child node in the queue.

Write member function `levelOrder` to perform a level-order traversal of a binary tree object. Modify the program of Figs. 21.2021.22 to use this function. [Note: You will also need to modify and incorporate the

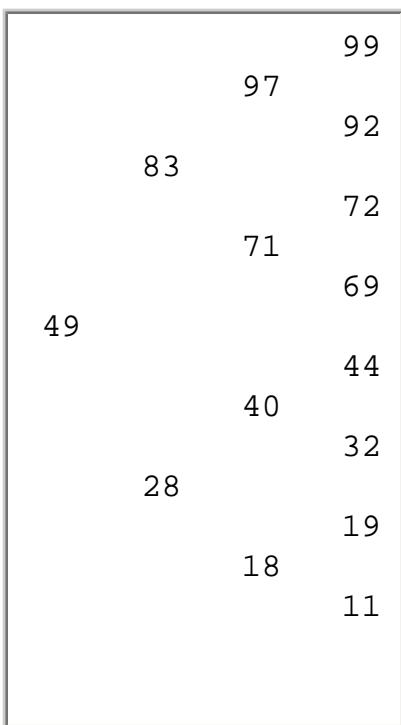
queue-processing functions of Fig. 21.16 in this program.]

## 21.25

(Printing Trees) Write a recursive member function `outputTree` to display a binary tree object on the screen. The function should output the tree row by row, with the top of the tree at the left of the screen and the bottom of the tree toward the right of the screen. Each row is output vertically. For example, the binary tree illustrated in Fig. 21.24 is output as follows:

---

[Page 1043]



Note that the rightmost leaf node appears at the top of the output in the rightmost column and the root node appears at the left of the output. Each column of output starts five spaces to the right of the previous column. Function `outputTree` should receive an argument `totalSpaces` representing the number of spaces preceding the value to be output (this variable should start at zero, so the root node is output at the left of the screen). The function uses a modified inorder traversal to output the tree it starts at the rightmost node in the tree and works back to the left. The algorithm is as follows:

While the pointer to the current node is not null

    Recursively call `outputTree` with the right subtree of the current node and  
    `totalSpaces + 5`

    Use a for structure to count from 1 to `totalSpaces` and output spaces

Output the value in the current node

Set the pointer to the current node to point to the left subtree of the current node

Increment `totalSpaces` by 5.

 PREV

page footer

NEXT 

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1044]

Simple uses the conditional `if...goto` statement and the unconditional `goto` statement to alter the flow of control during program execution. If the condition in the `if...goto` statement is true, control is transferred to a specific line of the program. The following relational and equality operators are valid in an `if...goto` statement: `<`, `>`, `<=`, `>=`, `==` and `!=`. The precedence of these operators is the same as in C++.

Let us now consider several programs that demonstrate Simple's features. The first program ([Fig. 21.26](#)) reads two integers from the keyboard, stores the values in variables `a` and `b` and computes and prints their sum (stored in variable `c`).

**Figure 21.26. Simple program that determines the sum of two integers.**

(This item is displayed on page 1045 in the print version)

```

1 10 rem determine and print the sum of two integers
2 15 rem
3 20 rem input the two integers
4 30 input a
5 40 input b
6 45 rem
7 50 rem add integers and store result in c
8 60 let c = a + b
9 65 rem
10 70 rem print the result
11 80 print c
12 90 rem terminate program execution
13 99 end

```

The program of [Fig. 21.27](#) determines and prints the larger of two integers. The integers are input from the keyboard and stored in `s` and `t`. The `if...goto` statement tests the condition `s >= t`. If the condition is true, control is transferred to line 90 and `s` is output; otherwise, `t` is output and control is transferred to the `end` statement in line 99, where the program terminates.

**Figure 21.27. Simple program that finds the larger of two integers.**

(This item is displayed on page 1045 in the print version)

```

1 10 rem determine the larger of two integers
2 20 input s
3 30 input t
4 32 rem
5 35 rem test if s >= t
6 40 if s >= t goto 90
7 45 rem
8 50 rem t is greater than s, so print t
9 60 print t
10 70 goto 99
11 75 rem
12 80 rem s is greater than or equal to t, so print s
13 90 print s
14 99 end

```

Simple does not provide a repetition statement (such as C++'s `for`, `while` or `do...while`). However, Simple can simulate each of C++'s repetition statements using the `if...goto` and `goto` statements.

[Figure 21.28](#) uses a sentinel-controlled loop to calculate the squares of several integers. Each integer is input from the keyboard and stored in variable `j`. If the value entered is the sentinel value - 9999, control is transferred to line 99, where the program terminates. Otherwise, `k` is assigned the square of `j`, `k` is output to the screen and control is passed to line 20, where the next integer is input.

### Figure 21.28. Calculate the squares of several integers.

(This item is displayed on pages 1045 - 1046 in the print version)

```

1 10 rem calculate the squares of several integers
2 20 input j
3 23 rem
4 25 rem test for sentinel value
5 30 if j == -9999 goto 99
6 33 rem
7 35 rem calculate square of j and assign result to k
8 40 let k = j * j
9 50 print k
10 53 rem
11 55 rem loop to get next j
12 60 goto 20
13 99 end

```

---

[Page 1045]

Using the sample programs of [Fig. 21.26](#), [Fig. 21.27](#) and [Fig. 21.28](#) as your guide, write a Simple program to accomplish each of the following:

**a.**

Input three integers, determine their average and print the result.

**b.**

Use a sentinel-controlled loop to input 10 integers and compute and print their sum.

**c.**

Use a counter-controlled loop to input seven integers, some positive and some negative, and compute and print their average.

**d.**

Input a series of integers and determine and print the largest. The first integer input indicates how many numbers should be processed.

**e.**

Input 10 integers and print the smallest.

**f.**

Calculate and print the sum of the even integers from 2 to 30.

**g.**

Calculate and print the product of the odd integers from 1 to 9.

---

[Page 1046]

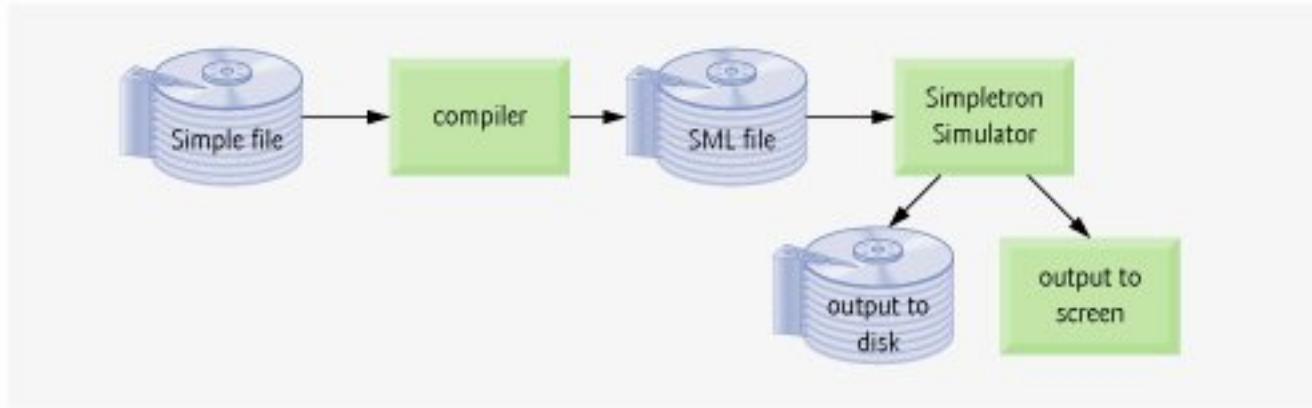
## 21.27

(Building a Compiler; Prerequisite: Complete [Exercises 8.18, 8.19, 21.12, 21.13](#) and [21.26](#)) Now that the Simple language has been presented ([Exercise 21.26](#)), we discuss how to build a Simple compiler. First, we consider the process by which a Simple program is converted to SML and executed by the Simpletron

simulator (see Fig. 21.29). A file containing a Simple program is read by the compiler and converted to SML code. The SML code is output to a file on disk, in which SML instructions appear one per line. The SML file is then loaded into the Simpletron simulator, and the results are sent to a file on disk and to the screen. Note that the Simpletron program developed in Exercise 8.19 took its input from the keyboard. It must be modified to read from a file so it can run the programs produced by our compiler.

**Figure 21.29. Writing, compiling and executing a Simple language program.**

[View full size image]



The Simple compiler performs two passes of the Simple program to convert it to SML. The first pass constructs a symbol table (object) in which every line number (object), variable name (object) and constant (object) of the Simple program is stored with its type and corresponding location in the final SML code (the symbol table is discussed in detail below). The first pass also produces the corresponding SML instruction object(s) for each of the Simple statements (object, etc.). As we will see, if the Simple program contains statements that transfer control to a line later in the program, the first pass results in an SML program containing some "unfinished" instructions. The second pass of the compiler locates and completes the unfinished instructions, and outputs the SML program to a file.

### First Pass

The compiler begins by reading one statement of the Simple program into memory. The line must be separated into its individual tokens (i.e., "pieces" of a statement) for processing and compilation (standard library function `strtok` can be used to facilitate this task). Recall that every statement begins with a line number followed by a command. As the compiler breaks a statement into tokens, if the token is a line number, a variable or a constant, it is placed in the symbol table. A line number is placed in the symbol table only if it is the first token in a statement. The `symbolTable` object is an array of `tableEntry` objects representing each symbol in the program. There is no restriction on the number of symbols that can appear in the program. Therefore, the `symbolTable` for a particular program could be large. Make the `symbolTable` a 100-element array for now. You can increase or decrease its size once the program is working.

Each `tableEntry` object contains three members. Member `symbol` is an integer containing the ASCII representation of a variable (remember that variable names are single characters), a line number or a constant. Member `type` is one of the following characters indicating the symbol's type: 'C' for constant, 'L' for line number and 'V' for variable. Member `location` contains the Simpletron memory location (00 to 99) to which the symbol refers. Simpletron memory is an array of 100 integers in which SML instructions and data are stored. For a line number, the location is the element in the Simpletron memory array at which the SML instructions for the Simple statement begin. For a variable or constant, the location is the element in the Simpletron memory array in which the variable or constant is stored. Variables and constants are allocated from the end of Simpletron's memory backward. The first variable or constant is stored in location at 99, the next in location at 98, etc.

The symbol table plays an integral part in converting Simple programs to SML. We learned in [Chapter 8](#) that an SML instruction is a four-digit integer composed of two parts—the operation code and the operand. The operation code is determined by commands in Simple. For example, the simple command `input` corresponds to SML operation code 10 (read), and the Simple command `print` corresponds to SML operation code 11 (write). The operand is a memory location containing the data on which the operation code performs its task (e.g., operation code 10 reads a value from the keyboard and stores it in the memory location specified by the operand). The compiler searches `symbolTable` to determine the Simpletron memory location for each symbol so the corresponding location can be used to complete the SML instructions.

The compilation of each Simple statement is based on its command. For example, after the line number in a `rem` statement is inserted in the symbol table, the remainder of the statement is ignored by the compiler because a remark is for documentation purposes only. The `input`, `print`, `goto` and `end` statements correspond to the SML `read`, `write`, `branch` (to a specific location) and `halt` instructions. Statements containing these Simple commands are converted directly to SML (note that a `goto` statement may contain an unresolved reference if the specified line number refers to a statement further into the Simple program file; this is sometimes called a forward reference).

When a `goto` statement is compiled with an unresolved reference, the SML instruction must be flagged to indicate that the second pass of the compiler must complete the instruction. The flags are stored in 100-element array `flags` of type `int` in which each element is initialized to -1. If the memory location to which a line number in the Simple program refers is not yet known (i.e., it is not in the symbol table), the line number is stored in array `flags` in the element with the same subscript as the incomplete instruction. The operand of the incomplete instruction is set to 00 temporarily. For example, an unconditional branch instruction (making a forward reference) is left as +4000 until the second pass of the compiler. The second pass of the compiler is described shortly.

Compilation of `if...goto` and `let` statements is more complicated than for other statements—they are the only statements that produce more than one SML instruction. For an `if...goto`, the compiler produces code to test the condition and to branch to another line if necessary. The result of the branch could be an unresolved reference. Each of the relational and equality operators can be simulated using SML's `branch zero` or `branch negative` instructions (or a combination of both).

For a `let` statement, the compiler produces code to evaluate an arbitrarily complex arithmetic expression

consisting of integer variables and/or constants. Expressions should separate each operand and operator with spaces. [Exercise 21.12](#) and [Exercise 21.13](#) presented the infix-to-postfix conversion algorithm and the postfix evaluation algorithm used by compilers to evaluate expressions. Before proceeding with your compiler, you should complete each of these exercises. When a compiler encounters an expression, it converts the expression from infix notation to postfix notation and then evaluates the postfix expression.

How is it that the compiler produces the machine language to evaluate an expression containing variables? The postfix evaluation algorithm contains a "hook" where the compiler can generate SML instructions rather than actually evaluating the expression. To enable this "hook" in the compiler, the postfix evaluation algorithm must be modified to search the symbol table for each symbol it encounters (and possibly insert it), determine the symbol's corresponding memory location and push the memory location onto the stack (instead of the symbol). When an operator is encountered in the postfix expression, the two memory locations at the top of the stack are popped and machine language for effecting the operation is produced, using the memory locations as operands. The result of each subexpression is stored in a temporary location in memory and pushed back onto the stack so that the evaluation of the postfix expression can continue. When postfix evaluation is complete, the memory location containing the result is the only location left on the stack. This is popped, and SML instructions are generated to assign the result to the variable at the left of the let statement.

---

[Page 1048]

## Second Pass

The second pass of the compiler performs two tasks: Resolve any unresolved references, and output the SML code to a file. Resolution of references occurs as follows:

a.

Search the `flags` array for an unresolved reference (i.e., an element with a value other than -1).

b.

Locate the object in array `symbolTable`, containing the symbol stored in the `flags` array (be sure that the type of the symbol is '`L`' for line number).

c.

Insert the memory location from member `location` into the instruction with the unresolved reference (remember that an instruction containing an unresolved reference has operand 00).

d.

Repeat Steps 1, 2 and 3 until the end of the `flags` array is reached.

After the resolution process is complete, the entire array containing the SML code is output to a disk file with one SML instruction per line. This file can be read by the Simpletron for execution (after the simulator is modified to read its input from a file). Compiling your first Simple program into an SML file and then executing that file should give you a real sense of personal accomplishment.

## A Complete Example

The following example illustrates a complete conversion of a Simple program to SML as it will be performed by the Simple compiler. Consider a Simple program that inputs an integer and sums the values from 1 to that integer. The program and the SML instructions produced by the first pass of the Simple compiler are illustrated in Fig. 21.30. The symbol table constructed by the first pass is shown in Fig. 21.31.

**Figure 21.30. SML instructions produced after the compiler's first pass.**

[Page 1049]

Simple program	SML location & instruction	Description
5 rem sum 1 to x	none	rem ignored
10 input x	00 +1099	read x into location 99
15 rem check y == x	none	rem ignored
20 if y == x goto 60	01 +2098	load y (98) into accumulator
	02 +3199	sub x (99) from accumulator
	03 +4200	branch zero to unresolved location
25 rem increment y	none	rem ignored
30 let y = y + 1	04 +2098	load y into accumulator
	05 +3097	add 1 (97) to accumulator
	06 +2196	store in temporary location 96
	07 +2096	load from temporary location 96
	08 +2198	store accumulator in y
35 rem add y to total	none	rem ignored
40 let t = t + y	09 +2095	load t (95) into accumulator

	10 +3098	add y to accumulator
	11 +2194	store in temporary location 94
	12 +2094	load from temporary location 94
	13 +2195	store accumulator in t
45 rem loop y	none	rem ignored
50 goto 20	14 +4001	branch to location 01
55 rem output result	none	rem ignored
60 print t	15 +1195	output t to screen
99 end	16 +4300	terminate execution

---

[Page 1049]

**Figure 21.31. Symbol table for program of Fig. 21.30.**

Symbol	Type	Location
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09

40	L	09
't'	V	95
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

[Page 1050]

Most Simple statements convert directly to single SML instructions. The exceptions in this program are remarks, the `if...goto` statement in line 20 and the `let` statements. Remarks do not translate into machine language. However, the line number for a remark is placed in the symbol table in case the line number is referenced in a `goto` statement or an `if...goto` statement. Line 20 of the program specifies that if the condition `y == x` is true, program control is transferred to line 60. Because line 60 appears later in the program, the first pass of the compiler has not as yet placed 60 in the symbol table (statement line numbers are placed in the symbol table only when they appear as the first token in a statement). Therefore, it is not possible at this time to determine the operand of the SML branch zero instruction at location 03 in the array of SML instructions. The compiler places 60 in location 03 of the `flags` array to indicate that the second pass completes this instruction.

We must keep track of the next instruction location in the SML array, because there is not a one-to-one correspondence between Simple statements and SML instructions. For example, the `if...goto` statement of line 20 compiles into three SML instructions. Each time an instruction is produced, we must increment the instruction counter to the next location in the SML array. Note that the size of Simpletron's memory could present a problem for Simple programs with many statements, variables and constants. It is conceivable that the compiler will run out of memory. To test for this case, your program should contain a data counter to keep track of the location at which the next variable or constant will be stored in the SML array. If the value of the instruction counter is larger than the value of the data counter, the SML array is full. In this case, the compilation process should terminate and the compiler should print an error message indicating that it ran out of memory during compilation. This serves to emphasize that, although the programmer is freed from the burdens of managing memory by the compiler, the compiler itself must carefully determine the placement of instructions and data in memory, and must check for such errors as memory being exhausted during the compilation process.

### A Step-by-Step View of the Compilation Process

Let us now walk through the compilation process for the Simple program in Fig. 21.30. The compiler

reads the first line of the program

```
5 rem sum 1 to x
```

into memory. The first token in the statement (the line number) is determined using `strtok` (see [Chapter 8](#) and [Chapter 21](#) for a discussion of C++'s C-style string-manipulation functions). The token returned by `strtok` is converted to an integer using `atoi`, so the symbol 5 can be located in the symbol table. If the symbol is not found, it is inserted in the symbol table. Since we are at the beginning of the program and this is the first line, no symbols are in the table yet. So 5 is inserted into the symbol table as type `L` (line number) and assigned the first location in SML array (00). Although this line is a remark, a space in the symbol table is still allocated for the line number (in case it is referenced by a `goto` or an `if...goto`). No SML instruction is generated for a `rem` statement, so the instruction counter is not incremented.

The statement

```
10 input x
```

is tokenized next. The line number 10 is placed in the symbol table as type `L` and assigned the first location in the SML array (00, because a remark began the program so the instruction counter is currently 00). The command `input` indicates that the next token is a variable (only a variable can appear in an `input` statement). Because `input` corresponds directly to an SML operation code, the compiler has to determine the location of `x` in the SML array. Symbol `x` is not found in the symbol table, so it is inserted into the symbol table as the ASCII representation of `x`, given type `V`, and assigned location 99 in the SML array (data storage begins at 99 and is allocated backward). SML code can now be generated for this statement. Operation code 10 (the SML read operation code) is multiplied by 100, and the location of `x` (as determined in the symbol table) is added to complete the instruction. The instruction is then stored in the SML array at location 00. The instruction counter is incremented by 1, because a single SML instruction was produced.

[Page 1051]

The statement

```
15 rem check y == x
```

is tokenized next. The symbol table is searched for line number 15 (which is not found). The line number is inserted as type `L` and assigned the next location in the array, 01 (remember that `rem` statements do not produce code, so the instruction counter is not incremented).

The statement

```
20 if y == x goto 60
```

is tokenized next. Line number 20 is inserted in the symbol table and given type L with the next location in the SML array 01. The command `if` indicates that a condition is to be evaluated. The variable `y` is not found in the symbol table, so it is inserted and given the type v and the SML location 98. Next, SML instructions are generated to evaluate the condition. Since there is no direct equivalent in SML for the `if...goto`, it must be simulated by performing a calculation using `x` and `y` and branching based on the result. If `y` is equal to `x`, the result of subtracting `x` from `y` is zero, so the branch zero instruction can be used with the result of the calculation to simulate the `if...goto` statement. The first step requires that `y` be loaded (from SML location 98) into the accumulator. This produces the instruction 01 +2098. Next, `x` is subtracted from the accumulator. This produces the instruction 02 +3199. The value in the accumulator may be zero, positive or negative. Since the operator is `==`, we want to branch zero. First, the symbol table is searched for the branch location (60 in this case), which is not found. So 60 is placed in the flags array at location 03, and the instruction 03 +4200 is generated (we cannot add the branch location, because we have not assigned a location to line 60 in the SML array yet). The instruction counter is incremented to 04.

The compiler proceeds to the statement

```
25 rem increment y
```

The line number 25 is inserted in the symbol table as type L and assigned SML location 04. The instruction counter is not incremented.

When the statement

```
30 let y = y + 1
```

is tokenized, the line number 30 is inserted in the symbol table as type L and assigned SML location 04. Command `let` indicates that the line is an assignment statement. First, all the symbols on the line are inserted in the symbol table (if they are not already there). The integer 1 is added to the symbol table as type c and assigned SML location 97. Next, the right side of the assignment is converted from infix to postfix notation. Then the postfix expression (`y 1 +`) is evaluated. Symbol `y` is located in the symbol table, and its corresponding memory location is pushed onto the stack. Symbol 1 is also located in the symbol table, and its corresponding memory location is pushed onto the stack. When the operator `+` is encountered, the postfix evaluator pops the stack into the right operand of the operator, pops the stack again into the left operand of the operator and produces the SML instructions

```
04 +2098 (load y)
```

```
05 +3097 (add 1)
```

The result of the expression is stored in a temporary location in memory (96) with instruction

[Page 1052]

06 +2196 (store temporary)

and the temporary location is pushed on the stack. Now that the expression has been evaluated, the result must be stored in *y* (i.e., the variable on the left side of =). So the temporary location is loaded into the accumulator, and the accumulator is stored in *y* with the instructions

07 +2096 (load temporary)

08 +2198 (store *y*)

The reader will immediately notice that SML instructions appear to be redundant. We will discuss this issue shortly.

When the statement

35 rem add *y* to total

is tokenized, line number 35 is inserted in the symbol table as type L and assigned location 09.

The statement

40 let *t* = *t* + *y*

is similar to line 30. The variable *t* is inserted in the symbol table as type v and assigned SML location 95. The instructions follow the same logic and format as line 30, and the instructions 09 +2095, 10 +3098, 11 +2194, 12 +2094 and 13 +2195 are generated. Note that the result of *t* + *y* is assigned to temporary location 94 before being assigned to *t* (95). Once again, the reader will note that the instructions in memory locations 11 and 12 appear to be redundant. Again, we will discuss this shortly.

The statement

45 rem loop *y*

is a remark, so line 45 is added to the symbol table as type `L` and assigned SML location 14.

The statement

```
50 goto 20
```

transfers control to line 20. Line number 50 is inserted in the symbol table as type `L` and assigned SML location 14. The equivalent of `goto` in SML is the unconditional branch (40) instruction that transfers control to a specific SML location. The compiler searches the symbol table for line 20 and finds that it corresponds to SML location 01. The operation code (40) is multiplied by 100, and location 01 is added to it to produce the instruction `14 +4001`.

The statement

```
55 rem output result
```

is a remark, so line 55 is inserted in the symbol table as type `L` and assigned SML location 15.

The statement

```
60 print t
```

is an output statement. Line number 60 is inserted in the symbol table as type `L` and assigned SML location 15. The equivalent of `print` in SML is operation code 11 (write). The location of `t` is determined from the symbol table and added to the result of the operation code multiplied by 100.

The statement

```
99 end
```

[Page 1053]

is the final line of the program. Line number 99 is stored in the symbol table as type `L` and assigned SML location 16. The `end` command produces the SML instruction `+4300` (43 is halt in SML), which is written as the final instruction in the SML memory array.

This completes the first pass of the compiler. We now consider the second pass. The `flags` array is searched for values other than -1. Location 03 contains 60, so the compiler knows that instruction 03 is

incomplete. The compiler completes the instruction by searching the symbol table for 60, determining its location and adding the location to the incomplete instruction. In this case, the search determines that line 60 corresponds to SML location 15, so the completed instruction 03 +4215 is produced, replacing 03 +4200. The Simple program has now been compiled successfully.

To build the compiler, you will have to perform each of the following tasks:

a.

Modify the Simpletron simulator program you wrote in [Exercise 8.19](#) to take its input from a file specified by the user (see [Chapter 17](#)). The simulator should output its results to a disk file in the same format as the screen output. Convert the simulator to be an object-oriented program. In particular, make each part of the hardware an object. Arrange the instruction types into a class hierarchy using inheritance. Then execute the program polymorphically by telling each instruction to execute itself with an `executeInstruction` message.

b.

Modify the infix-to-postfix conversion algorithm of [Exercise 21.12](#) to process multidigit integer operands and single-letter variable name operands. [Hint: C++ Standard Library function `strtok` can be used to locate each constant and variable in an expression, and constants can be converted from strings to integers using standard library function `atoi` (`<csdlib>`).] [Note: The data representation of the postfix expression must be altered to support variable names and integer constants.]

c.

Modify the postfix evaluation algorithm to process multidigit integer operands and variable name operands. Also, the algorithm should now implement the "hook" discussed previously so that SML instructions are produced rather than directly evaluating the expression. [Hint: Standard library function `strtok` can be used to locate each constant and variable in an expression, and constants can be converted from strings to integers using standard library function `atoi`.] [Note: The data representation of the postfix expression must be altered to support variable names and integer constants.]

d.

Build the compiler. Incorporate parts (b) and (c) for evaluating expressions in `let` statements. Your program should contain a function that performs the first pass of the compiler and a function that performs the second pass of the compiler. Both functions can call other functions to accomplish their tasks. Make your compiler as object oriented as possible.

- 21.28** (Optimizing the Simple Compiler) When a program is compiled and converted into SML, a set of instructions is generated. Certain combinations of instructions often repeat themselves, usually in triplets called productions. A production normally consists of three instructions such as load, add, and store. For example, Fig. 21.32 illustrates five of the SML instructions that were produced in the compilation of the program in Fig. 21.30. The first three instructions are the production that adds 1 to  $y$ . Note that instructions 06 and 07 store the accumulator value in temporary location 96 and load the value back into the accumulator so instruction 08 can store the value in location 98. Often a production is followed by a load instruction for the same location that was just stored. This code can be optimized by eliminating the store instruction and the subsequent load instruction that operate on the same memory location, thus enabling the Simpletron to execute the program faster. Figure 21.33 illustrates the optimized SML for the program of Fig. 21.30. Note that there are four fewer instructions in the optimized code—a memory-space savings of 25 percent.

**Figure 21.32. Nonoptimized code from the program of Fig. 21.30.**

(This item is displayed on page 1054 in the print version)

```

1 04 +2098 (load)
2 05 +3097 (add)
3 06 +2196 (store)
4 07 +2096 (load)
5 08 +2198 (store)

```

**Figure 21.33. Optimized code for the program of Fig. 21.30.**

(This item is displayed on page 1054 in the print version)

Simple program	SML location & instruction	Description
5 rem sum 1 to $x$	none	rem ignored
10 input $x$	00 +1099	read $x$ into location 99
15 rem check $y == x$	none	rem ignored

20 if y == x goto 60	01 +2098	load y (98) into accumulator
	02 +3199	sub x (99) from accumulator
	03 +4211	branch to location 11 if zero
25 rem increment y	none	rem ignored
30 let y = y + 1	04 +2098	load y into accumulator
	05 +3097	add 1 (97) to accumulator
	06 +2198	store accumulator in y (98)
35 rem add y to total	none	rem ignored
40 let t = t + y	07 +2096	load t from location (96)
	08 +3098	add y (98) to accumulator
	09 +2196	store accumulator in t (96)
45 rem loop y	none	rem ignored
50 goto 20	10 +4001	branch to location 01
55 rem output result	none	rem ignored
60 print t	11 +1196	output t (96) to screen
99 end	12 +4300	terminate execution

Modify the compiler to provide an option for optimizing the Simpletron Machine Language code it produces. Manually compare the nonoptimized code with the optimized code, and calculate the percentage reduction.

- 21.29** (Modifications to the Simple Compiler) Perform the following modifications to the Simple compiler. Some of these modifications may also require modifications to the Simpletron Simulator program written in [Exercise 8.19](#).

a.

Allow the modulus operator (%) to be used in let statements. Simpletron Machine Language must be modified to include a modulus instruction.

b.

Allow exponentiation in a let statement using ^ as the exponentiation operator. Simpletron Machine Language must be modified to include an exponentiation instruction.

c.

Allow the compiler to recognize uppercase and lowercase letters in Simple statements (e.g., 'A' is equivalent to 'a'). No modifications to the Simulator are required.

d.

Allow input statements to read values for multiple variables such as input x, y. No modifications to the Simpletron Simulator are required.

---

[Page 1055]

e.

Allow the compiler to output multiple values in a single print statement such as print a, b, c. No modifications to the Simpletron Simulator are required.

f.

Add syntax-checking capabilities to the compiler so error messages are output when syntax errors are encountered in a Simple program. No modifications to the Simpletron Simulator are required.

g.

Allow arrays of integers. No modifications to the Simpletron Simulator are required.

**h.**

Allow subroutines specified by the Simple commands `gosub` and `return`. Command `gosub` passes program control to a subroutine, and command `return` passes control back to the statement after the `gosub`. This is similar to a function call in C++. The same subroutine can be called from many `gosub` commands distributed throughout a program. No modifications to the Simpletron Simulator are required.

**i.**

Allow repetition statements of the form

```
for x = 2 to 10 step 2
 Simple statements
next
```

This `for` statement loops from 2 to 10 with an increment of 2. The `next` line marks the end of the body of the `for`. No modifications to the Simpletron Simulator are required.

**j.**

Allow repetition statements of the form

```
for x = 2 to 10
 Simple statements
next
```

This `for` statement loops from 2 to 10 with a default increment of 1. No modifications to the Simpletron Simulator are required.

**k.**

Allow the compiler to process string input and output. This requires the Simpletron Simulator to be modified to process and store string values. [Hint: Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII decimal equivalent of a character. Add a machine-language instruction that will print a string beginning at a certain Simpletron memory location. The first half of the word at that location is a count of the number of characters in the string (i.e., the length of the string). Each succeeding half word contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and prints the string by translating each two-digit number into its equivalent

character.]

I.

Allow the compiler to process floating-point values in addition to integers. The Simpletron Simulator must also be modified to process floating-point values.

- 21.30** (A Simple Interpreter) An interpreter is a program that reads a high-level language program statement, determines the operation to be performed by the statement and executes the operation immediately. The high-level language program is not converted into machine language first. Interpreters execute slowly because each statement encountered in the program must first be deciphered. If statements are contained in a loop, the statements are deciphered each time they are encountered in the loop. Early versions of the BASIC programming language were implemented as interpreters.

Write an interpreter for the Simple language discussed in [Exercise 21.26](#). The program should use the infix-to-postfix converter developed in [Exercise 21.12](#) and the postfix evaluator developed in [Exercise 21.13](#) to evaluate expressions in a let statement. The same restrictions placed on the Simple language in [Exercise 21.26](#) should be adhered to in this program. Test the interpreter with the Simple programs written in [Exercise 21.26](#). Compare the results of running these programs in the interpreter with the results of compiling the Simple programs and running them in the Simpletron Simulator built in [Exercise 8.19](#).

---

[Page 1056]

- 21.31** (Insert/Delete Anywhere in a Linked List) Our linked list class template allowed insertions and deletions at only the front and the back of the linked list. These capabilities were convenient for us when we used private inheritance and composition to produce a stack class template and a queue class template with a minimal amount of code by reusing the list class template. Actually, linked lists are more general than those we provided. Modify the linked list class template we developed in this chapter to handle insertions and deletions anywhere in the list.

- 21.32** (List and Queues without Tail Pointers) Our implementation of a linked list ([Figs. 21.321.5](#)) used both a `firstPtr` and a `lastPtr`. The `lastPtr` was useful for the `insertAtBack` and `removeFromBack` member functions of the `List` class. The `insertAtBack` function corresponds to the `enqueue` member function of the `Queue` class. Rewrite the `List` class so that it does not use a `lastPtr`. Thus, any operations on the tail of a list must begin searching the list from the front. Does this affect our implementation of the `Queue` class ([Fig. 21.16](#))?

- 21.33** Use the composition version of the stack program (Fig. 21.15) to form a complete working stack program. Modify this program to inline the member functions. Compare the two approaches. Summarize the advantages and disadvantages of inlining member functions.
- 21.34** (Performance of Binary Tree Sorting and Searching) One problem with the binary tree sort is that the order in which the data is inserted affects the shape of the tree for the same collection of data, different orderings can yield binary trees of dramatically different shapes. The performance of the binary tree sorting and searching algorithms is sensitive to the shape of the binary tree. What shape would a binary tree have if its data was inserted in increasing order? in decreasing order? What shape should the tree have to achieve maximal searching performance?
- 21.35** (Indexed Lists) As presented in the text, linked lists must be searched sequentially. For large lists, this can result in poor performance. A common technique for improving list searching performance is to create and maintain an index to the list. An index is a set of pointers to various key places in the list. For example, an application that searches a large list of names could improve performance by creating an index with 26 entries one for each letter of the alphabet. A search operation for a last name beginning with 'Y' would first search the index to determine where the 'Y' entries begin and "jump into" the list at that point and search linearly until the desired name is found. This would be much faster than searching the linked list from the beginning. Use the `List` class of Figs. 21.321.5 as the basis of an `IndexedList` class. Write a program that demonstrates the operation of indexed lists. Be sure to include member functions `insertInIndexedList`, `searchIndexedList` and `deleteFromIndexedList`.

 PREVNEXT 

page footer

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1058]

## Outline

[22.1 Introduction](#)

[22.2 Structure Definitions](#)

[22.3 Initializing Structures](#)

[22.4 Using Structures with Functions](#)

[22.5 typedef](#)

[22.6 Example: High-Performance Card Shuffling and Dealing Simulation](#)

[22.7 Bitwise Operators](#)

[22.8 Bit Fields](#)

[22.9 Character-Handling Library](#)

[22.10 Pointer-Based String-Conversion Functions](#)

[22.11 Search Functions of the Pointer-Based String-Handling Library](#)

[22.12 Memory Functions of the Pointer-Based String-Handling Library](#)

[22.13 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

## Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1058 (continued)]

## 22.1. Introduction

In this chapter, we discuss structures and the manipulation of bits, characters and C-style strings. Many of the techniques we present here are included for the benefit of the C++ programmer who will work with C, and early C++, legacy code.

The designers of C++ evolved structures into the notion of a class. Like a class, C++ structures can contain access specifiers, member functions, constructors and destructors. In fact, the only difference between structures and classes in C++ is that structure members default to `public` access and class members default to `private` access when no access specifiers are used. Classes have been covered thoroughly in the book, so there is really no need for us to discuss structures in detail. Our presentation of structures in this chapter focuses on their use in C, where structures contain only `public` data members. This use of structures is typical of the legacy C code and early C++ code you'll see in industry.

We discuss how to declare structures, initialize structures and pass structures to functions. Then, we present a high-performance card shuffling and dealing simulation in which we use structure objects and C-style strings to represent the cards. We discuss the bitwise operators that allow programmers to access and manipulate the individual bits in bytes of data. We also present bitfieldsspecial structures that can be used to specify the exact number of bits a variable occupies in memory. These bit manipulation techniques are common in C and C++ programs that interact directly with hardware devices that have limited memory. The chapter finishes with examples of many character and C-style string manipulation functionssome of which are designed to process blocks of memory as arrays of bytes.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1059]

```
struct Card
{
 char *face;
 char *suit;
}; // end struct Card
```

Keyword **struct** introduces the definition for structure `Card`. The identifier `Card` is the **structure name** and is used in C++ to declare variables of the **structure type** (in C, the type name of the preceding structure is `struct Card`). In this example, the structure type is `Card`. Data (and possibly functions just as with classes) declared within the braces of the structure definition are the structure's **members**. Members of the same structure must have unique names, but two different structures may contain members of the same name without conflict. Each structure definition must end with a semicolon.

### Common Programming Error 22.1



Forgetting the semicolon that terminates a structure definition is a syntax error.

The definition of `Card` contains two members of type `char *face` and `suit`. Structure members can be variables of the fundamental data types (e.g., `int`, `double`, etc.) or aggregates, such as arrays, other structures and or classes. Data members in a single structure definition can be of many data types. For example, an `Employee` structure might contain character-string members for the first and last names, an `int` member for the employee's age, a `char` member containing '`M`' or '`F`' for the employee's gender, a `double` member for the employee's hourly salary and so on.

A structure cannot contain an instance of itself. For example, a structure variable `Card` cannot be declared in the definition for structure `Card`. A pointer to a `Card` structure, however, can be included. A structure containing a member that is a pointer to the same structure type is referred to as a **self-referential structure**. We used a similar construct self-referential classes in [Chapter 21](#), Data Structures, to build various kinds of linked data structures.

The `Card` structure definition does not reserve any space in memory; rather, it creates a new data type that is used to declare structure variables. Structure variables are declared like variables of other types. The following declarations

```
Card oneCard;
Card deck[52];
Card *cardPtr;
```

declare oneCard to be a structure variable of type Card, deck to be an array with 52 elements of type Card and cardPtr to be a pointer to a Card structure. Variables of a given structure type can also be declared by placing a comma-separated list of the variable names between the closing brace of the structure definition and the semicolon that ends the structure definition. For example, the preceding declarations could have been incorporated into the Card structure definition as follows:

```
struct Card
{
 char *face;
 char *suit;
} oneCard, deck[52], *cardPtr;
```

[Page 1060]

The structure name is optional. If a structure definition does not contain a structure name, variables of the structure type may be declared only between the closing right brace of the structure definition and the semicolon that terminates the structure definition.

### Software Engineering Observation 22.1



Provide a structure name when creating a structure type. The structure name is required for declaring new variables of the structure type later in the program, declaring parameters of the structure type and, if the structure is being used like a C++ class, specifying the name of the constructor and destructor.

The only valid built-in operations that may be performed on structure objects are assigning a structure object to a structure object of the same type, taking the address (&) of a structure object, accessing the members of a structure object (in the same manner as members of a class are accessed) and using the sizeof operator to determine the size of a structure. As with classes, most operators can be overloaded to work with objects of a structure type.

Structure members are not necessarily stored in consecutive bytes of memory. Sometimes there are "holes" in a structure, because some computers store specific data types only on certain memory boundaries, such as half-word, word or double-word boundaries. A word is a standard memory unit used to store data in a computer usually two bytes or four bytes and typically four bytes on today's popular 32-bit systems. Consider the following structure definition in which structure objects sample1 and sample2 of type Example are declared:

```
struct Example
{
 char c;
 int i;
} sample1, sample2;
```

A computer with two-byte words might require that each of the members of `Example` be aligned on a word boundary (i.e., at the beginning of a wordthis is machine dependent). [Figure 22.1](#) shows a sample storage alignment for an object of type `Example` that has been assigned the character 'a' and the integer 97 (the bit representations of the values are shown). If the members are stored beginning at word boundaries, there is a one-byte hole (byte 1 in the figure) in the storage for objects of type `Example`. The value in the 1-byte hole is undefined. If the member values of `sample1` and `sample2` are in fact equal, the structure objects are not necessarily equal, because the undefined 1-byte holes are not likely to contain identical values.

**Figure 22.1. Possible storage alignment for a variable of type `Example`, showing an undefined area in memory.**

[\[View full size image\]](#)

Byte	0	1	2	3
	01100001		00000000	01100001

## Common Programming Error 22.2



Comparing structures is a compilation error.

[Page 1061]

## Portability Tip 22.1



Because the size of data items of a particular type is machine dependent, and because storage alignment considerations are machine dependent, so too is the representation of a structure.

 PREV

NEXT 

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1061 (continued)]

## 22.3. Initializing Structures

Structures can be initialized using initializer lists, as is done with arrays. For example, the declaration

```
Card oneCard = { "Three", "Hearts" };
```

creates Card variable oneCard and initializes member face to "Three" and member suit to "Hearts". If there are fewer initializers in the list than members in the structure, the remaining members are initialized to their default values. Structure variables declared outside a function definition (i.e., externally) are initialized to their default values if they are not explicitly initialized in the external declaration. Structure variables may also be set in assignment expressions by assigning a structure variable of the same type or by assigning values to the individual data members of the structure.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 1061 (continued)]

## 22.4. Using Structures with Functions

There are two ways to pass the information in structures to functions. You can either pass the entire structure or pass the individual members of a structure. By default, structures are passed by value. Structures and their members can also be passed by reference by passing either references or pointers.

To pass a structure by reference, pass the address of the structure object or a reference to the structure object. Arrays of structures like all other arrays are passed by reference.

In [Chapter 7](#), we stated that an array could be passed by value by using a structure. To pass an array by value, create a structure (or a class) with the array as a member, then pass an object of that structure (or class) type to a function by value. Because structure objects are passed by value, the array member, too, is passed by value.

### Performance Tip 22.1



Passing structures (and especially large structures) by reference is more efficient than passing them by value (which requires the entire structure to be copied).

[◀ PREV](#)

[NEXT ▶](#)

[page footer](#)

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1062]

Creating a new name with `typedef` does not create a new type; `typedef` simply creates a new type name that can then be used in the program as an alias for an existing type name.

## Portability Tip 22.2



Synonyms for built-in data types can be created with `typedef` to make programs more portable. For example, a program can use `typedef` to create alias `Integer` for four-byte integers. `Integer` can then be aliased to `int` on systems with four-byte integers and can be aliased to `long int` on systems with two-byte integers where `long int` values occupy four bytes. Then, the programmer simply declares all four-byte integer variables to be of type `Integer`.

PREV

NEXT

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1064]

**Figure 22.4. High-performance card shuffling and dealing simulation.**

```
1 // Fig. 22.4: fig22_04.cpp
2 // Card shuffling and dealing program.
3 #include "DeckOfCards.h" // DeckOfCards class definition
4
5 int main()
6 {
7 DeckOfCards deckOfCards; // create DeckOfCards object
8
9 deckOfCards.shuffle(); // shuffle the cards in the deck
10 deckOfCards.deal(); // deal the cards in the deck
11 return 0; // indicates successful termination
12 } // end main
```

King of Clubs	Ten of Diamonds
Five of Diamonds	Jack of Clubs
Seven of Spades	Five of Clubs
Three of Spades	King of Hearts
Ten of Clubs	Eight of Spades
Eight of Hearts	Six of Hearts
Nine of Diamonds	Nine of Clubs
Three of Diamonds	Queen of Hearts
Six of Clubs	Seven of Hearts
Seven of Diamonds	Jack of Diamonds
Jack of Spades	King of Diamonds
Deuce of Diamonds	Four of Clubs
Three of Clubs	Five of Hearts
Eight of Clubs	Ace of Hearts
Deuce of Spades	Ace of Clubs
Ten of Spades	Eight of Diamonds
Ten of Hearts	Six of Spades
Queen of Diamonds	Nine of Hearts
Seven of Clubs	Queen of Clubs
Deuce of Clubs	Queen of Spades
Three of Hearts	Five of Spades
Deuce of Hearts	Jack of Hearts
Four of Hearts	Ace of Diamonds
Nine of Spades	Four of Diamonds

Ace of Spades  
Four of Spades

Six of Diamonds  
King of Spades

In the program, the constructor initializes the `Card` array in order with character strings representing the Ace through the King of each suit. Function `shuffle` is where the high-performance shuffling algorithm is implemented. The function loops through all 52 cards (array subscripts 0 to 51). For each card, a number between 0 and 51 is picked randomly. Next, the current `Card` structure and the randomly selected `Card` structure are swapped in the array. A total of 52 swaps are made in a single pass of the entire array, and the array of `Card` structures is shuffled! Unlike the shuffling algorithm presented in [Chapter 8](#), this algorithm does not suffer from indefinite postponement. Because the `Card` structures were swapped in place in the array, the high-performance dealing algorithm implemented in function `deal` requires only one pass of the array to deal the shuffled cards.

---

[Page 1065]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1066]

**Figure 22.5. Bitwise operators.**

Operator	Name	Description
&	bitwise AND	The bits in the result are set to 1 if the corresponding bits in the two operands are both 1.
	bitwise inclusive OR	The bits in the result are set to 1 if one or both of the corresponding bits in the two operands is 1.
^	bitwise exclusive OR	The bits in the result are set to 1 if exactly one of the corresponding bits in the two operands is 1.
<<	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from right with 0 bits.
>>	right shift with sign extension	Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine dependent.
~	bitwise complement	All 0 bits are set to 1 and all 1 bits are set to 0.

## Printing a Binary Representation of an Integral Value

When using the bitwise operators, it is useful to illustrate their precise effects by printing values in their binary representation. The program of Fig. 22.6 prints an unsigned integer in its binary representation in groups of eight bits each.

**Figure 22.6. Printing an unsigned integer in bits.**

(This item is displayed on pages 1066 - 1067 in the print version)

```
1 // Fig. 22.6: fig22_06.cpp
2 // Printing an unsigned integer in bits.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 void displayBits(unsigned); // prototype
12
13 int main()
14 {
15 unsigned inputValue; // integral value to print in binary
16
17 cout << "Enter an unsigned integer: ";
18 cin >> inputValue;
19 displayBits(inputValue);
20 return 0;
21 } // end main
22
23 // display bits of an unsigned integer value
24 void displayBits(unsigned value)
25 {
26 const int SHIFT = 8 * sizeof(unsigned) - 1;
27 const unsigned MASK = 1 << SHIFT;
28
29 cout << setw(10) << value << " = ";
30
31 // display bits
32 for (unsigned i = 1; i <= SHIFT + 1; i++)
33 {
34 cout << (value & MASK ? '1' : '0');
35 value <<= 1; // shift value left by 1
36
37 if (i % 8 == 0) // output a space after 8 bits
38 cout << ' ';
39 } // end for
40
41 cout << endl;
42 } // end function displayBits
```

```
Enter an unsigned integer: 65000
65000 = 00000000 00000000 11111101 11101000
```

```
Enter an unsigned integer: 29
29 = 00000000 00000000 00000000 00011101
```

[Page 1067]

Function `displayBits` (lines 2442) uses the bitwise AND operator to combine variable `value` with constant `MASK`. Often, the bitwise AND operator is used with an operand called a **mask** an integer value with specific bits set to 1. Masks are used to hide some bits in a value while selecting other bits. In `displayBits`, line 27 assigns constant `MASK` the value `1 << SHIFT`. The value of constant `SHIFT` was calculated in line 26 with the expression

```
8 * sizeof(unsigned) - 1
```

which multiplies the number of bytes an `unsigned` object requires in memory by 8 (the number of bits in a byte) to get the total number of bits required to store an `unsigned` object, then subtracts 1. The bit representation of `1 << SHIFT` on a computer that represents `unsigned` objects in four bytes of memory is

```
10000000 00000000 00000000 00000000
```

The left-shift operator shifts the value 1 from the low-order (rightmost) bit to the high-order (leftmost) bit in `MASK`, and fills in 0 bits from the right. Line 34 determines whether a 1 or a 0 should be printed for the current leftmost bit of variable `value`. Assume that variable `value` contains 65000 (00000000 00000000 11111101 11101000). When `value` and `MASK` are combined using `&`, all the bits except the high-order bit in variable `value` are "masked off" (hidden), because any bit "ANDed" with 0 yields 0. If the leftmost bit is 1, `value & MASK` evaluates to

[Page 1068]

```

00000000 00000000 11111101 11101000 (value)
10000000 00000000 00000000 00000000 (MASK)

00000000 00000000 00000000 00000000 (value & MASK)

```

which is interpreted as `false`, and 0 is printed. Then line 35 shifts variable `value` left by one bit with the expression `value <= 1` (i.e., `value = value << 1`). These steps are repeated for each bit variable `value`. Eventually, a bit with a value of 1 is shifted into the leftmost bit position, and the bit manipulation is as follows:

```

11111101 11101000 00000000 00000000 (value)
10000000 00000000 00000000 00000000 (MASK)

10000000 00000000 00000000 00000000 (value & MASK)

```

Because both left bits are 1s, the result of the expression is nonzero (true) and a value of 1 is printed. [Figure 22.7](#) summarizes the results of combining two bits with the bitwise AND operator.

**Figure 22.7. Results of combining two bits with the bitwise AND operator (&).**

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

### Common Programming Error 22.3



Using the logical AND operator (`&&`) for the bitwise AND operator (`&`) and vice versa is a logic error.

The program of Fig. 22.8 demonstrates the bitwise AND operator, the bitwise inclusive OR operator, the bitwise exclusive OR operator and the bitwise complement operator. Function `displayBits` (lines 5775) prints the unsigned integer values.

**Figure 22.8. Bitwise AND, bitwise inclusive-OR, bitwise exclusive-OR and bitwise complement operators.**

(This item is displayed on pages 1069 - 1070 in the print version)

```

1 // Fig. 22.8: fig22_08.cpp
2 // Using the bitwise AND, bitwise inclusive OR, bitwise
3 // exclusive OR and bitwise complement operators.
4 #include <iostream>
5 using std::cout;
6
7 #include <iomanip>
8 using std::endl;
9 using std::setw;
10
11 void displayBits(unsigned); // prototype
12
13 int main()
14 {
15 unsigned number1;
16 unsigned number2;
17 unsigned mask;
18 unsigned setBits;
19
20 // demonstrate bitwise &
21 number1 = 2179876355;
22 mask = 1;
23 cout << "The result of combining the following\n";
24 displayBits(number1);
25 displayBits(mask);
26 cout << "using the bitwise AND operator & is\n";
27 displayBits(number1 & mask);
28
29 // demonstrate bitwise |
30 number1 = 15;
31 setBits = 241;
32 cout << "\nThe result of combining the following\n";
33 displayBits(number1);
34 displayBits(setBits);
35 cout << "using the bitwise inclusive OR operator | is\n";
36 displayBits(number1 | setBits);
37
38 // demonstrate bitwise exclusive OR

```

```
39 number1 = 139;
40 number2 = 199;
41 cout << "\nThe result of combining the following\n";
42 displayBits(number1);
43 displayBits(number2);
44 cout << "using the bitwise exclusive OR operator ^ is\n";
45 displayBits(number ^ number2);
46
47 // demonstrate bitwise complement
48 number1 = 21845;
49 cout << "\nThe one's complement of\n";
50 displayBits(number1);
51 cout << "is" << endl;
52 displayBits(~number1);
53 return 0;
54 } // end main
55
56 // display bits of an unsigned integer value
57 void displayBits(unsigned value)
58 {
59 const int SHIFT = 8 * sizeof(unsigned) - 1;
60 const unsigned MASK = 1 << SHIFT;
61
62 cout << setw(10) << value << " = ";
63
64 // display bits
65 for (unsigned i = 1; i <= SHIFT + 1; i++)
66 {
67 cout << (value & MASK ? '1' : '0');
68 value <<= 1; // shift value left by 1
69
70 if (i % 8 == 0) // output a space after 8 bits
71 cout << ' ';
72 } // end for
73
74 cout << endl;
75 } // end function displayBits
```

```

The result of combining the following
2179876355 = 10000001 11101110 01000110 00000011
 1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
 1 = 00000000 00000000 00000000 00000001

The result of combining the following
 15 = 00000000 00000000 00000000 00001111
 241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
 255 = 00000000 00000000 00000000 11111111

The result of combining the following
 139 = 00000000 00000000 00000000 10001011
 199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
 76 = 00000000 00000000 00000000 01001100

The one's complement of
 21845 = 00000000 00000000 01010101 01010101
is
 4294945450 = 11111111 11111111 10101010 10101010

```

## Bitwise AND Operator (&)

In Fig. 22.8, line 21 assigns 2179876355 (10000001 11101110 01000110 00000011) to variable number1, and line 22 assigns 1 (00000000 00000000 00000000 00000001) to variable mask. When mask and number1 are combined using the bitwise AND operator (&) in the expression number1 & mask (line 27), the result is 00000000 00000000 00000000 00000001. All the bits except the low-order bit in variable number1 are "masked off" (hidden) by "ANDing" with constant MASK.

[Page 1069]

## Bitwise Inclusive OR Operator (|)

The bitwise inclusive-OR operator is used to set specific bits to 1 in an operand. In Fig. 22.8, line 30 assigns 15 (00000000 00000000 00000000 00001111) to variable number1, and line 31 assigns 241 (00000000 00000000 00000000 11110001) to variable setBits. When number1 and setBits are combined using the bitwise OR operator in the expression number1 | setBits (line 36),

the result is 255 (00000000 00000000 00000000 11111111). Figure 22.9 summarizes the results of combining two bits with the bitwise inclusive-OR operator.

[Page 1071]

**Figure 22.9. Combining two bits with the bitwise inclusive-OR operator (|).**

Bit 1	Bit 2	Bit 1   Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

#### Common Programming Error 22.4



Using the logical OR operator (||) for the bitwise OR operator (|) and vice versa is a logic error.

#### Bitwise Exclusive OR (^)

The bitwise exclusive OR operator (^) sets each bit in the result to 1 if exactly one of the corresponding bits in its two operands is 1. In Fig. 22.8, lines 3940 assign variables `number1` and `number2` the values 139 (00000000 00000000 00000000 10001011) and 199 (00000000 00000000 00000000 11000111), respectively. When these variables are combined with the exclusive-OR operator in the expression `number1 ^ number2` (line 45), the result is 00000000 00000000 00000000 01001100. Figure 22.10 summarizes the results of combining two bits with the bitwise exclusive-OR operator.

**Figure 22.10. Combining two bits with the bitwise exclusive-OR operator (^).**

<b>Bit 1</b>	<b>Bit 2</b>	<b>Bit 1 ^ Bit 2</b>
0	0	0
1	0	1
0	1	1
1	1	0

## Bitwise Complement (~)

The bitwise complement operator (~) sets all 1 bits in its operand to 0 in the result and sets all 0 bits to 1 in the result otherwise referred to as "taking the one's complement of the value." In Fig. 22.8, line 48 assigns variable `number1` the value 21845 (00000000 00000000 01010101 01010101). When the expression `~number1` evaluates, the result is (11111111 11111111 10101010 10101010).

Figure 22.11 demonstrates the left-shift operator (<<) and the right-shift operator (>>). Function `displayBits` (lines 3149) prints the `unsigned` integer values.

## Figure 22.11. Bitwise shift operators.

(This item is displayed on pages 1072 - 1073 in the print version)

```

1 // Fig. 22.11: fig22_11.cpp
2 // Using the bitwise shift operators.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 void displayBits(unsigned); // prototype
11
12 int main()
13 {
14 unsigned number1 = 960;
15
16 // demonstrate bitwise left shift
17 cout << "The result of left shifting\n";
18 displayBits(number1);

```

```

19 cout << "8 bit positions using the left-shift operator is\n";
20 displayBits(number1 << 8);
21
22 // demonstrate bitwise right shift
23 cout << "\nThe result of right shifting\n";
24 displayBits(number1);
25 cout << "8 bit positions using the right-shift operator is\n";
26 displayBits(number1 >> 8);
27 return 0;
28 } // end main
29
30 // display bits of an unsigned integer value
31 void displayBits(unsigned value)
32 {
33 const int SHIFT = 8 * sizeof(unsigned) - 1;
34 const unsigned MASK = 1 << SHIFT;
35
36 cout << setw(10) << value << " = ";
37
38 // display bits
39 for (unsigned i = 1; i <= SHIFT + 1; i++)
40 {
41 cout << (value & MASK ? '1' : '0');
42 value <<= 1; // shift value left by 1
43
44 if (i % 8 == 0) // output a space after 8 bits
45 cout << ' ';
46 } // end for
47
48 cout << endl;
49 } // end function displayBits

```

The result of left shifting  
960 = 00000000 00000000 00000011 11000000  
8 bit positions using the left-shift operator is  
245760 = 00000000 00000011 11000000 00000000

The result of right shifting  
960 = 00000000 00000000 00000011 11000000  
8 bit positions using the right-shift operator is  
3 = 00000000 00000000 00000000 00000011

---

[Page 1072]

## Left-Shift Operator

The left-shift operator (`<<`) shifts the bits of its left operand to the left by the number of bits specified in its right operand. Bits vacated to the right are replaced with 0s; bits shifted off the left are lost. In the program of Fig. 22.11, line 14 assigns variable `number1` the value 960 (00000000 00000000 00000011 11000000). The result of left-shifting variable `number1` 8 bits in the expression `number1 << 8` (line 20) is 245760 (00000000 00000011 11000000 00000000).

---

[Page 1073]

## Right-Shift Operator

The right-shift operator (`>>`) shifts the bits of its left operand to the right by the number of bits specified in its right operand. Performing a right shift on an `unsigned` integer causes the vacated bits at the left to be replaced by 0s; bits shifted off the right are lost. In the program of Fig. 22.11, the result of right-shifting `number1` in the expression `number1 >> 8` (line 26) is 3 (00000000 00000000 00000000 00000011).

### Common Programming Error 22.5



The result of shifting a value is undefined if the right operand is negative or if the right operand is greater than or equal to the number of bits in which the left operand is stored.

### Portability Tip 22.4



The result of right-shifting a signed value is machine dependent. Some machines fill with zeros and others use the sign bit.

## Bitwise Assignment Operators

Each bitwise operator (except the bitwise complement operator) has a corresponding assignment operator. These **bitwise assignment operators** are shown in Fig. 22.12; they are used in a similar manner to the arithmetic assignment operators introduced in Chapter 2.

**Figure 22.12. Bitwise assignment operators.**

<b>Bitwise assignment operators</b>	
<b>&amp;=</b>	Bitwise AND assignment operator.
<b> =</b>	Bitwise inclusive-OR assignment operator.
<b>^=</b>	Bitwise exclusive-OR assignment operator.
<b>&lt;&lt;=</b>	Left-shift assignment operator.
<b>&gt;&gt;=</b>	Right-shift with sign extension assignment operator.

---

[Page 1074]

Figure 22.13 shows the precedence and associativity of the operators introduced up to this point in the text. They are shown top to bottom in decreasing order of precedence.

**Figure 22.13. Operator precedence and associativity.**

<b>Operators</b>							<b>Associativity</b>	<b>Type</b>	
:: (unary; right to left)			:: (binary; left to right)					left to right	highest
( )	[ ]	.	->	++	--	static_cast< type >()	left to right	unary	
++	--	+	-	!	delete sizeof			right to left	unary
*	~	&	new						
*	/	%					left to right	multiplicative	
+	-						left to right	additive	
<<	>>						left to right	shifting	
<	<=	>	>=				left to right	relational	
==	!=						left to right	equality	

&												left to right	bitwise AND
^												left to right	bitwise XOR
												left to right	bitwise OR
&&												left to right	logical AND
												left to right	logical OR
? :												right to left	conditional
=    +=    -=    *=    /=    %=    &=     =    ^=    <<=    >>=												right to left	assignment
,												left to right	comma

 PREVNEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1075]

The definition contains three unsigned bit fields `face`, `suit` and `color` used to represent a card from a deck of 52 cards. A bit field is declared by following an integral type or `enum` type member with a colon (`:`) and an integer constant representing the **width of the bit field** (i.e., the number of bits in which the member is stored). The width must be an integer constant.

The preceding structure definition indicates that member `face` is stored in 4 bits, member `suit` in 2 bits and member `color` in 1 bit. The number of bits is based on the desired range of values for each structure member. Member `face` stores values between 0 (Ace) and 12 (King). 4 bits can store a value between 0 and 15. Member `suit` stores values between 0 and 3 (0 = Diamonds, 1 = Hearts, 2 = Clubs, 3 = Spades). 2 bits can store a value between 0 and 3. Finally, member `color` stores either 0 (Red) or 1 (Black). 1 bit can store either 0 or 1.

The program in [Figs. 22.14](#)[22.16](#) creates array `deck` containing 52 `BitCard` structures (line 21 of [Fig. 22.14](#)). The constructor inserts the 52 cards in the `deck` array, and function `deal` prints the 52 cards. Notice that bit fields are accessed exactly as any other structure member is (lines 1820 and 2833 of [Fig. 22.15](#)). The member `color` is included as a means of indicating the card color on a system that allows color displays.

**Figure 22.14. Header file for class DeckOfCards.**

```

1 // Fig. 22.14: DeckOfCards.h
2 // Definition of class DeckOfCards that
3 // represents a deck of playing cards.
4
5 // BitCard structure definition with bit fields
6 struct BitCard
7 {
8 unsigned face : 4; // 4 bits; 0-15
9 unsigned suit : 2; // 2 bits; 0-3
10 unsigned color : 1; // 1 bit; 0-1
11 } // end struct BitCard
12
13 // DeckOfCards class definition
14 class DeckOfCards
15 {
16 public:
17 DeckOfCards(); // constructor initializes deck
18 void deal(); // deals cards in deck
19

```

```

20 private:
21 BitCard deck[52]; // represents deck of cards
22 };// end class DeckOfCards

```

**Figure 22.15. Class file for DeckOfCards.**

(This item is displayed on pages 1075 - 1076 in the print version)

```

1 // Fig. 22.15: DeckOfCards.cpp
2 // Member-function definitions for class DeckOfCards that simulates
3 // the shuffling and dealing of a deck of playing cards.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include "DeckOfCards.h" // DeckOfCards class definition
12
13 // no-argument DeckOfCards constructor initializes deck
14 DeckOfCards::DeckOfCards()
15 {
16 for (int i = 0; i <= 51; i++)
17 {
18 deck[i].face = i % 13; // faces in order
19 deck[i].suit = i / 13; // suits in order
20 deck[i].color = i / 26; // colors in order
21 } // end for
22 } // end no-argument DeckOfCards constructor
23
24 // deal cards in deck
25 void DeckOfCards::deal()
26 {
27 for (int k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++)
28 cout << "Card:" << setw(3) << deck[k1].face
29 << " Suit:" << setw(2) << deck[k1].suit
30 << " Color:" << setw(2) << deck[k1].color
31 << " " << "Card:" << setw(3) << deck[k2].face
32 << " Suit:" << setw(2) << deck[k2].suit
33 << " Color:" << setw(2) << deck[k2].color << endl;
34 } // end function deal

```

**Figure 22.16. Bit fields used to store a deck of cards.**

(This item is displayed on page 1077 in the print version)

```
1 // Fig. 22.16: fig22_16.cpp
2 // Card shuffling and dealing program.
3 #include "DeckOfCards.h" // DeckOfCards class definition
4
5 int main()
6 {
7 DeckOfCards deckOfCards; // create DeckOfCards object
8 deckOfCards.deal(); // deal the cards in the deck
9 return 0; // indicates successful termination
10 } // end main
```

Card: 0 Suit: 0 Color: 0	Card: 0 Suit: 2 Color: 1
Card: 1 Suit: 0 Color: 0	Card: 1 Suit: 2 Color: 1
Card: 2 Suit: 0 Color: 0	Card: 2 Suit: 2 Color: 1
Card: 3 Suit: 0 Color: 0	Card: 3 Suit: 2 Color: 1
Card: 4 Suit: 0 Color: 0	Card: 4 Suit: 2 Color: 1
Card: 5 Suit: 0 Color: 0	Card: 5 Suit: 2 Color: 1
Card: 6 Suit: 0 Color: 0	Card: 6 Suit: 2 Color: 1
Card: 7 Suit: 0 Color: 0	Card: 7 Suit: 2 Color: 1
Card: 8 Suit: 0 Color: 0	Card: 8 Suit: 2 Color: 1
Card: 9 Suit: 0 Color: 0	Card: 9 Suit: 2 Color: 1
Card: 10 Suit: 0 Color: 0	Card: 10 Suit: 2 Color: 1
Card: 11 Suit: 0 Color: 0	Card: 11 Suit: 2 Color: 1
Card: 12 Suit: 0 Color: 0	Card: 12 Suit: 2 Color: 1
Card: 0 Suit: 1 Color: 0	Card: 0 Suit: 3 Color: 1
Card: 1 Suit: 1 Color: 0	Card: 1 Suit: 3 Color: 1
Card: 2 Suit: 1 Color: 0	Card: 2 Suit: 3 Color: 1
Card: 3 Suit: 1 Color: 0	Card: 3 Suit: 3 Color: 1
Card: 4 Suit: 1 Color: 0	Card: 4 Suit: 3 Color: 1
Card: 5 Suit: 1 Color: 0	Card: 5 Suit: 3 Color: 1
Card: 6 Suit: 1 Color: 0	Card: 6 Suit: 3 Color: 1
Card: 7 Suit: 1 Color: 0	Card: 7 Suit: 3 Color: 1
Card: 8 Suit: 1 Color: 0	Card: 8 Suit: 3 Color: 1
Card: 9 Suit: 1 Color: 0	Card: 9 Suit: 3 Color: 1
Card: 10 Suit: 1 Color: 0	Card: 10 Suit: 3 Color: 1
Card: 11 Suit: 1 Color: 0	Card: 11 Suit: 3 Color: 1
Card: 12 Suit: 1 Color: 0	Card: 12 Suit: 3 Color: 1

---

[Page 1076]

It is possible to specify an **unnamed bit field**, in which case the field is used as **padding** in the structure. For example, the structure definition uses an unnamed 3-bit field as paddingnothing can be stored in those 3 bits. Member b is stored in another storage unit.

```
struct Example
{
 unsigned a : 13;
 unsigned : 3; // align to next storage-unit boundary
 unsigned b : 4;
}; // end struct Example
```

An **unnamed bit field with a zero width** is used to align the next bit field on a new storage-unit boundary. For example, the structure definition

```
struct Example
{
 unsigned a : 13;
 unsigned : 0; // align to next storage-unit boundary
 unsigned b : 4;
}; // end struct Example
```

uses an unnamed 0-bit field to skip the remaining bits (as many as there are) of the storage unit in which a is stored and align b on the next storage-unit boundary.

---

[Page 1077]

## Portability Tip 22.5



Bit-field manipulations are machine dependent. For example, some computers allow bit fields to cross word boundaries, whereas others do not.

## Common Programming Error 22.6



Attempting to access individual bits of a bit field with subscripting as if they were elements of an array is a compilation error. Bit fields are not "arrays of bits."

## Common Programming Error 22.7



Attempting to take the address of a bit field (the & operator may not be used with bit fields because a pointer can designate only a particular byte in memory and bit fields can start in the middle of a byte) is a compilation error.

---

[Page 1078]

## Performance Tip 22.3



Although bit fields save space, using them can cause the compiler to generate slower-executing machine-language code. This occurs because it takes extra machine-language operations to access only portions of an addressable storage unit. This is one of many examples of the space-time trade-offs that occur in computer science.



page footer



The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1079]

Prototype

Description

```
int isdigit(int c)
```

Returns `true` if `c` is a digit and `false` otherwise.

```
int isalpha(int c)
```

Returns `true` if `c` is a letter and `false` otherwise.

```
int isalnum(int c)
```

Returns `true` if `c` is a digit or a letter and `false` otherwise.

```
int isxdigit(int c)
```

Returns `TRUE` if `c` is a hexadecimal digit character and `false` otherwise. (See [Appendix D](#), Number Systems, for a detailed explanation of binary, octal, decimal and hexadecimal numbers.)

```
int islower(int c)
```

Returns `true` if `c` is a lowercase letter and `false` otherwise.

```
int isupper(int c)
```

Returns `TRUE` if `c` is an uppercase letter; `false` otherwise.

```
int tolower(int c)
```

If `c` is an uppercase letter, `tolower` returns `c` as a lowercase letter. Otherwise, `tolower` returns the argument unchanged.

```
int toupper(int c)
```

If `c` is a lowercase letter, `toupper` returns `c` as an uppercase letter. Otherwise, `toupper` returns the argument unchanged.

```
int isspace(int c)
```

Returns `true` if `c` is a white-space character newline ('`\n`'), space ('`'`'), form feed ('`\f`'), carriage return ('`\r`'), horizontal tab ('`\t`'), or vertical tab ('`\v`') and `false` otherwise.

```
int iscntrl(int c)
```

Returns `TRUE` if `c` is a control character, such as newline ('`\n`'), form feed ('`\f`'), carriage return ('`\r`'), horizontal tab ('`\t`'), vertical tab ('`\v`'), alert ('`\a`'), or backspace ('`\b`') and `false` otherwise.

```
int ispunct(int c)
```

Returns `true` if `c` is a printing character other than a space, a digit, or a letter and `false` otherwise.

```
int isprint(int c)
```

Returns `true` value if `c` is a printing character including space ('`'`) and `false` otherwise.

```
int isgraph(int c)
```

Returns `TRUE` if `c` is a printing character other than space ('`'`) and `false` otherwise.

[Page 1079]

**Figure 22.18** demonstrates functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. Function `isdigit` determines whether its argument is a digit (09). Function `isalpha` determines whether its argument is an uppercase letter (A-Z) or a lowercase letter (az). Function `isalnum` determines whether its argument is an uppercase letter, a lowercase letter or a digit. Function `isxdigit` determines whether its argument is a hexadecimal digit (AF, af, 09).

**Figure 22.18. Character-handling functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`.**

(This item is displayed on pages 1079 - 1080 in the print version)

```

1 // Fig. 22.18: fig22_18.cpp
2 // Using functions isdigit, isalpha, isalnum and isxdigit.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cctype> // character-handling function prototypes
8 using std::isalnum;
9 using std::isalpha;
10 using std::isdigit;
11 using std::isxdigit;
12
13 int main()
14 {
15 cout << "According to isdigit:\n"
16 << (isdigit('8') ? "8 is a" : "8 is not a") << " digit\n"
17 << (isdigit('#') ? "# is a" : "# is not a") << " digit\n";
18
19 cout << "\nAccording to isalpha:\n"
20 << (isalpha('A') ? "A is a" : "A is not a") << " letter\n"
21 << (isalpha('b') ? "b is a" : "b is not a") << " letter\n"
22 << (isalpha('&') ? "& is a" : "& is not a") << " letter\n"
23 << (isalpha('4') ? "4 is a" : "4 is not a") << " letter\n";
24
25 cout << "\nAccording to isalnum:\n"
26 << (isalnum('A') ? "A is a" : "A is not a")
27 << " digit or a letter\n"
28 << (isalnum('8') ? "8 is a" : "8 is not a")
29 << " digit or a letter\n"
30 << (isalnum('#') ? "# is a" : "# is not a")
31 << " digit or a letter\n";
32
33 cout << "\nAccording to isxdigit:\n"
34 << (isxdigit('F') ? "F is a" : "F is not a")
35 << " hexadecimal digit\n"
36 << (isxdigit('J') ? "J is a" : "J is not a")
37 << " hexadecimal digit\n"
38 << (isxdigit('7') ? "7 is a" : "7 is not a")
39 << " hexadecimal digit\n"
40 << (isxdigit('$') ? "$ is a" : "$ is not a")
41 << " hexadecimal digit\n"
42 << (isxdigit('f') ? "f is a" : "f is not a")
43 << " hexadecimal digit" << endl;
44 return 0;
45 } // end main

```

```
According to isdigit:
8 is a digit
is not a digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

[Page 1080]

Figure 22.18 uses the conditional operator (?:) with each function to determine whether the string " is a " or the string " is not a " should be printed in the output for each character tested. For example, line 16 indicates that if '8' is a digit i.e., if isdigit returns a true (nonzero) value the string "8 is a " is printed. If '8' is not a digit (i.e., if isdigit returns 0), the string "8 is not a " is printed.

Figure 22.19 demonstrates functions **islower**, **isupper**, **tolower** and **toupper**. Function **islower** determines whether its argument is a lowercase letter (az). Function **isupper** determines whether its argument is an uppercase letter (AZ). Function **tolower** converts an uppercase letter to a lowercase letter and returns the lowercase letter if the argument is not an uppercase letter, **tolower** returns the argument value unchanged. Function **toupper** converts a lowercase letter to an uppercase letter and returns the uppercase letter if the argument is not a lowercase letter, **toupper** returns the argument value unchanged.

[Page 1081]

**Figure 22.19. Character-handling functions `islower`, `isupper`, `tolower` and `toupper`.**

(This item is displayed on pages 1081 - 1082 in the print version)

```

1 // Fig. 22.19: fig22_19.cpp
2 // Using functions islower, isupper, tolower and toupper.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cctype> // character-handling function prototypes
8 using std::islower;
9 using std::isupper;
10 using std::tolower;
11 using std::toupper;
12
13 int main()
14 {
15 cout << "According to islower:\n"
16 << (islower('p') ? "p is a" : "p is not a")
17 << " lowercase letter\n"
18 << (islower('P') ? "P is a" : "P is not a")
19 << " lowercase letter\n"
20 << (islower('5') ? "5 is a" : "5 is not a")
21 << " lowercase letter\n"
22 << (islower('!') ? "!" is a" : "!" is not a")
23 << " lowercase letter\n";
24
25 cout << "\nAccording to isupper:\n"
26 << (isupper('D') ? "D is an" : "D is not an")
27 << " uppercase letter\n"
28 << (isupper('d') ? "d is an" : "d is not an")
29 << " uppercase letter\n"
30 << (isupper('8') ? "8 is an" : "8 is not an")
31 << " uppercase letter\n"
32 << (isupper('$') ? "$ is an" : "$ is not an")
33 << " uppercase letter\n";
34
35 cout << "\nu converted to uppercase is "
36 << static_cast< char >(toupper('u'))
37 << "\n7 converted to uppercase is "
38 << static_cast< char >(toupper('7'))
39 << "\n$ converted to uppercase is "
40 << static_cast< char >(toupper('$'))
41 << "\nL converted to lowercase is "
42 << static_cast< char >(tolower('L')) << endl;
43 return 0;

```

```
44 } // end main
```

```
According to islower:
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l
```

[Page 1082]

**Figure 22.20** demonstrates functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`. Function `isspace` determines whether its argument is a white-space character, such as space (' '), form feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t') or vertical tab ('\v'). Function `iscntrl` determines whether its argument is a control character such as horizontal tab ('\t'), vertical tab ('\v'), form feed ('\f'), alert ('\a'), backspace ('\b'), carriage return ('\r') or newline ('\n'). Function `ispunct` determines whether its argument is a printing character other than a space, digit or letter, such as \$, #, (, ), [ , ], {, }, ;, : or %. Function `isprint` determines whether its argument is a character that can be displayed on the screen (including the space character). Function `isgraph` tests for the same characters as `isprint`; however, the space character is not included.

[Page 1084]

**Figure 22.20. Character-handling functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`.**

(This item is displayed on pages 1082 - 1083 in the print version)

```

1 // Fig. 22.20: fig22_20.cpp
2 // Using functions isspace, iscntrl, ispunct, isprint, isgraph.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cctype> // character-handling function prototypes
8 using std::iscntrl;
9 using std::isgraph;
10 using std::isprint;
11 using std::ispunct;
12 using std::isspace;
13
14 int main()
15 {
16 cout << "According to isspace:\nNewline "
17 << (isspace('\n') ? "is a" : "is not a")
18 << " whitespace character\nHorizontal tab "
19 << (isspace('\t') ? "is a" : "is not a")
20 << " whitespace character\n"
21 << (isspace('%') ? "% is a" : "% is not a")
22 << " whitespace character\n";
23
24 cout << "\nAccording to iscntrl:\nNewline "
25 << (iscntrl('\n') ? "is a" : "is not a")
26 << " control character\n"
27 << (iscntrl('$') ? "$ is a" : "$ is not a")
28 << " control character\n";
29
30 cout << "\nAccording to ispunct:\n"
31 << (ispunct(';') ? ";" is a" : ";" is not a")
32 << " punctuation character\n"
33 << (ispunct('Y') ? "Y is a" : "Y is not a")
34 << " punctuation character\n"
35 << (ispunct('#') ? "# is a" : "# is not a")
36 << " punctuation character\n";
37
38 cout << "\nAccording to isprint:\n"
39 << (isprint('$') ? "$ is a" : "$ is not a")
40 << " printing character\nAlert "
41 << (isprint('\a') ? "is a" : "is not a")
42 << " printing character\nSpace "
43 << (isprint(' ') ? "is a" : "is not a")
44 << " printing character\n";
45
46 cout << "\nAccording to isgraph:\n"
47 << (isgraph('Q') ? "Q is a" : "Q is not a")

```

```
48 << " printing character other than a space\nSpace "
49 << (isgraph(' ') ? "is a" : "is not a")
50 << " printing character other than a space" << endl;
51 return 0;
52 } // end main
```

According to isspace:  
Newline is a whitespace character  
Horizontal tab is a whitespace character  
% is not a whitespace character

According to iscntrl:  
Newline is a control character  
\$ is not a control character

According to ispunct:  
; is a punctuation character  
Y is not a punctuation character  
# is a punctuation character

According to isprint:  
\$ is a printing character  
Alert is not a printing character  
Space is a printing character

According to isgraph:  
Q is a printing character other than a space  
Space is not a printing character other than a space

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1085]

Prototype

Description

```
double atof(const char *nPtr)
```

Converts the string `nPtr` to `double`. If the string cannot be converted, 0 is returned.

```
int atoi(const char *nPtr)
```

Converts the string `nPtr` to `int`. If the string cannot be converted, 0 is returned.

```
long atol(const char *nPtr)
```

Converts the string `nPtr` to `long int`. If the string cannot be converted, 0 is returned.

```
double strtod(const char *nPtr, char **endPtr)
```

Converts the string `nPtr` to `double`. `endPtr` is the address of a pointer to the rest of the string after the `double`. If the string cannot be converted, 0 is returned.

```
long strtol(const char *nPtr, char **endPtr, int base)
```

Converts the string `nPtr` to `long`. `endPtr` is the address of a pointer to the rest of the string after the `long`. If the string cannot be converted, 0 is returned. The `base` parameter indicates the base of the number to convert (e.g., 8 for octal, 10 for decimal or 16 for hexadecimal). The default is decimal.

```
unsigned long strtoul(const char *nPtr, char **endPtr, int base)
```

Converts the string `nPtr` to `unsigned long`. `endPtr` is the address of a pointer to the rest of the string after the `unsigned long`. If the string cannot be converted, 0 is returned. The `base` parameter indicates the base of the number to convert (e.g., 8 for octal, 10 for decimal or 16 for hexadecimal). The default is decimal.

---

[Page 1085]

Function `atof` (Fig. 22.22, line 12) converts its argument a string that represents a floating-point number to a double value. The function returns the double value. If the string cannot be converted for example, if the first character of the string is not a digit function `atof` returns zero.

**Figure 22.22. String-conversion function `atof`.**

```

1 // Fig. 22.22: fig22_22.cpp
2 // Using atof.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // atof prototype
8 using std::atof;
9
10 int main()
11 {
12 double d = atof("99.0"); // convert string to double
13
14 cout << "The string \"99.0\" converted to double is " << d
15 << "\nThe converted value divided by 2 is " << d / 2.0 << endl;
16 return 0;
17 } // end main

```

The string "99.0" converted to double is 99  
 The converted value divided by 2 is 49.5

Function `atoi` (Fig. 22.23, line 12) converts its argument a string of digits that represents an integer to an `int` value. The function returns the `int` value. If the string cannot be converted, function `atoi` returns zero.

**Figure 22.23. String-conversion function atoi.**

(This item is displayed on page 1086 in the print version)

```
1 // Fig. 22.23: Fig22_26.cpp
2 // Using atoi.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // atoi prototype
8 using std::atoi;
9
10 int main()
11 {
12 int i = atoi("2593"); // convert string to int
13
14 cout << "The string \"2593\" converted to int is " << i
15 << "\nThe converted value minus 593 is " << i - 593 << endl;
16 return 0;
17 } // end main
```

```
The string "2593" converted to int is 2593
The converted value minus 593 is 2000
```

Function `atoi` (Fig. 22.24, line 12) converts its argument a string of digits representing a long integer to a long value. The function returns the long value. If the string cannot be converted, function `atoi` returns zero. If `int` and `long` are both stored in four bytes, function `atoi` and function `atol` work identically.

[Page 1086]

**Figure 22.24. String-conversion function atol.**

```

1 // Fig. 22.24: fig22_24.cpp
2 // Using atol.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // atol prototype
8 using std::atol;
9
10 int main()
11 {
12 long x = atol("1000000"); // convert string to long
13
14 cout << "The string \"1000000\" converted to long is " << x
15 << "\nThe converted value divided by 2 is " << x / 2 << endl;
16 return 0;
17 } // end main

```

The string "1000000" converted to long int is 1000000  
 The converted value divided by 2 is 500000

Function **strtod** (Fig. 22.25) converts a sequence of characters representing a floating-point value to double. Function **strtod** receives two arguments a string (`char *`) and the address of a `char *` pointer (i.e., a `char **`). The string contains the character sequence to be converted to double. The second argument enables **strtod** to modify a `char *` pointer in the calling function, such that the pointer points to the location of the first character after the converted portion of the string. Line 16 indicates that `d` is assigned the double value converted from `string` and that `stringPtr` is assigned the location of the first character after the converted value (51.2) in `string`.

---

[Page 1087]

**Figure 22.25. String-conversion function `strtod`.**

```

1 // Fig. 22.25: fig22_25.cpp
2 // Using strtod.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // strtod prototype
8 using std::strtod;
9
10 int main()
11 {
12 double d;
13 const char *string1 = "51.2% are admitted";
14 char *stringPtr;
15
16 d = strtod(string1, &stringPtr); // convert characters to double
17
18 cout << "The string \" " << string1
19 << "\" is converted to the\ndouble value " << d
20 << " and the string \" " << stringPtr << "\"" << endl;
21 return 0;
22 } // end main

```

The string "51.2% are admitted" is converted to the double value 51.2 and the string "% are admitted"

Function **strtol** (Fig. 22.26) converts to long a sequence of characters representing an integer. The function receives three arguments a string (`char *`), the address of a `char *` pointer and an integer. The string contains the character sequence to convert. The second argument is assigned the location of the first character after the converted portion of the string. The integer specifies the base of the value being converted. Line 16 indicates that `x` is assigned the `long` value converted from `string` and that `remainderPtr` is assigned the location of the first character after the converted value (-1234567) in `string1`. Using a null pointer for the second argument causes the remainder of the string to be ignored. The third argument, 0, indicates that the value to be converted can be in octal (base 8), decimal (base 10) or hexadecimal (base 16). This is determined by the initial characters in the string. 0x indicates hexadecimal and a number from 19 indicates decimal.

**Figure 22.26. String-conversion function `strtol`.**

(This item is displayed on page 1088 in the print version)

```

1 // Fig. 22.26: fig22_26.cpp
2 // Using strtol.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // strtol prototype
8 using std::strtol;
9
10 int main()
11 {
12 long x;
13 const char *string1 = "-1234567abc";
14 char *remainderPtr;
15
16 x = strtol(string1, &remainderPtr, 0); // convert characters to long
17
18 cout << "The original string is \""
19 << string1
20 << "\nThe converted value is " << x
21 << "\nThe remainder of the original string is \""
22 << remainderPtr
23 << "\nThe converted value plus 567 is " << x + 567 << endl;
24 return 0;
25 } // end main

```

```

The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000

```

In a call to function `strtol`, the base can be specified as zero or as any value between 2 and 36. (See [Appendix D](#) for a detailed explanation of the octal, decimal, hexadecimal and binary number systems.) Numeric representations of integers from base 11 to base 36 use the characters AZ to represent the values 10 to 35. For example, hexadecimal values can consist of the digits 09 and the characters AF. A base-11 integer can consist of the digits 09 and the character A. A base-24 integer can consist of the digits 09 and the characters AN. A base-36 integer can consist of the digits 09 and the characters AZ. [Note: The case of the letter used is ignored.]

Function **strtoul** (Fig. 22.27) converts to `unsigned long` a sequence of characters representing an `unsigned long` integer. The function works identically to function `strtol`. Line 17 indicates that `x` is assigned the `unsigned long` value converted from `string` and that `remainderPtr` is assigned the location of the first character after the converted value (1234567) in `string1`. The third argument, 0, indicates that the value to be converted can be in octal, decimal or hexadecimal format, depending on the initial characters.

**Figure 22.27. String-conversion function `strtoul`.**

(This item is displayed on pages 1088 - 1089 in the print version)

```

1 // Fig. 22.27: fig22_27.cpp
2 // Using strtoul.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // strtoul prototype
8 using std::strtoul;
9
10 int main()
11 {
12 unsigned long x;
13 const char *string1 = "1234567abc";
14 char *remainderPtr;
15
16 // convert a sequence of characters to unsigned long
17 x = strtoul(string1, &remainderPtr, 0);
18
19 cout << "The original string is \""
20 << string1
21 << "\nThe converted value is " << x
22 << "\nThe remainder of the original string is \""
23 << remainderPtr
24 << "\nThe converted value minus 567 is " << x - 567 << endl;
25 return 0;
26 } // end main

```

```

The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000

```

 PREV

NEXT 

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1090]

Prototype

Description

```
char *strchr(const char *s, int c)
```

Locates the first occurrence of character *c* in string *s*. If *c* is found, a pointer to *c* in *s* is returned. Otherwise, a null pointer is returned.

```
char * strrchr(const char *s, int c)
```

Searches from the end of string *s* and locates the last occurrence of character *c* in string *s*. If *c* is found, a pointer to *c* in string *s* is returned. Otherwise, a null pointer is returned.

```
size_t strspn(const char *s1, const char *s2)
```

Determines and returns the length of the initial segment of string *s1* consisting only of characters contained in string *s2*.

```
char *strpbrk(const char *s1, const char *s2)
```

Locates the first occurrence in string *s1* of any character in string *s2*. If a character from string *s2* is found, a pointer to the character in string *s1* is returned. Otherwise, a null pointer is returned.

```
size_t strcspn(const char *s1, const char *s2)
```

Determines and returns the length of the initial segment of string *s1* consisting of characters not contained in string *s2*.

```
char *strchr(const char *s1, const char *s2)
```

Locates the first occurrence in string *s1* of string *s2*. If the string is found, a pointer to the string in *s1* is returned. Otherwise, a null pointer is returned.

### Portability Tip 22.6



Type `size_t` is a system-dependent synonym for either type `unsigned long` or type `unsigned int`.

---

[Page 1090]

Function `strchr` searches for the first occurrence of a character in a string. If the character is found, `strchr` returns a pointer to the character in the string; otherwise, `strchr` returns a null pointer. The program of Fig. 22.29 uses `strchr` (lines 17 and 25) to search for the first occurrences of '`'a'`' and '`'z'`' in the string "This is a test".

**Figure 22.29. String-search function `strchr`.**

(This item is displayed on pages 1090 - 1091 in the print version)

```

1 // Fig. 22.29: fig22_29.cpp
2 // Using strchr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // strchr prototype
8 using std::strchr;
9
10 int main()
11 {
12 const char *string1 = "This is a test";
13 char character1 = 'a';

```

```

14 char character2 = 'z';
15
16 // search for character1 in string1
17 if (strchr(string1, character1) != NULL)
18 cout << '\'' << character1 << "' was found in \""
19 << string1 << "\".\n";
20 else
21 cout << '\'' << character1 << "' was not found in \""
22 << string1 << "\".\n";
23
24 // search for character2 in string1
25 if (strchr(string1, character2) != NULL)
26 cout << '\'' << character2 << "' was found in \""
27 << string1 << "\".\n";
28 else
29 cout << '\'' << character2 << "' was not found in \""
30 << string1 << "\"."
31
32 return 0;
33 } // end main

```

'a' was found in "This is a test".  
 'z' was not found in "This is a test".

[Page 1091]

Function **strcspn** (Fig. 22.30, line 18) determines the length of the initial part of the string in its first argument that does not contain any characters from the string in its second argument. The function returns the length of the segment.

**Figure 22.30. String-search function strcspn.**

```

1 // Fig. 22.30: fig22_30.cpp
2 // Using strcspn.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // strcspn prototype
8 using std::strcspn;
9
10 int main()
11 {
12 const char *string1 = "The value is 3.14159";
13 const char *string2 = "1234567890";
14
15 cout << "string1 = " << string1 << "\nstring2 = " << string2
16 << "\n\nThe length of the initial segment of string1"
17 << "\ncontaining no characters from string2 = "
18 << strcspn(string1, string2) << endl;
19
20 } // end main

```

```

string1 = The value is 3.14159
string2 = 1234567890

The length of the initial segment of string1
containing no characters from string2 = 13

```

Function **strpbrk** searches for the first occurrence in its first string argument of any character in its second string argument. If a character from the second argument is found, **strpbrk** returns a pointer to the character in the first argument; otherwise, **strpbrk** returns a null pointer. Line 16 of Fig. 22.31 locates the first occurrence in **string1** of any character from **string2**.

---

[Page 1092]

**Figure 22.31. String-search function strpbrk.**

```

1 // Fig. 22.31: fig22_31.cpp
2 // Using strpbrk.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // strpbrk prototype
8 using std::strpbrk;
9
10 int main()
11 {
12 const char *string1 = "This is a test";
13 const char *string2 = "beware";
14
15 cout << "Of the characters in \""
16 << *strpbrk(string1, string2) << '\' is the first character "
17 << "to appear in\n\""
18 << string1 << '\'' << endl;
19 return 0;
20 } // end main

```

Of the characters in "beware"  
 'a' is the first character to appear in  
 "This is a test"

Function **strrchr** searches for the last occurrence of the specified character in a string. If the character is found, **strrchr** returns a pointer to the character in the string; otherwise, **strrchr** returns 0. Line 18 of [Fig. 22.32](#) searches for the last occurrence of the character 'z' in the string "A zoo has many animals including zebras".

**Figure 22.32. String-search function **strrchr**.**

(This item is displayed on pages 1092 - 1093 in the print version)

```

1 // Fig. 22.32: fig22_32.cpp
2 // Using strrchr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // strrchr prototype
8 using std::strrchr;
9
10 int main()
11 {
12 const char *string1 = "A zoo has many animals including zebras";
13 char c = 'z';
14
15 cout << "string1 = " << string1 << "\n" << endl;
16 cout << "The remainder of string1 beginning with the\n"
17 << "last occurrence of character '"'
18 << c << "' is: \" " << strrchr(string1, c) << '\"' << endl;
19 return 0;
20 } // end main

```

string1 = A zoo has many animals including zebras

The remainder of string1 beginning with the  
last occurrence of character 'z' is: "zebras"

Function **strspn** (Fig. 22.33, line 18) determines the length of the initial part of the string in its first argument that contains only characters from the string in its second argument. The function returns the length of the segment.

**Figure 22.33. String-search function strspn.**

(This item is displayed on page 1093 in the print version)

```

1 // Fig. 22.33: fig22_33.cpp
2 // Using strspn.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // strspn prototype
8 using std::strspn;
9
10 int main()
11 {
12 const char *string1 = "The value is 3.14159";
13 const char *string2 = "aehilis Tuv";
14
15 cout << "string1 = " << string1 << "\nstring2 = " << string2
16 << "\n\nThe length of the initial segment of string1\n"
17 << "containing only characters from string2 = "
18 << strspn(string1, string2) << endl;
19 return 0;
20 } // end main

```

string1 = The value is 3.14159  
 string2 = aehilis Tuv

The length of the initial segment of string1  
 containing only characters from string2 = 13

Function **strstr** searches for the first occurrence of its second string argument in its first string argument. If the second string is found in the first string, a pointer to the location of the string in the first argument is returned; otherwise, it returns 0. Line 18 of [Fig. 22.34](#) uses **strstr** to find the string "def" in the string "abcdefabcdef".

[Page 1093]

### Figure 22.34. String-search function **strstr**.

(This item is displayed on pages 1093 - 1094 in the print version)

```
1 // Fig. 22.34: fig22_34.cpp
2 // Using strstr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // strstr prototype
8 using std::strstr;
9
10 int main()
11 {
12 const char *string1 = "abcdefabcdef";
13 const char *string2 = "def";
14
15 cout << "string1 = " << string1 << "\nstring2 = " << string2
16 << "\n\nThe remainder of string1 beginning with the\n"
17 << "first occurrence of string2 is: "
18 << strstr(string1, string2) << endl;
19 return 0;
20 } // end main
```

```
string1 = abcdefabcdef
string2 = def
```

```
The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1095]

### Figure 22.36. Memory-handling function `memcpy`.

(This item is displayed on pages 1095 - 1096 in the print version)

```
1 // Fig. 22.36: fig22_36.cpp
2 // Using memcpy.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // memcpy prototype
8 using std::memcpy;
9
10 int main()
11 {
12 char s1[17];
13
14 // 17 total characters (includes terminating null)
15 char s2[] = "Copy this string";
16
17 memcpy(s1, s2, 17); // copy 17 characters from s2 to s1
18
19 cout << "After s2 is copied into s1 with memcpy,\n"
20 << "s1 contains \" " << s1 << '\"' << endl;
21 return 0;
22 } // end main
```

After s2 is copied into s1 with `memcpy`,  
s1 contains "Copy this string"

[Page 1096]

Function `memmove`, like `memcpy`, copies a specified number of bytes from the object pointed to by its second argument into the object pointed to by its first argument. Copying is performed as if the bytes were copied from the second argument to a temporary array of characters, and then copied from the temporary array to the first argument. This allows characters from one part of a string to be copied into another part of the same string.

### Common Programming Error 22.8



String-manipulation functions other than `memmove` that copy characters have undefined results when copying takes place between parts of the same string.

The program in Fig. 22.37 uses `memmove` (line 16) to copy the last 10 bytes of array `x` into the first 10 bytes of array `x`.

**Figure 22.37. Memory-handling function `memmove`.**

```

1 // Fig. 22.37: fig22_37.cpp
2 // Using memmove.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // memmove prototype
8 using std::memmove;
9
10 int main()
11 {
12 char x[] = "Home Sweet Home";
13
14 cout << "The string in array x before memmove is: " << x;
15 cout << "\nThe string in array x after memmove is: "
16 << static_cast< char * >(memmove(x, &x[5], 10)) << endl;
17 return 0;
18 } // end main

```

The string in array x before memmove is: Home Sweet Home  
 The string in array x after memmove is: Sweet Home Home

[Page 1097]

Function `memcmp` (Fig. 22.38, lines 19, 20 and 21) compares the specified number of characters of its first argument with the corresponding characters of its second argument. The function returns a value greater than zero if the first argument is greater than the second argument, zero if the arguments are equal, and a value less than zero if the first argument is less than the second argument. [Note: With some compilers, function `memcmp` returns -1, 0 or 1, as in the sample output of Fig. 22.38. With other compilers, this function returns 0 or the difference between the numeric codes of the first characters that differ in the strings being compared. For example, when `s1` and `s2` are compared, the first character that differs between them is the fifth character of each string (numeric code 69) for `s1` and x (numeric code 72) for `s2`. In this case, the return value will be 19 (or -19 when `s2` is compared to `s1`).]

**Figure 22.38. Memory-handling function `memcmp`.**

```
1 // Fig. 22.38: fig22_38.cpp
2 // Using memcmp.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstring> // memcmp prototype
11 using std::memcmp;
12
13 int main()
14 {
15 char s1[] = "ABCDEFG";
16 char s2[] = "ABCDXYZ";
17
18 cout << "s1 = " << s1 << "\ns2 = " << s2 << endl
19 << "\nmemcmp(s1, s2, 4) = " << setw(3) << memcmp(s1, s2, 4)
20 << "\nmemcmp(s1, s2, 7) = " << setw(3) << memcmp(s1, s2, 7)
21 << "\nmemcmp(s2, s1, 7) = " << setw(3) << memcmp(s2, s1, 7)
22 << endl;
23 return 0;
24 } // end main
```

```
s1 = ABCDEFG
s2 = ABCDXYZ

memcmp(s1, s2, 4) = 0
memcmp(s1, s2, 7) = -1
memcmp(s2, s1, 7) = 1
```

Function `memchr` searches for the first occurrence of a byte, represented as `unsigned char`, in the specified number of bytes of an object. If the byte is found in the object, a pointer to it is returned; otherwise, the function returns a null pointer. Line 16 of Fig. 22.39 searches for the character (byte) '`r`' in the string "This is a string".

---

[Page 1098]

**Figure 22.39. Memory-handling function `memchr`.**

```
1 // Fig. 22.39: fig22_39.cpp
2 // Using memchr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // memchr prototype
8 using std::memchr;
9
10 int main()
11 {
12 char s[] = "This is a string";
13
14 cout << "s = " << s << "\n" << endl;
15 cout << "The remainder of s after character 'r' is found is \""
16 << static_cast< char * >(memchr(s, 'r', 16)) << '\"' << endl;
17 return 0;
18 } // end main
```

```
s = This is a string
```

The remainder of s after character 'r' is found is "ring"

Function `memset` copies the value of the byte in its second argument into a specified number of bytes of the object pointed to by its first argument. Line 16 in Fig. 22.40 uses `memset` to copy 'b' into the first 7 bytes of `string1`.

**Figure 22.40. Memory-handling function `memset`.**

```

1 // Fig. 22.40: fig22_40.cpp
2 // Using memset.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // memset prototype
8 using std::memset;
9
10 int main()
11 {
12 char string1[15] = "BBBBBBBBBBBBBBB";
13
14 cout << "string1 = " << string1 << endl;
15 cout << "string1 after memset = "
16 << static_cast< char * >(memset(string1, 'b', 7)) << endl;
17 return 0;
18 } // end main

```

```

string1 = BBBB BBBB BBBB BBBB
string1 after memset = bbbbbbbBBBBBBB

```



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1099]

## 22.13. Wrap-Up

This chapter introduced `struct` definitions, initializing `structs` and using them with functions. We discussed `typedef`, using it to create aliases to help promote portability. We also introduced bitwise operators to manipulate data and bit fields for storing data compactly. You also learned about the string-conversion functions in `<cstlib>` and the string-processing functions in `<cstring>`. In the next chapter, we continue our discussion of data structures by discussing containers data structures defined in the C++ Standard Template Library. We also present the many algorithms defined in the STL as well.

[◀ PREV](#)[NEXT ▶](#)**page footer**

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1100]

- Bit fields reduce storage use by storing data in the minimum number of bits required. Bit-field members must be declared as `int` or `unsigned`.
- A bit field is declared by following an `unsigned` or `int` member name with a colon and the width of the bit field.
- The bit-field width must be an integer constant.
- If a bit field is specified without a name, the field is used as padding in the structure.
- An unnamed bit field with width 0 aligns the next bit field on a new machine-word boundary.
- Function `islower` determines whether its argument is a lowercase letter (az). Function `isupper` determines whether its argument is an uppercase letter (AZ).
- Function `isdigit` determines whether its argument is a digit (09).
- Function `isalpha` determines whether its argument is an uppercase (AZ) or lowercase letter (a-z).
- Function `isalnum` determines whether its argument is an uppercase letter (AZ), a lowercase letter (az), or a digit (0-9).
- Function `isxdigit` determines whether its argument is a hexadecimal digit (AF, a-f, 0-9).
- Function `toupper` converts a lowercase letter to an uppercase letter. Function `tolower` converts an uppercase letter to a lowercase letter.
- Function `isspace` determines whether its argument is one of the following white-space characters: ' ' (space), '\f', '\n', '\r', '\t' or '\v'.
- Function `iscntrl` determines whether its argument is a control character, such as '\t', '\v', '\f', '\a', '\b', '\r' or '\n'.
- Function `ispunct` determines whether its argument is a printing character other than a space, a digit or a letter.
- Function `isprint` determines whether its argument is any printing character, including space.
- Function `isgraph` determines whether its argument is a printing character other than space.
- Function `atof` converts its argument a string beginning with a series of digits that represents a floating-point number to a double value.
- Function `atoi` converts its argument a string beginning with a series of digits that represents an integer to an int value.
- Function `atol` converts its argument a string beginning with a series of digits that represents a long integer to a long value.
- Function `strtod` converts a sequence of characters representing a floating-point value to double. The function receives two arguments a string (`char *`) and the address of a `char *` pointer. The string contains the character sequence to be converted, and the pointer to `char *` is assigned the remainder of the string after the conversion.
- Function `strtol` converts a sequence of characters representing an integer to long. The function receives three arguments a string (`char *`), the address of a `char *` pointer and an integer. The string contains the character sequence to be converted, the pointer to `char *` is assigned the location of the first character after the converted value and the integer specifies the base of the value being converted.
- Function `strtoul` converts a sequence of characters representing an integer to unsigned long. The function receives three arguments a string (`char *`), the address of a `char *` pointer and an integer. The string contains the character sequence to be converted, the pointer to `char *` is assigned

the location of the first character after the converted value and the integer specifies the base of the value being converted.

[Page 1101]

- Function `strchr` searches for the first occurrence of a character in a string. If the character is found, `strchr` returns a pointer to the character in the string; otherwise, `strchr` returns a null pointer.
- Function `strcspn` determines the length of the initial part of the string in its first argument that does not contain any characters from the string in its second argument. The function returns the length of the segment.
- Function `strpbrk` searches for the first occurrence in its first argument of any character that appears in its second argument. If a character from the second argument is found, `strpbrk` returns a pointer to the character; otherwise, `strpbrk` returns a null pointer.
- Function `strrchr` searches for the last occurrence of a character in a string. If the character is found, `strrchr` returns a pointer to the character in the string; otherwise, it returns a null pointer.
- Function `strspn` determines the length of the initial part of the string in its first argument that contains only characters from the string in its second argument and returns the length of the segment.
- Function `strstr` searches for the first occurrence of its second string argument in its first string argument. If the second string is found in the first string, a pointer to the location of the string in the first argument is returned; otherwise it returns 0.
- Function `memcpy` copies a specified number of characters from the object to which its second argument points into the object to which its first argument points. The function can receive a pointer to any object. The pointers are received by `memcpy` as `void` pointers and converted to `char` pointers for use in the function. Function `memcpy` manipulates the bytes of its argument as characters.
- Function `memmove` copies a specified number of bytes from the object pointed to by its second argument to the object pointed to by its first argument. Copying is accomplished as if the bytes were copied from the second argument to a temporary character array, and then copied from the temporary array to the first argument.
- Function `memcmp` compares the specified number of characters of its first and second arguments.
- Function `memchr` searches for the first occurrence of a byte, represented as `unsigned char`, in the specified number of bytes of an object. If the byte is found, a pointer to it is returned; otherwise, a null pointer is returned.
- Function `memset` copies its second argument, treated as an `unsigned char`, to a specified number of bytes of the object pointed to by the first argument.



page footer



The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1102]

[memchr](#)

[memcmp](#)

[memcpy](#)

[memmove](#)

[memset](#)

[one's complement](#)

[padding](#)

[self-referential structure](#)

[strchr](#)

[strcspn](#)

[string-conversion functions](#)

[strupr](#)

[strrchr](#)

[strspn](#)

[strstr](#)

[strtod](#)

[strtol](#)

[strtoul](#)

`struct`

structure name

structure type

`tolower`

`toupper`

`typedef`

unnamed bit field

width of a bit field

zero-width bit field

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1103]

## 22.4

Find the error in each of the following:

a.

Assume that `struct Card` has been defined as containing two pointers to type `char` namely, `face` and `suit`. Also, the variable `c` has been declared to be of type `Card`, and the variable `cPtr` has been declared to be of type pointer to `Card`. Variable `cPtr` has been assigned the address of `c`.

```
cout << *cPtr.face << endl;
```

b.

Assume that `struct Card` has been defined as containing two pointers to type `char` namely, `face` and `suit`. Also, the array `hearts[ 13 ]` has been declared to be of type `Card`. The following statement should print the member `face` of element 10 of the array.

```
cout << hearts.face << endl;
```

c.

```
struct Person
{
 char lastName[15];
 char firstName[15];
 int age;
}
```

d.

Assume that variable `p` has been declared as type `Person` and that variable `c` has been declared as type `Card`.

```
p = c;
```

## 22.5

Write a single statement to accomplish each of the following. Assume that variables `c` (which stores a character), `x`, `y` and `z` are of type `int`; variables `d`, `e` and `f` are of type `double`; variable `ptr` is of type `char *` and arrays `s1[ 100 ]` and `s2[ 100 ]` are of type `char`.

a.

Convert the character stored in variable `c` to an uppercase letter. Assign the result to variable `c`.

b.

Determine if the value of variable `c` is a digit. Use the conditional operator as shown in Figs. 22.1822.20 to print " is a " or " is not a " when the result is displayed.

c.

Convert the string "1234567" to `long`, and print the value.

d.

Determine whether the value of variable `c` is a control character. Use the conditional operator to print " is a " or " is not a " when the result is displayed.

e.

Assign to `ptr` the location of the last occurrence of `c` in `s1`.

f.

Convert the string "8.63582" to `double`, and print the value.

g.

Determine whether the value of `c` is a letter. Use the conditional operator to print " is a " or " is not a " when the result is displayed.

h.

Assign to `ptr` the location of the first occurrence of `s2` in `s1`.

i.

Determine whether the value of variable `c` is a printing character. Use the conditional operator to

print " is a " or " is not a " when the result is displayed.

j.

Assign to `ptr` the location of the first occurrence in `s1` of any character from `s2`.

k.

Assign to `ptr` the location of the first occurrence of `c` in `s1`.

l.

Convert the string `"-21"` to `int`, and print the value.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1104]

•

False. `typedef` is used to define aliases for previously defined data types.

•

False. Structures are passed to functions by value by default and may be passed by reference.

## 22.3

a.

```
struct Part
{
 int partNumber;
 char partName[26];
};
```

b.

```
typedef part * partptr
```

c.

```
Part a;
Part b[10];
Part *ptr;
```

d.

```
cin >> a.partNumber >> a.partName;
```

e.

```
b[3] = a;
```

f.

```
ptr = b;
```

**g.**

```
cout << (ptr + 3)->partNumber << ' '
<< (ptr + 3)->partName << endl;
```

## 22.4

**a.**

Error: The parentheses that should enclose \*cPtr have been omitted, causing the order of evaluation of the expression to be incorrect.

**b.**

Error: The array subscript has been omitted. The expression should be hearts[ 10 ].face.

**c.**

Error: A semicolon is required to end a structure definition.

**d.**

Error: Variables of different structure types cannot be assigned to one another.

## 22.5

**a.**

```
c = toupper(c);
```

**b.**

```
cout << '\'' << c << '\' '
<< (isdigit(c) ? "is a" : "is not a")
<< " digit" << endl;
```

**c.**

```
cout << atol("1234567") << endl;
```

**d.**

```
cout << '\'' << c << "\' "
<< (iscntrl(c) ? "is a" : "is not a")
<< " control character" << endl;
```

**e.**

```
ptr = strrchr(s1, c);
```

**f.**

```
out << atof("8.63582") << endl;
```

**g.**

```
cout << '\'' << c << "\' "
<< (isalpha(c) ? "is a" : "is not a")
<< " letter" << endl;
```

**h.**

```
ptr = strstr(s1, s2);
```

**i.**

```
cout << '\'' << c << "\' "
<< (isprint(c) ? "is a" : "is not a")
<< " printing character" << endl;
```

**j.**

```
ptr = strpbrk(s1, s2);
```

**k.**

```
ptr = strchr(s1, c);
```

**l.**

```
cout << atoi("-21") << endl;
```



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1105]

•

A structure called `Address` that contains character arrays `streetAddress[ 25 ]`, `city[ 20 ]`, `state[ 3 ]` and `zipCode[ 6 ]`.

•

Structure `Student`, containing arrays `firstName[ 15 ]` and `lastName[ 15 ]` and variable `homeAddress` of type struct `Address` from part (b).

•

Structure `Test`, containing 16 bit fields with widths of 1 bit. The names of the bit fields are the letters `a` to `p`.

## 22.7

Consider the following structure definitions and variable declarations:

```
struct Customer {
 char lastName[15];
 char firstName[15];
 int customerNumber;

 struct {
 char phoneNumber[11];
 char address[50];
 char city[15];
 char state[3];
 char zipCode[6];
 } personal;

} customerRecord, *customerPtr;

customerPtr = &customerRecord;
```

Write a separate expression that accesses the structure members in each of the following parts:

a.

Member lastName of structure customerRecord.

b.

Member lastName of the structure pointed to by customerPtr.

c.

Member firstName of structure customerRecord.

d.

Member firstName of the structure pointed to by customerPtr.

e.

Member customerNumber of structure customerRecord.

f.

Member customerNumber of the structure pointed to by customerPtr.

g.

Member phoneNumber of member personal of structure customerRecord.

h.

Member phoneNumber of member personal of the structure pointed to by customerPtr.

i.

Member address of member personal of structure customerRecord.

j.

Member address of member personal of the structure pointed to by customerPtr.

k.

Member city of member personal of structure customerRecord.

**I.**

Member `city` of member `personal` of the structure pointed to by `customerPtr`.

**m.**

Member `state` of member `personal` of structure `customerRecord`.

**n.**

Member `state` of member `personal` of the structure pointed to by `customerPtr`.

**o.**

Member `zipCode` of member `personal` of structure `customerRecord`.

**p.**

Member `zipCode` of member `personal` of the structure pointed to by `customerPtr`.

## 22.8

Modify the program of Fig. 22.14 to shuffle the cards using a high-performance shuffle, as shown in Fig. 22.3. Print the resulting deck in two-column format, as in Fig. 22.4. Precede each card with its color.

## 22.9

Write a program that right-shifts an integer variable 4 bits. The program should print the integer in bits before and after the shift operation. Does your system place zeros or ones in the vacated bits?

## 22.10

Left-shifting an `unsigned` integer by 1 bit is equivalent to multiplying the value by 2. Write function `power2` that takes two integer arguments, `number` and `pow`, and calculates

```
number * 2pow
```

Use a shift operator to calculate the result. The program should print the values as integers and as bits.

## 22.11

The left-shift operator can be used to pack two character values into a two-byte unsigned integer variable. Write a program that inputs two characters from the keyboard and passes them to function `packCharacters`. To pack two characters into an unsigned integer variable, assign the first character to the unsigned variable, shift the unsigned variable left by 8 bit positions and combine the unsigned variable with the second character using the bitwise inclusive-OR operator. The program should output the characters in their bit format before and after they are packed into the unsigned integer to prove that they are in fact packed correctly in the unsigned variable.

## 22.12

Using the right-shift operator, the bitwise AND operator and a mask, write function `unpackCharacters` that takes the unsigned integer from [Exercise 22.11](#) and unpacks it into two characters. To unpack two characters from an unsigned two-byte integer, combine the unsigned integer with the mask 65280 (11111111 00000000) and right-shift the result 8 bits. Assign the resulting value to a `char` variable. Then, combine the unsigned integer with the mask 255 (00000000 11111111). Assign the result to another `char` variable. The program should print the unsigned integer in bits before it is unpacked, then print the characters in bits to confirm that they were unpacked correctly.

## 22.13

If your system uses four-byte integers, rewrite the program of [Exercise 22.11](#) to pack four characters.

## 22.14

If your system uses four-byte integers, rewrite the function `unpackCharacters` of [Exercise 22.12](#) to unpack four characters. Create the masks you need to unpack the 4 characters by left-shifting the value 255 in the mask variable by 8 bits 0, 1, 2 or 3 times (depending on the byte you are unpacking).

## 22.15

Write a program that reverses the order of the bits in an unsigned integer value. The program should input the value from the user and call function `reverseBits` to print the bits in reverse order. Print the value in bits both before and after the bits are reversed to confirm that the bits are reversed properly.

## 22.16

Write a program that demonstrates passing an array by value. [Hint: Use a `struct`.] Prove that a copy was passed by modifying the array copy in the called function.

## 22.17

Write a program that inputs a character from the keyboard and tests the character with each function in

the character-handling library. Print the value returned by each function.

## 22.18

The following program uses function `multiple` to determine whether the integer entered from the keyboard is a multiple of some integer `x`. Examine function `multiple`, then determine the value of `x`.

```
1 // Exercise 22.19: ex22_19.cpp
2 // This program determines if a value is a multiple of X.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 bool multiple(int);
10
11 int main()
12 {
13 int y;
14
15 cout << "Enter an integer between 1 and 32000: ";
16 cin >> y;
```

---

[Page 1107]

```
17
18 if (multiple(y))
19 cout << y << " is a multiple of X" << endl;
20 else
21 cout << y << " is not a multiple of X" << endl;
22
23 return 0;
24
25 } // end main
26
27 // determine if num is a multiple of X
28 bool multiple(int num)
29 {
30 bool mult = true;
31
32 for (int i = 0, mask = 1; i < 10; i++, mask <= 1)
33
34 if ((num & mask) != 0) {
35 mult = false;
36 break;
37
38 } // end if
```

```

39 return mult;
40
41 } // end function multiple

```

## 22.19

What does the following program do?

```

1 // Exercise 22.20: ex22_20.cpp
2 #include <iostream>
3
4 using std::cout;
5 using std::cin;
6 using std::endl;
7 using std::boolalpha;
8
9 bool mystery(unsigned);
10
11 int main()
12 {
13 unsigned x;
14
15 cout << "Enter an integer: ";
16 cin >> x;
17 cout << boolalpha
18 << "The result is " << mystery(x) << endl;
19
20 return 0;
21
22 } // end main
23

```

[Page 1108]

```

24 // What does this function do?
25 bool mystery(unsigned bits)
26 {
27 const int SHIFT = 8 * sizeof(unsigned) - 1;
28 const unsigned MASK = 1 << SHIFT;
29 unsigned total = 0;
30
31 for (int i = 0; i < SHIFT + 1; i++, bits <<= 1)
32
33 if ((bits & MASK) == MASK)
34 ++total;
35
36 return !(total % 2);

```

```
37
38 } // end function mystery
```

## 22.20

Write a program that inputs a line of text with `istream` member function `getline` (as in [Chapter 15](#)) into character array `s[ 100 ]`. Output the line in uppercase letters and lowercase letters.

## 22.21

Write a program that inputs four strings that represent integers, converts the strings to integers, sums the values and prints the total of the four values. Use only the C-style string-processing techniques shown in this chapter.

## 22.22

Write a program that inputs four strings that represent floating-point values, converts the strings to double values, sums the values and prints the total of the four values. Use only the C-style string-processing techniques shown in this chapter.

## 22.23

Write a program that inputs a line of text and a search string from the keyboard. Using function `strstr`, locate the first occurrence of the search string in the line of text, and assign the location to variable `searchPtr` of type `char *`. If the search string is found, print the remainder of the line of text beginning with the search string. Then use `strstr` again to locate the next occurrence of the search string in the line of text. If a second occurrence is found, print the remainder of the line of text beginning with the second occurrence. [Hint: The second call to `strstr` should contain the expression `searchPtr + 1` as its first argument.]

## 22.24

Write a program based on the program of [Exercise 22.23](#) that inputs several lines of text and a search string, then uses function `strstr` to determine the total number of occurrences of the string in the lines of text. Print the result.

## 22.25

Write a program that inputs several lines of text and a search character and uses function `strchr` to determine the total number of occurrences of the character in the lines of text.

## 22.26

Write a program based on the program of [Exercise 22.25](#) that inputs several lines of text and uses function `strchr` to determine the total number of occurrences of each letter of the alphabet in the text. Uppercase and lowercase letters should be counted together. Store the totals for each letter in an array, and print the values in tabular format after the totals have been determined.

## 22.27

The chart in [Appendix B](#) shows the numeric code representations for the characters in the ASCII character set. Study this chart, and then state whether each of the following is true or false:

a.

The letter "A" comes before the letter "B."

b.

The digit "9" comes before the digit "0."

c.

The commonly used symbols for addition, subtraction, multiplication and division all come before any of the digits.

d.

The digits come before the letters.

e.

If a sort program sorts strings into ascending sequence, then the program will place the symbol for a right parenthesis before the symbol for a left parenthesis.

---

[Page 1109]

## 22.28

Write a program that reads a series of strings and prints only those strings beginning with the letter "b."

## 22.29

Write a program that reads a series of strings and prints only those strings that end with the letters "ED."

## 22.30

Write a program that inputs an ASCII code and prints the corresponding character. Modify this program so that it generates all possible three-digit codes in the range 000255 and attempts to print the corresponding characters. What happens when this program is run?

### 22.31

Using the ASCII character chart in [Appendix B](#) as a guide, write your own versions of the character-handling functions in [Fig. 22.17](#).

### 22.32

Write your own versions of the functions in [Fig. 22.21](#) for converting strings to numbers.

### 22.33

Write your own versions of the functions in [Fig. 22.28](#) for searching strings.

### 22.34

Write your own versions of the functions in [Fig. 22.35](#) for manipulating blocks of memory.

### 22.35

(Project: A Spelling Checker) Many popular word-processing software packages have builtin spell checkers. We used spell-checking capabilities in preparing this book and discovered that, no matter how careful we thought we were in writing a chapter, the software was always able to find a few more spelling errors than we were able to catch manually.

In this project, you are asked to develop your own spell-checker utility. We make suggestions to help get you started. You should then consider adding more capabilities. You might find it helpful to use a computerized dictionary as a source of words.

Why do we type so many words with incorrect spellings? In some cases, it is because we simply do not know the correct spelling, so we make a "best guess." In some cases, it is because we transpose two letters (e.g., "defualt" instead of "default"). Sometimes we double-type a letter accidentally (e.g., "hanndy" instead of "handy"). Sometimes we type a nearby key instead of the one we intended (e.g., "biryhday" instead of "birthday"). And so on.

Design and implement a spell-checker program. Your program maintains an array `wordList` of character strings. You can either enter these strings or obtain them from a computerized dictionary.

Your program asks a user to enter a word. The program then looks up that word in the `wordList` array. If the word is present in the array, your program should print "Word is spelled correctly."

If the word is not present in the array, your program should print "Word is not spelled correctly." Then your program should try to locate other words in `wordList` that might be the word the user intended to type. For example, you can try all possible single transpositions of adjacent letters to discover that the word "default" is a direct match to a word in `wordList`. Of course, this implies that your program will check all other single transpositions, such as "edfault," "dfeault," "deafult," "defalut" and "default." When you find a new word that matches one in `wordList`, print that word in a message such as "Did you mean "default?"."

Implement other tests, such as the replacing of each double letter with a single letter and any other tests you can develop to improve the value of your spell checker.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1111]

## Outline

### [23.1 Introduction to the Standard Template Library \(STL\)](#)

#### [23.1.1 Introduction to Containers](#)

#### [23.1.2 Introduction to Iterators](#)

#### [23.1.3 Introduction to Algorithms](#)

### [23.2 Sequence Containers](#)

#### [23.2.1 `vector` Sequence Container](#)

#### [23.2.2 `list` Sequence Container](#)

#### [23.2.3 `deque` Sequence Container](#)

### [23.3 Associative Containers](#)

#### [23.3.1 `multiset` Associative Container](#)

#### [23.3.2 `set` Associative Container](#)

#### [23.3.3 `multimap` Associative Container](#)

#### [23.3.4 `map` Associative Container](#)

### [23.4 Container Adapters](#)

#### [23.4.1 `stack` Adapter](#)

#### [23.4.2 `queue` Adapter](#)

#### [23.4.3 `priority\_queue` Adapter](#)

## 23.5 Algorithms

23.5.1 `fill`, `fill_n`, `generate` and `generate_n`

23.5.2 `equal`, `mismatch` and `lexicographical_compare`

23.5.3 `remove`, `remove_if`, `remove_copy` and `remove_copy_if`

23.5.4 `replace`, `replace_if`, `replace_copy` and `replace_copy_if`

23.5.5 Mathematical Algorithms

23.5.6 Basic Searching and Sorting Algorithms

23.5.7 `swap`, `iter_swap` and `swap_ranges`

23.5.8 `copy_backward`, `merge`, `unique` and `reverse`

23.5.9 `inplace_merge`, `unique_copy` and `reverse_copy`

23.5.10 Set Operations

23.5.11 `lower_bound`, `upper_bound` and `equal_range`

23.5.12 Heapsort

23.5.13 `min` and `max`

23.5.14 STL Algorithms Not Covered in This Chapter

23.6 Class `bitset`

23.7 Function Objects

23.8 Wrap-Up

23.9 STL Internet and Web Resources

Summary

## Terminology

### Self-Review Exercises

### Answers to Self-Review Exercises

### Exercises

### Recommended Readings

 PREV

NEXT 

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1113]

## Software Engineering Observation 23.1



The STL approach allows general programs to be written so that the code does not depend on the underlying container. Such a programming style is called generic programming.

In [Chapter 21](#), we studied data structures. We built linked lists, queues, stacks and trees. We carefully wove link objects together with pointers. Pointer-based code is complex, and the slightest omission or oversight can lead to serious memory-access violations and memory-leak errors with no compiler complaints. Implementing additional data structures, such as deques, priority queues, sets and maps, requires substantial extra work. In addition, if many programmers on a large project implement similar containers and algorithms for different tasks, the code becomes difficult to modify, maintain and debug. An advantage of the STL is that programmers can reuse the STL containers, iterators and algorithms to implement common data representations and manipulations. This reuse can save substantial development time, money and effort.

## Software Engineering Observation 23.2



Avoid reinventing the wheel; program with the reusable components of the C++ Standard Library. STL includes many of the most popular data structures as containers and provides various popular algorithms to process data in these containers.

## Error-Prevention Tip 23.1



When programming pointer-based data structures and algorithms, we must do our own debugging and testing to be sure our data structures, classes and algorithms function properly. It is easy to make errors when manipulating pointers at this low level. Memory leaks and memory-access violations are common in such custom code. For most programmers, and for most of the applications they will need to write, the prepackaged, templated containers of the STL are sufficient. Using the STL helps programmers reduce testing and debugging time. One caution is that, for large projects, template compile time can be significant.

This chapter introduces the STL. It is by no means complete or comprehensive. However, it is a friendly, accessible chapter that should convince you of the value of the STL in software reuse and encourage further study.

### 23.1.1. Introduction to Containers

The STL container types are shown in Fig. 23.1. The containers are divided into three major categories **sequence containers**, **associative containers** and **container adapters**.

[Page 1114]

**Figure 23.1. Standard Library container classes.**

Standard Library container class	Description
Sequence containers	
vector	rapid insertions and deletions at back direct access to any element
deque	rapid insertions and deletions at front or back direct access to any element
list	doubly linked list, rapid insertion and deletion anywhere
Associative containers	
set	rapid lookup, no duplicates allowed
multiset	rapid lookup, duplicates allowed
map	one-to-one mapping, no duplicates allowed, rapid key-based lookup
multimap	one-to-many mapping, duplicates allowed, rapid key-based lookup
Container adapters	
stack	last-in, first-out (LIFO)
queue	first-in, first-out (FIFO)
priority_queue	highest-priority element is always the first element out

## STL Containers Overview

The sequence containers (also referred to as [sequential containers](#)) represent linear data structures, such as vectors and linked lists. Associative containers are nonlinear containers that typically can locate elements stored in the containers quickly. Such containers can store sets of values or [key/value pairs](#). The sequence containers and associative containers are collectively referred to as the first-class containers. As we saw in [Chapter 21](#), stacks and queues actually are constrained versions of sequential containers. For this reason, STL implements stacks and queues as container adapters that enable a program to view a sequential container in a constrained manner. There are four other container types that are considered "near-containers" C-like pointer-based arrays (discussed in [Chapter 7](#)), strings (discussed in [Chapter 18](#)), bitsets for maintaining sets of flag values and valarrays for performing high-speed mathematical vector operations (this last class is optimized for computation performance and is not as flexible as the first-class containers). These four types are considered "near containers" because they exhibit capabilities similar to those of the first-class containers, but do not support all the first-class-container capabilities.

### STL Container Common Functions

All STL containers provide similar functionality. Many generic operations, such as member function `size`, apply to all containers, and other operations apply to subsets of similar containers. This encourages extensibility of the STL with new classes. [Figure 23.2](#) describes the functions common to all Standard Library containers. [Note: Overloaded operators `operator<`, `operator<=`, `operator>`, `operator>=`, `operator==` and `operator!=` are not provided for `priority_queues`.]

---

[Page 1115]

**Figure 23.2. STL container common functions.**

Common member functions for all STL containers	Description
default constructor	A constructor to provide a default initialization of the container. Normally, each container has several constructors that provide different initialization methods for the container.
copy constructor	A constructor that initializes the container to be a copy of an existing container of the same type.
destructor	Destructor function for cleanup after a container is no longer needed.

<code>empty</code>	Returns <code>true</code> if there are no elements in the container; otherwise, returns <code>false</code> .
<code>size</code>	Returns the number of elements currently in the container.
<code>operator=</code>	Assigns one container to another.
<code>operator&lt;</code>	Returns <code>true</code> if the first container is less than the second container; otherwise, returns <code>false</code> .
<code>operator&lt;=</code>	Returns <code>TRue</code> if the first container is less than or equal to the second container; otherwise, returns <code>false</code> .
<code>operator&gt;</code>	Returns <code>true</code> if the first container is greater than the second container; otherwise, returns <code>false</code> .
<code>operator&gt;=</code>	Returns <code>true</code> if the first container is greater than or equal to the second container; otherwise, returns <code>false</code> .
<code>operator==</code>	Returns <code>true</code> if the first container is equal to the second container; otherwise, returns <code>false</code> .
<code>operator!=</code>	Returns <code>TRue</code> if the first container is not equal to the second container; otherwise, returns <code>false</code> .
<code>swap</code>	Swaps the elements of two containers.

### Functions found only in first-class containers

<code>max_size</code>	Returns the maximum number of elements for a container.
<code>begin</code>	The two versions of this function return either an <code>iterator</code> or a <code>const_iterator</code> that refers to the first element of the container.
<code>end</code>	The two versions of this function return either an <code>iterator</code> or a <code>const_iterator</code> that refers to the next position after the end of the container.
<code>rbegin</code>	The two versions of this function return either a <code>reverse_iterator</code> or a <code>const_reverse_iterator</code> that refers to the last element of the container.
<code>rend</code>	The two versions of this function return either a <code>reverse_iterator</code> or a <code>const_reverse_iterator</code> that refers to the next position after the last element of the reversed container.
<code>erase</code>	Erases one or more elements from the container.
<code>clear</code>	Erases all elements from the container.

## STL Container Header Files

The header files for each of the Standard Library containers are shown in Fig. 23.3. The contents of these header files are all in namespace std.<sup>[1]</sup>

[1] Some older C++ compilers do not support the new-style header files. Many of these compilers provide their own versions of the header-file names. See your compiler documentation for more information on the STL support your compiler provides.

**Figure 23.3. Standard Library container header files.**

Standard Library container header files	
<vector>	
<list>	
<deque>	
<queue>	Contains both queue and priority_queue.
<stack>	
<map>	Contains both map and multimap.
<set>	Contains both set and multiset.
<bitset>	

## First-Class Container Common `typedefs`

Figure 23.4 shows the common `typedefs` (to create synonyms or aliases for lengthy type names) found in first-class containers. These `typedefs` are used in generic declarations of variables, parameters to functions and return values from functions. For example, `value_type` in each container is always a `typedef` that represents the type of value stored in the container.

**Figure 23.4. `typedefs` found in first-class containers.**

---

[Page 1117]

<b>typedef</b>	<b>Description</b>
<code>value_type</code>	The type of element stored in the container.
<code>reference</code>	A reference to the type of element stored in the container.
<code>const_reference</code>	A constant reference to the type of element stored in the container. Such a reference can be used only for reading elements in the container and for performing <code>const</code> operations.
<code>pointer</code>	A pointer to the type of element stored in the container.
<code>iterator</code>	An iterator that points to the type of element stored in the container.
<code>const_iterator</code>	A constant iterator that points to the type of element stored in the container and can be used only to read elements.
<code>reverse_iterator</code>	A reverse iterator that points to the type of element stored in the container. This type of iterator is for iterating through a container in reverse.
<code>const_reverse_iterator</code>	A constant reverse iterator that points to the type of element stored in the container and can be used only to read elements. This type of iterator is for iterating through a container in reverse.
<code>difference_type</code>	The type of the result of subtracting two iterators that refer to the same container (operator <code>-</code> is not defined for iterators of lists and associative containers).
<code>size_type</code>	The type used to count items in a container and index through a sequence container (cannot index through a <code>list</code> ).

---

[Page 1117]

### Performance Tip 23.3



STL generally avoids inheritance and virtual functions in favor of using generic programming with templates to achieve better execution-time performance.

## Portability Tip 23.1



Programming with STL will enhance the portability of your code.

When preparing to use an STL container, it is important to ensure that the type of element being stored in the container supports a minimum set of functionality. When an element is inserted into a container, a copy of that element is made. For this reason, the element type should provide its own copy constructor and assignment operator. [Note: This is required only if default memberwise copy and default memberwise assignment do not perform proper copy and assignment operations for the element type.] Also, the associative containers and many algorithms require elements to be compared. For this reason, the element type should provide an equality operator (==) and a less-than operator (<).

### Software Engineering Observation 23.3



The STL containers technically do not require their elements to be comparable with the equality and less-than operators unless a program uses a container member function that must compare the container elements (e.g., the sort function in class list). Unfortunately, some prestandard C++ compilers are not capable of ignoring parts of a template that are not used in a particular program. On compilers with this problem, you may not be able to use the STL containers with objects of classes that do not define overloaded less-than and equality operators.

#### 23.1.2. Introduction to Iterators

Iterators have many features in common with pointers and are used to point to the elements of first-class containers (and for a few other purposes, as we will see). Iterators hold state information sensitive to the particular containers on which they operate; thus, iterators are implemented appropriately for each type of container. Certain iterator operations are uniform across containers. For example, the dereferencing operator (\*) dereferences an iterator so that you can use the element to which it points. The ++ operation on an iterator moves it to the next element of the container (much as incrementing a pointer into an array aims the pointer at the next element of the array).

---

[Page 1118]

STL first-class containers provide member functions `begin` and `end`. Function `begin` returns an iterator pointing to the first element of the container. Function `end` returns an iterator pointing to the first element past the end of the container (an element that doesn't exist). If iterator `i` points to a particular element, then `++i` points to the "next" element and `*i` refers to the element pointed to by `i`. The iterator resulting

from `end` can be used only in an equality or inequality comparison to determine whether the "moving iterator" (`i` in this case) has reached the end of the container.

We use an object of type `iterator` to refer to a container element that can be modified. We use an object of type `const_iterator` to refer to a container element that cannot be modified.

## Using `istream_iterator` for Input and Using `ostream_iterator` for Output

We use iterators with **sequences** (also called **ranges**). These sequences can be in containers, or they can be **input sequences** or **output sequences**. The program of Fig. 23.5 demonstrates input from the standard input (a sequence of data for input into a program), using an `istream_iterator`, and output to the standard output (a sequence of data for output from a program), using an `ostream_iterator`. The program inputs two integers from the user at the keyboard and displays the sum of the integers. [2]

[2] The examples in this chapter precede each use of an STL function and each definition of an STL container object with the "std:::" prefix rather than placing the `using` declarations or directives at the beginning of the program, as was shown in most prior examples. Differences in compilers and the complex code generated when using STL make it difficult to construct a proper set of `using` declarations or directives that enable the programs to compile without errors. To allow these programs to compile on the widest variety of platforms, we chose the "std:::" prefix approach.

**Figure 23.5. Input and output stream iterators.**

(This item is displayed on page 1119 in the print version)

```

1 // Fig. 23.5: Fig23_05.cpp
2 // Demonstrating input and output with iterators.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iterator> // ostream_iterator and istream_iterator
9
10 int main()
11 {
12 cout << "Enter two integers: ";
13
14 // create istream_iterator for reading int values from cin
15 std::istream_iterator< int > inputInt(cin);
16
17 int number1 = *inputInt; // read int from standard input
18 ++inputInt; // move iterator to next input value
19 int number2 = *inputInt; // read int from standard input

```

```

20
21 // create ostream_iterator for writing int values to cout
22 std::ostream_iterator< int > outputInt(cout);
23
24 cout << "The sum is: ";
25 *outputInt = number1 + number2; // output result to cout
26 cout << endl;
27 return 0;
28 } // end main

```

Enter two integers: 12 25  
 The sum is: 37

Line 15 creates an `istream_iterator` that is capable of extracting (inputting) `int` values in a type-safe manner from the standard input object `cin`. Line 17 dereferences iterator `inputInt` to read the first integer from `cin` and assigns that integer to `number1`. Note that the dereferencing operator `*` applied to `inputInt` gets the value from the stream associated with `inputInt`; this is similar to dereferencing a pointer. Line 18 positions iterator `inputInt` to the next value in the input stream. Line 19 inputs the next integer from `inputInt` and assigns it to `number2`.

Line 22 creates an `ostream_iterator` that is capable of inserting (outputting) `int` values in the standard output object `cout`. Line 25 outputs an integer to `cout` by assigning to `*outputInt` the sum of `number1` and `number2`. Notice the use of the dereferencing operator `*` to use `*outputInt` as an lvalue in the assignment statement. If you want to output another value using `outputInt`, the iterator must be incremented with `++` (both the prefix and postfix increment can be used, but the prefix form should be preferred for performance reasons.).

### Error-Prevention Tip 23.2



The `*` (dereferencing) operator of any `const` iterator returns a `const` reference to the container element, disallowing the use of non-`const` member functions.



Attempting to dereference an iterator positioned outside its container is a runtime logic error. In particular, the iterator returned by `end` cannot be dereferenced or incremented.

## Common Programming Error 23.2



Attempting to create a `non-const` iterator for a `const` container results in a compilation error.

## Iterator Categories and Iterator Category Hierarchy

[Figure 23.6](#) shows the categories of iterators used by the STL. Each category provides a specific set of functionality. [Figure 23.7](#) illustrates the hierarchy of iterator categories. As you follow the hierarchy from top to bottom, each iterator category supports all the functionality of the categories above it in the figure. Thus the "weakest" iterator types are at the top and the most powerful one is at the bottom. Note that this is not an inheritance hierarchy.

**Figure 23.6. Iterator categories.**

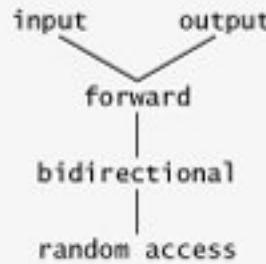
(This item is displayed on page 1120 in the print version)

Category	Description
input	Used to read an element from a container. An input iterator can move only in the forward direction (i.e., from the beginning of the container to the end) one element at a time. Input iterators support only one-pass algorithms-the same input iterator cannot be used to pass through a sequence twice.
output	Used to write an element to a container. An output iterator can move only in the forward direction one element at a time. Output iterators support only one-pass algorithms-the same output iterator cannot be used to pass through a sequence twice.
forward	Combines the capabilities of input and output iterators and retains their position in the container (as state information).

bidirectional	Combines the capabilities of a forward iterator with the ability to move in the backward direction (i.e., from the end of the container toward the beginning). Bidirectional iterators support multipass algorithms.
random access	Combines the capabilities of a bidirectional iterator with the ability to directly access any element of the container, i.e., to jump forward or backward by an arbitrary number of elements.

**Figure 23.7. Iterator category hierarchy.**

(This item is displayed on page 1120 in the print version)

[\[View full size image\]](#)

The iterator category that each container supports determines whether that container can be used with specific algorithms in the STL. Containers that support random-access iterators can be used with all algorithms in the STL. As we will see, pointers into arrays can be used in place of iterators in most STL algorithms, including those that require random-access iterators. [Figure 23.8](#) shows the iterator category of each of the STL containers. Note that only vectors, deques, lists, sets, multisets, maps and multimaps (i.e., the first-class containers) are traversable with iterators.

[\[Page 1120\]](#)**Figure 23.8. Iterator types supported by each Standard Library container.**

(This item is displayed on page 1121 in the print version)

Container	Type of iterator supported

### Sequence containers (first class)

vector	random access
deque	random access
list	bidirectional

### Associative containers (first class)

set	bidirectional
multiset	bidirectional
map	bidirectional
multimap	bidirectional

### Container adapters

stack	no iterators supported
queue	no iterators supported
priority_queue	no iterators supported

### Software Engineering Observation 23.4



Using the "weakest iterator" that yields acceptable performance helps produce maximally reusable components. For example, if an algorithm requires only forward iterators, it can be used with any container that supports forward iterators, *bidirectional iterators* or random-access iterators. However, an algorithm that requires random-access iterators can be used only with containers that have random-access iterators.

## Predefined Iterator `typedefs`

Figure 23.9 shows the predefined iterator `typedefs` that are found in the class definitions of the STL containers. Not every `typedef` is defined for every container. We use `const` versions of the iterators for traversing read-only containers. We use reverse iterators to traverse containers in the reverse direction.

---

[Page 1121]

**Figure 23.9. Iterator `typedefs`.**

Predefined <code>typedefs</code> for iterator types	Direction of <code>++</code>	Capability
<code>iterator</code>	forward	read/write
<code>const_iterator</code>	forward	read
<code>reverse_iterator</code>	backward	read/write
<code>const_reverse_iterator</code>	backward	read

### Error-Prevention Tip 23.3



Operations performed on a `const_iterator` return `const` references to prevent modification to elements of the container being manipulated. Using `const_iterators` in preference to `iterators` where appropriate is another example of the principle of least privilege.

## Iterator Operations

Figure 23.10 shows some operations that can be performed on each iterator type. Note that the operations for each iterator type include all operations preceding that type in the figure. Note also that, for input iterators and output iterators, it is not possible to save the iterator and then use the saved value later.

---

[Page 1122]

**Figure 23.10. Iterator operations for each type of iterator.**

<b>Iterator operation</b>	<b>Description</b>
<b>All iterators</b>	
<code>++p</code>	Preincrement an iterator.
<code>p++</code>	Postincrement an iterator.
<b>Input iterators</b>	
<code>*p</code>	Dereference an iterator.
<code>p = p1</code>	Assign one iterator to another.
<code>p == p1</code>	Compare iterators for equality.
<code>p != p1</code>	Compare iterators for inequality.
<b>Output iterators</b>	
<code>*p</code>	Dereference an iterator.
<code>p = p1</code>	Assign one iterator to another.
<b>Forward iterators</b>	Forward iterators provide all the functionality of both input iterators and output iterators.
<b>Bidirectional iterators</b>	
<code>--p</code>	Predecrement an iterator.
<code>p--</code>	Postdecrement an iterator.
<b>Random-access iterators</b>	
<code>p += i</code>	Increment the iterator <code>p</code> by <code>i</code> positions.
<code>p -= i</code>	Decrement the iterator <code>p</code> by <code>i</code> positions.
<code>p + i</code>	Expression value is an iterator positioned at <code>p</code> incremented by <code>i</code> positions.
<code>p - i</code>	Expression value is an iterator positioned at <code>p</code> decremented by <code>i</code> positions.
<code>p[ i ]</code>	Return a reference to the element offset from <code>p</code> by <code>i</code> positions
<code>p &lt; p1</code>	Return <code>true</code> if iterator <code>p</code> is less than iterator <code>p1</code> (i.e., iterator <code>p</code> is before iterator <code>p1</code> in the container); otherwise, return <code>false</code> .

<code>p &lt;= p1</code>	Return <code>TRue</code> if iterator <code>p</code> is less than or equal to iterator <code>p1</code> (i.e., iterator <code>p</code> is before iterator <code>p1</code> or at the same location as iterator <code>p1</code> in the container); otherwise, return <code>false</code> .
<code>p &gt; p1</code>	Return <code>TRue</code> if iterator <code>p</code> is greater than iterator <code>p1</code> (i.e., iterator <code>p</code> is after iterator <code>p1</code> in the container); otherwise, return <code>false</code> .
<code>p &gt;= p1</code>	Return <code>TRue</code> if iterator <code>p</code> is greater than or equal to iterator <code>p1</code> (i.e., iterator <code>p</code> is after iterator <code>p1</code> or at the same location as iterator <code>p1</code> in the container); otherwise, return <code>false</code> .

### 23.1.3. Introduction to Algorithms

The STL provides algorithms that can be used generically across a variety of containers. STL provides many algorithms you will use frequently to manipulate containers. Inserting, deleting, searching, sorting and others are appropriate for some or all of the STL containers.

---

[Page 1123]

The STL includes approximately 70 standard algorithms. We provide live-code examples of most of these and summarize the others in tables. The algorithms operate on container elements only indirectly through iterators. Many algorithms operate on sequences of elements defined by pairs of iterators a first iterator pointing to the first element of the sequence and a second iterator pointing to one element past the last element of the sequence. Also, it is possible to create your own new algorithms that operate in a similar fashion so they can be used with the STL containers and iterators.

Algorithms often return iterators that indicate the results of the algorithms. Algorithm `find`, for example, locates an element and returns an iterator to that element. If the element is not found, `find` returns the "one past the end" iterator that was passed in to define the end of the range to be searched, which can be tested to determine whether an element was not found. The `find` algorithm can be used with any first-class STL container. STL algorithms create yet another opportunity for reuseusing the rich collection of popular algorithms can save programmers much time and effort.

If an algorithm uses less powerful iterators, the algorithm can also be used with containers that support more powerful iterators. Some algorithms demand powerful iterators; e.g., `sort` demands random-access iterators.

Software Engineering Observation 23.5



The STL is implemented concisely. Until now, class designers would have associated the algorithms with the containers by making the algorithms member functions of the containers. The STL takes a different approach. The algorithms are separated from the containers and operate on elements of the containers only indirectly through iterators. This separation makes it easier to write generic algorithms applicable to many container classes.

### Software Engineering Observation 23.6



The STL is extensible. It is straightforward to add new algorithms and to do so without changes to STL containers.

### Software Engineering Observation 23.7



STL algorithms can operate on STL containers and on pointer-based, C-like arrays.

### Portability Tip 23.2



Because STL algorithms process containers only indirectly through iterators, one algorithm can often be used with many different containers.

Figure 23.11 shows many of the **mutating-sequence algorithms**i.e., the algorithms that result in modifications of the containers to which the algorithms are applied.

### Figure 23.11. Mutating-sequence algorithms.

---

[Page 1124]

#### Mutating-sequence algorithms

copy	remove	reverse_copy
copy_backward	remove_copy	rotate
fill	remove_copy_if	rotate_copy
fill_n	remove_if	stable_partition
generate	replace	swap
generate_n	replace_copy	swap_ranges
iter_swap	replace_copy_if	transform
partition	replace_if	unique
random_shuffle	reverse	unique_copy

---

[Page 1124]

Figure 23.12 shows many of the nonmodifying sequence algorithms i.e., the algorithms that do not result in modifications of the containers to which they are applied. Figure 23.13 shows the numerical algorithms of the header file [`<numeric>`](#).

### Figure 23.12. Nonmutating sequence algorithms.

<b>Nonmodifying sequence algorithms</b>		
adjacent_find	find	find_if
count	find_each	mismatch
count_if	find_end	search
equal	find_first_of	search_n

### Figure 23.13. Numerical algorithms from header file [`<numeric>`](#).

## Numerical algorithms from header file <numeric>

accumulate	partial_sum
inner_product	adjacent_difference

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1125]

### Performance Tip 23.4



Insertion at the back of a `vector` is efficient. The `vector` simply grows, if necessary, to accommodate the new item. It is expensive to insert (or delete) an element in the middle of a `vector` the entire portion of the `vector` after the insertion (or deletion) point must be moved, because `vector` elements occupy contiguous cells in memory just as C or C++ "raw" arrays do.

[Figure 23.2](#) presented the operations common to all the STL containers. Beyond these operations, each container typically provides a variety of other capabilities. Many of these capabilities are common to several containers, but they are not always equally efficient for each container. The programmer must choose the container most appropriate for the application.

### Performance Tip 23.5



Applications that require frequent insertions and deletions at both ends of a container normally use a `deque` rather than a `vector`. Although we can insert and delete elements at the front and back of both a `vector` and a `deque`, class `deque` is more efficient than `vector` for doing insertions and deletions at the front.

### Performance Tip 23.6



Applications with frequent insertions and deletions in the middle and/or at the extremes of a container normally use a `list`, due to its efficient implementation of insertion and deletion anywhere in the data structure.

In addition to the common operations described in [Fig. 23.2](#), the sequence containers have several other common operations: `front` to return a reference to the first element in the container, `back` to return a reference to the last element in the container, `push_back` to insert a new element at the end of the container and `pop_back` to remove the last element of the container.

#### 23.2.1. `vector` Sequence Container

Class template `vector` provides a data structure with contiguous memory locations. This enables efficient, direct access to any element of a `vector` via the subscript operator `[ ]`, exactly as with a C or C++ "raw"

array. Class template `vector` is most commonly used when the data in the container must be sorted and easily accessible via a subscript. When a `vector`'s memory is exhausted, the `vector` allocates a larger contiguous area of memory, copies the original elements into the new memory and deallocates the old memory.

#### Performance Tip 23.7



Choose the `vector` container for the best random-access performance.

#### Performance Tip 23.8



Objects of class template `vector` provide rapid indexed access with the overloaded subscript operator `[ ]` because they are stored in contiguous memory like a C or C++ raw array.

#### Performance Tip 23.9



It is faster to insert many elements at once than one at a time.

---

[Page 1126]

An important part of every container is the type of iterator it supports. This determines which algorithms can be applied to the container. A `vector` supports random-access iterators i.e., all iterator operations shown in Fig. 23.10 can be applied to a `vector` iterator. All STL algorithms can operate on a `vector`. The iterators for a `vector` are normally implemented as pointers to elements of the `vector`. Each STL algorithm that takes iterator arguments requires those iterators to provide a minimum level of functionality. If an algorithm requires a forward iterator, for example, that algorithm can operate on any container that provides forward iterators, bidirectional iterators or random-access iterators. As long as the container supports the algorithm's minimum iterator functionality, the algorithm can operate on the container.

## Using Vector and Iterators

Figure 23.14 illustrates several functions of the `vector` class template. Many of these functions are available in every first-class container. You must include header file `<vector>` to use class template `vector`.

**Figure 23.14. Standard Library `vector` class template.**

(This item is displayed on pages 1126 - 1127 in the print version)

```

1 // Fig. 23.14: Fig23_14.cpp
2 // Demonstrating Standard Library vector class template.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <vector> // vector class-template definition
8 using std::vector;
9
10 // prototype for function template printVector
11 template < typename T > void printVector(const vector< T > &integers);
12
13 int main()
14 {
15 const int SIZE = 6; // define array size
16 int array[SIZE] = { 1, 2, 3, 4, 5, 6 }; // initialize array
17 vector< int > integers; // create vector of ints
18
19 cout << "The initial size of integers is: " << integers.size()
20 << "\nThe initial capacity of integers is: " << integers.capacity();
21
22 // function push_back is in every sequence collection
23 integers.push_back(2);
24 integers.push_back(3);
25 integers.push_back(4);
26
27 cout << "\nThe size of integers is: " << integers.size()
28 << "\nThe capacity of integers is: " << integers.capacity();
29 cout << "\n\nOutput array using pointer notation: ";
30
31 // display array using pointer notation
32 for (int *ptr = array; ptr != array + SIZE; ptr++)
33 cout << *ptr << ' ';
34
35 cout << "\nOutput vector using iterator notation: ";
36 printVector(integers);
37 cout << "\nReversed contents of vector integers: ";
38
39 // two const reverse iterators
40 vector< int >::const_reverse_iterator reverseIterator;
41 vector< int >::const_reverse_iterator tempIterator = integers.rend();
42
43 // display vector in reverse order using reverse_iterator
44 for (reverseIterator = integers.rbegin();
```

```

45 reverseIterator != tempIterator; ++reverseIterator)
46 cout << *reverseIterator << ' ';
47
48 cout << endl;
49 return 0;
50 } // end main
51
52 // function template for outputting vector elements
53 template < typename T > void printVector(const vector< T > &integers2)
54 {
55 typename vector< T >::const_iterator constIterator; // const_iterator
56
57 // display vector elements using const_iterator
58 for (constIterator = integers2.begin();
59 constIterator != integers2.end(); ++constIterator)
60 cout << *constIterator << ' ';
61 } // end function printVector

```

```

The initial size of integers is: 0
The initial capacity of integers is: 0
The size of integers is: 3
The capacity of integers is: 4

Output array using pointer notation: 1 2 3 4 5 6
Output vector using iterator notation: 2 3 4
Reversed contents of vector integers: 4 3 2

```

[Page 1127]

Line 17 defines an instance called `integers` of class template `vector` that stores `int` values. When this object is instantiated, an empty `vector` is created with size 0 (i.e., the number of elements stored in the `vector`) and capacity 0 (i.e., the number of elements that can be stored without allocating more memory to the `vector`).

Lines 19 and 20 demonstrate the `size` and `capacity` functions; each initially returns 0 for `vector v` in this example. Function `sizeavailable` in every container returns the number of elements currently stored in the container. Function `capacity` returns the number of elements that can be stored in the `vector` before the `vector` needs to dynamically resize itself to accommodate more elements.

Lines 2325 use function `push_back` available in all sequence containers to add an element to the end of the

`vector`. If an element is added to a full `vector`, the `vector` increases its size some STL implementations have the `vector` double its capacity.

[Page 1128]

### Performance Tip 23.10



It can be wasteful to double a `vector`'s size when more space is needed. For example, a full `vector` of 1,000,000 elements resizes to accommodate 2,000,000 elements when a new element is added. This leaves 999,999 unused elements. Programmers can use `resize` to control space usage better.

Lines 27 and 28 use `size` and `capacity` to illustrate the new size and capacity of the `vector` after the three `push_back` operations. Function `size` returns the number of elements added to the `vector`. Function `capacity` returns 4, indicating that we can add one more element before the `vector` needs to add more memory. When we added the first element, the `vector` allocated space for one element, and the size became 1 to indicate that the `vector` contained only one element. When we added the second element, the capacity doubled to 2 and the size became 2 as well. When we added the third element, the capacity doubled again to 4. So we can actually add another element before the `vector` needs to allocation more space. When the `vector` eventually fills its allocated capacity and the program attempts to add one more element to the `vector`, the `vector` will double its capacity to 8 elements.

The manner in which a `vector` grows to accommodate more elements a time consuming operation is not specified by the C++ Standard Document. C++ library implementors use various clever schemes to minimize the overhead of resizing a `vector`. Hence, the output of this program may vary, depending on the version of `vector` that comes with your compiler. Some library implementors allocate a large initial capacity. If a `vector` stores a small number of elements, such capacity may be a waste of space. However, it can greatly improve performance if a program adds many elements to a `vector` and does not have to reallocate memory to accommodate those elements. This is a classic space-time trade-off. Library implementors must balance the amount of memory used against the amount of time required to perform various `vector` operations.

Lines 3233 demonstrate how to output the contents of an array using pointers and pointer arithmetic. Line 36 calls function `printVector` (defined at lines 5361) to output the contents of a `vector` using iterators. Function template `printVector` receives a `const` reference to a `vector< T >` as its argument. Line 55 defines a `const_iterator` called `constIterator` that iterates through the `vector` and outputs its contents. Notice that the declaration in line 55 is prefixed with the keyword `typename`. Because `printVector` is a function template and `vector< T >` will be specialized differently for each function-template specialization, the compiler cannot tell at compile time whether or not `vector< T >:: const_iterator` is a type. In particular specialization, `const_iterator` could be a `static` variable. The compiler needs this information to compile the program correctly. Therefore, you must tell the compiler that a qualified name, whether the qualifier is a dependent type, is expected to be a type in every specialization.

A `const_iterator` enables the program to read the elements of the `vector`, but does not allow the program to modify the elements. The `for` statement at lines 5860 initializes `constIterator` using `vector` member function `begin`, which returns a `const_iterator` to the first element in the `vector`. There is another version of `begin` that returns an `iterator` that can be used for non-`const` containers. Note that a `const_iterator` is returned because the identifier `integers2` was declared `const` in the parameter list of function `printVector`. The loop continues as long as `constIterator` has not reached the end of the `vector`. This is determined by comparing `constIterator` to the result of `integers2.end()`, which returns an iterator indicating the location past the last element of the `vector`. If `constIterator` is equal to this value, the end of the `vector` has been reached. Functions `begin` and `end` are available for all first-class containers. The body of the loop dereferences iterator `constIterator` to get the value in the current element of the `vector`. Remember that the iterator acts like a pointer to the element and that operator `*` is overloaded to return a reference to the element. The expression `+constIterator` (line 59) positions the iterator to the next element of the `vector`.

---

[Page 1129]

#### Performance Tip 23.11



Use prefix increment when applied to STL iterators because the prefix increment operator does not return a value that must be stored in a temporary object.

#### Error-Prevention Tip 23.4



Only random-access iterators support `<`. It is better to use `!=` and `end` to test for the end of a container.

Line 40 declares a `const_reverse_iterator` that can be used to iterate through a `vector` backward. Line 41 declares a `const_reverse_iterator` variable `tempIterator` and initializes it to the iterator returned by function `rend` (i.e., the iterator for the ending point when iterating through the container in reverse). All first-class containers support this type of iterator. Lines 4446 use a `for` statement similar to that in function `printVector` to iterate through the `vector`. In this loop, function `rbegin` (i.e., the iterator for the starting point when iterating through the container in reverse) and `tempIterator` delineate the range of elements to output. As with functions `begin` and `end`, `rbegin` and `rend` can return a `const_reverse_iterator` or a `reverse_iterator`, based on whether or not the container is constant.

#### Performance Tip 23.12



For performance reasons, capture the loop ending value before the loop and compare against that, rather than having a (potentially expensive) function call for each iteration.

## Vector Element-Manipulation Functions

**Figure 23.15** illustrates functions that enable retrieval and manipulation of the elements of a `vector`. Line 17 uses an overloaded `vector` constructor that takes two iterators as arguments to initialize `integers`. Remember that pointers into an array can be used as iterators. Line 17 initializes `integers` with the contents of array `array` up to but not including location `array + SIZE`.

**Figure 23.15. `vector` class template element-manipulation functions.**

(This item is displayed on pages 1129 - 1131 in the print version)

```

1 // Fig. 23.15: Fig23_15.cpp
2 // Testing Standard Library vector class template
3 // element-manipulation functions.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <vector> // vector class-template definition
9 #include <algorithm> // copy algorithm
10 #include <iterator> // ostream_iterator iterator
11 #include <stdexcept> // out_of_range exception
12
13 int main()
14 {
15 const int SIZE = 6;
16 int array[SIZE] = { 1, 2, 3, 4, 5, 6 };
17 std::vector< int > integers(array, array + SIZE);
18 std::ostream_iterator< int > output(cout, " ");
19
20 cout << "Vector integers contains: ";
21 std::copy(integers.begin(), integers.end(), output);
22
23 cout << "\nFirst element of integers: " << integers.front()
24 << "\nLast element of integers: " << integers.back();
25
26 integers[0] = 7; // set first element to 7
27 integers.at(2) = 10; // set element at position 2 to 10
28
29 // insert 22 as 2nd element
30 integers.insert(integers.begin() + 1, 22);

```

```
31
32 cout << "\n\nContents of vector integers after changes: ";
33 std::copy(integers.begin(), integers.end(), output);
34
35 // access out-of-range element
36 try
37 {
38 integers.at(100) = 777;
39 } // end try
40 catch (std::out_of_range outOfRange) // out_of_range exception
41 {
42 cout << "\n\nException: " << outOfRange.what();
43 } // end catch
44
45 // erase first element
46 integers.erase(integers.begin());
47 cout << "\n\nVector integers after erasing first element: ";
48 std::copy(integers.begin(), integers.end(), output);
49
50 // erase remaining elements
51 integers.erase(integers.begin(), integers.end());
52 cout << "\nAfter erasing all elements, vector integers "
53 << (integers.empty() ? "is" : "is not") << " empty";
54
55 // insert elements from array
56 integers.insert(integers.begin(), array, array + SIZE);
57 cout << "\n\nContents of vector integers before clear: ";
58 std::copy(integers.begin(), integers.end(), output);
59
60 // empty integers; clear calls erase to empty a collection
61 integers.clear();
62 cout << "\nAfter clear, vector integers "
63 << (integers.empty() ? "is" : "is not") << " empty" << endl;
64 return 0;
65 } // end main
```

```

Vector integers contains: 1 2 3 4 5 6
First element of integers: 1
Last element of integers: 6

Contents of vector integers after changes: 7 22 2 10 4 5 6

Exception: invalid vector<T> subscript

Vector integers after erasing first element: 22 2 10 4 5 6
After erasing all elements, vector integers is empty

Contents of vector integers before clear: 1 2 3 4 5 6
After clear, vector integers is empty

```

[Page 1131]

Line 18 defines an `ostream_iterator` called `output` that can be used to output integers separated by single spaces via `cout`. An `ostream_iterator< int >` is a type-safe output mechanism that outputs only values of type `int` or a compatible type. The first argument to the constructor specifies the output stream, and the second argument is a string specifying the separator for the values output in this case, the string contains a space character. We use the `ostream_iterator` (defined in header `<iterator>`) to output the contents of the `vector` in this example.

Line 21 uses algorithm `copy` from the Standard Library to output the entire contents of `vector integers` to the standard output. Algorithm `copy` copies each element in the container starting with the location specified by the iterator in its first argument and continuing up to but not including the location specified by the iterator in its second argument. The first and second arguments must satisfy input iterator requirements they must be iterators through which values can be read from a container. Also, applying `++` to the first iterator must eventually cause it to reach the second iterator argument in the container. The elements are copied to the location specified by the output iterator (i.e., an iterator through which a value can be stored or output) specified as the last argument. In this case, the output iterator is an `ostream_iterator` (`output`) that is attached to `cout`, so the elements are copied to the standard output. To use the algorithms of the Standard Library, you must include the header file `<algorithm>`.

Lines 2324 use functions `front` and `back` (available for all sequence containers) to determine the first and last element of the `vector`, respectively. Notice the difference between functions `front` and `begin`. Function `front` returns a reference to the first element in the `vector`, while function `begin` returns a random access iterator pointing to the first element in the `vector`. Also notice the difference between functions `back` and `end`. Function `back` returns a reference to the last element in the `vector`, while function `end` returns a random access iterator pointing to the end of the `vector` (the location after the last

element).

[Page 1132]

### Common Programming Error 23.3



The `vector` must not be empty; otherwise, results of the `front` and `back` functions are undefined.

Lines 2627 illustrate two ways to subscript through a `vector` (which also can be used with the `deque` containers). Line 26 uses the subscript operator that is overloaded to return either a reference to the value at the specified location or a constant reference to that value, depending on whether the container is constant. Function `at` (line 27) performs the same operation, but with bounds checking. Function `at` first checks the value supplied as an argument and determines whether it is in the bounds of the `vector`. If not, function `at` throws an `out_of_bounds` exception defined in header `<stdexcept>` (as demonstrated in lines 3643). [Figure 23.16](#) shows some of the STL exception types. (The Standard Library exception types are discussed in [Chapter 16](#), Exception Handling.)

**Figure 23.16. Some STL exception types.**

STL exception types	Description
<code>out_of_range</code>	Indicates when subscript is out of range.e.g., when an invalid subscript is specified to <code>vector</code> member function <code>at</code> .
<code>invalid_argument</code>	Indicates an invalid argument was passed to a function.
<code>length_error</code>	Indicates an attempt to create too long a container, <code>string</code> , etc.
<code>bad_alloc</code>	Indicates that an attempt to allocate memory with <code>s</code> (or with an allocator) failed because not enough memory was available.

Line 30 uses one of the three overloaded `insert` functions provided by each sequence container. Line 30 inserts the value 22 before the element at the location specified by the iterator in the first argument. In this example, the iterator is pointing to the second element of the `vector`, so 22 is inserted as the second element and the original second element becomes the third element of the `vector`. Other versions of `insert` allow inserting multiple copies of the same value starting at a particular position in the container, or inserting a range of values from another container (or array), starting at a particular position in the original container.

Lines 46 and 51 use the two `erase` functions that are available in all first-class containers. Line 46 indicates that the element at the location specified by the iterator argument should be removed from the container (in this example, the element at the beginning of the `vector`). Line 51 specifies that all elements in the range starting with the location of the first argument up to but not including the location of the second argument should be erased from the container. In this example, all the elements are erased from the `vector`. Line 53 uses function `empty` (available for all containers and adapters) to confirm that the `vector` is empty.

### Common Programming Error 23.4



Erasing an element that contains a pointer to a dynamically allocated object does not delete that object; this can lead to a memory leak.

Line 56 demonstrates the version of function `insert` that uses the second and third arguments to specify the starting location and ending location in a sequence of values (possibly from another container; in this case, from array of integers `array`) that should be inserted into the `vector`. Remember that the ending location specifies the position in the sequence after the last element to be inserted; copying is performed up to but not including this location.

[Page 1133]

Finally, line 61 uses function `clear` (found in all first-class containers) to empty the `vector`. This function calls the version of `erase` used in line 51 to empty the `vector`.

[Note: Other functions that are common to all containers and common to all sequence containers have not yet been covered. We will cover most of these in the next few sections. We will also cover many functions that are specific to each container.]

#### 23.2.2. `list` Sequence Container

The `list` sequence container provides an efficient implementation for insertion and deletion operations at any location in the container. If most of the insertions and deletions occur at the ends of the container, the `deque` data structure (Section 23.2.3) provides a more efficient implementation. Class template `list` is implemented as a doubly linked list; every node in the `list` contains a pointer to the previous node in the `list` and to the next node in the `list`. This enables class template `list` to support bidirectional iterators that allow the container to be traversed both forward and backward. Any algorithm that requires input, output, forward or bidirectional iterators can operate on a `list`. Many of the `list` member functions manipulate the elements of the container as an ordered set of elements.

In addition to the member functions of all STL containers in Fig. 23.2 and the common member functions of

all sequence containers discussed in Section 23.2, class template `list` provides nine other member functions `splice`, `push_front`, `pop_front`, `remove`, `remove_if`, `unique`, `merge`, `reverse` and `sort`. Several of these member functions are list-optimized implementations of STL algorithms presented in Section 23.5. Figure 23.17 demonstrates several features of class `list`. Remember that many of the functions presented in Figs. 23.1423.15 can be used with class `list`. Header file `<list>` must be included to use class `list`.

### Figure 23.17. Standard Library `list` class template.

(This item is displayed on pages 1133 - 1135 in the print version)

```

1 // Fig. 23.17: Fig23_17.cpp
2 // Standard library list class template test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <list> // list class-template definition
8 #include <algorithm> // copy algorithm
9 #include <iostream> // ostream_iterator
10
11 // prototype for function template printList
12 template < typename T > void printList(const std::list< T > &listRef);
13
14 int main()
15 {
16 const int SIZE = 4;
17 int array[SIZE] = { 2, 6, 4, 8 };
18 std::list< int > values; // create list of ints
19 std::list< int > otherValues; // create list of ints
20
21 // insert items in values
22 values.push_front(1);
23 values.push_front(2);
24 values.push_back(4);
25 values.push_back(3);
26
27 cout << "values contains: ";
28 printList(values);
29
30 values.sort(); // sort values
31 cout << "\nvalues after sorting contains: ";
32 printList(values);
33
34 // insert elements of array into otherValues
35 otherValues.insert(otherValues.begin(), array, array + SIZE);
36 cout << "\nAfter insert, otherValues contains: ";

```

```
37 printList(otherValues);
38
39 // remove otherValues elements and insert at end of values
40 values.splice(values.end(), otherValues);
41 cout << "\nAfter splice, values contains: ";
42 printList(values);
43
44 values.sort(); // sort values
45 cout << "\nAfter sort, values contains: ";
46 printList(values);
47
48 // insert elements of array into otherValues
49 otherValues.insert(otherValues.begin(), array, array + SIZE);
50 otherValues.sort();
51 cout << "\nAfter insert, otherValues contains: ";
52 printList(otherValues);
53
54 // remove otherValues elements and insert into values in sorted order
55 values.merge(otherValues);
56 cout << "\nAfter merge:\n values contains: ";
57 printList(values);
58 cout << "\n otherValues contains: ";
59 printList(otherValues);
60
61 values.pop_front(); // remove element from front
62 values.pop_back(); // remove element from back
63 cout << "\nAfter pop_front and pop_back:\n values contains: ";
64 printList(values);
65
66 values.unique(); // remove duplicate elements
67 cout << "\nAfter unique, values contains: ";
68 printList(values);
69
70 // swap elements of values and otherValues
71 values.swap(otherValues);
72 cout << "\nAfter swap:\n values contains: ";
73 printList(values);
74 cout << "\n otherValues contains: ";
75 printList(otherValues);
76
77 // replace contents of values with elements of otherValues
78 values.assign(otherValues.begin(), otherValues.end());
79 cout << "\nAfter assign, values contains: ";
80 printList(values);
81
82 // remove otherValues elements and insert into values in sorted order
83 values.merge(otherValues);
84 cout << "\nAfter merge, values contains: ";
85 printList(values);
```

```

86
87 values.remove(4); // remove all 4s
88 cout << "\nAfter remove(4), values contains: ";
89 printList(values);
90 cout << endl;
91 return 0;
92 } // end main
93
94 // printList function template definition; uses
95 // ostream_iterator and copy algorithm to output list elements
96 template < typename T > void printList(const std::list< T > &listRef)
97 {
98 if (listRef.empty()) // list is empty
99 cout << "List is empty";
100 else
101 {
102 std::ostream_iterator< T > output(cout, " ");
103 std::copy(listRef.begin(), listRef.end(), output);
104 } // end else
105 } // end function printList

```

```

values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
After insert, otherValues contains: 2 6 4 8
After splice, values contains: 1 2 3 4 2 6 4 8
After sort, values contains: 1 2 2 3 4 4 6 8
After insert, otherValues contains: 2 4 6 8
After merge:
 values contains: 1 2 2 2 3 4 4 4 6 6 8 8
 otherValues contains: List is empty
After pop_front and pop_back:
 values contains: 2 2 2 3 4 4 4 6 6 8
After unique, values contains: 2 3 4 6 8
After swap:
 values contains: List is empty
 otherValues contains: 2 3 4 6 8
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8 8
After remove(4), values contains: 2 2 3 3 6 6 8 8

```

Lines 1819 instantiate two `list` objects capable of storing integers. Lines 2223 use function `push_front` to insert integers at the beginning of `values`. Function `push_front` is specific to classes `list` and `deque` (not to `vector`). Lines 2425 use function `push_back` to insert integers at the end of `values`. Remember that function `push_back` is common to all sequence containers.

Line 30 uses `list` member function `sort` to arrange the elements in the `list` in ascending order. [Note: This is different from the `sort` in the STL algorithms.] A second version of function `sort` that allows the programmer to supply a binary predicate function that takes two arguments (values in the list), performs a comparison and returns a `bool` value indicating the result. This function determines the order in which the elements of the `list` are sorted. This version could be particularly useful for a `list` that stores pointers rather than values. [Note: We demonstrate a unary predicate function in Fig. 23.28. A unary predicate function takes a single argument, performs a comparison using that argument and returns a `bool` value indicating the result.]

Line 40 uses `list` function `splice` to remove the elements in `otherValues` and insert them into `values` before the iterator position specified as the first argument. There are two other versions of this function. Function `splice` with three arguments allows one element to be removed from the container specified as the second argument from the location specified by the iterator in the third argument. Function `splice` with four arguments uses the last two arguments to specify a range of locations that should be removed from the container in the second argument and placed at the location specified in the first argument.

After inserting more elements in `otherValues` and sorting both `values` and `otherValues`, line 55 uses `list` member function `merge` to remove all elements of `otherValues` and insert them in sorted order into `values`. Both `lists` must be sorted in the same order before this operation is performed. A second version of `merge` enables the programmer to supply a predicate function that takes two arguments (values in the list) and returns a `bool` value. The predicate function specifies the sorting order used by `merge`.

Line 61 uses `list` function `pop_front` to remove the first element in the `list`. Line 62 uses function `pop_back` (available for all sequence containers) to remove the last element in the `list`.

Line 66 uses `list` function `unique` to remove duplicate elements in the `list`. The `list` should be in sorted order (so that all duplicates are side by side) before this operation is performed, to guarantee that all duplicates are eliminated. A second version of `unique` enables the programmer to supply a predicate function that takes two arguments (values in the list) and returns a `bool` value specifying whether two elements are equal.

Line 71 uses function `swap` (available to all containers) to exchange the contents of `values` with the contents of `otherValues`.

Line 78 uses `list` function `assign` to replace the contents of `values` with the contents of `otherValues` in the range specified by the two iterator arguments. A second version of `assign` replaces the original contents with copies of the value specified in the second argument. The first argument of the function specifies the number of copies. Line 87 uses `list` function `remove` to delete all copies of the value 4 from the `list`.

### 23.2.3. deque Sequence Container

Class `deque` provides many of the benefits of a `vector` and a `list` in one container. The term `deque` is short for "double-ended queue." Class `deque` is implemented to provide efficient indexed access (using subscripting) for reading and modifying its elements, much like a `vector`. Class `deque` is also implemented for efficient insertion and deletion operations at its front and back, much like a `list` (although a `list` is also capable of efficient insertions and deletions in the middle of the `list`). Class `deque` provides support for random-access iterators, so `deques` can be used with all STL algorithms. One of the most common uses of a `deque` is to maintain a first-in, first-out queue of elements. In fact, a `deque` is the default underlying implementation for the `queue` adaptor (Section 23.4.2).

[Page 1137]

Additional storage for a `deque` can be allocated at either end of the `deque` in blocks of memory that are typically maintained as an array of pointers to those blocks.<sup>[3]</sup> Due to the noncontiguous memory layout of a `deque`, a `deque` iterator must be more intelligent than the pointers that are used to iterate through `vectors` or pointer-based arrays.

<sup>[3]</sup> This is an implementation-specific detail, not a requirement of the C++ standard.

#### Performance Tip 23.13



In general, `deque` has slightly higher overhead than `vector`.

#### Performance Tip 23.14



Insertions and deletions in the middle of a `deque` are optimized to minimize the number of elements copied, so it is more efficient than a `vector` but less efficient than a `list` for this kind of modification.

Class `deque` provides the same basic operations as class `vector`, but adds member functions `push_front` and `pop_front` to allow insertion and deletion at the beginning of the `deque`, respectively.

Figure 23.18 demonstrates features of class `deque`. Remember that many of the functions presented in Fig. 23.14, Fig. 23.15 and Fig. 23.17 also can be used with class `deque`. Header file `<deque>` must be included to use class `deque`.

**Figure 23.18. Standard Library deque class template.**

(This item is displayed on pages 1137 - 1138 in the print version)

```
1 // Fig. 23.18: Fig23_18.cpp
2 // Standard Library class deque test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <deque> // deque class-template definition
8 #include <algorithm> // copy algorithm
9 #include <iterator> // ostream_iterator
10
11 int main()
12 {
13 std::deque< double > values; // create deque of doubles
14 std::ostream_iterator< double > output(cout, " ");
15
16 // insert elements in values
17 values.push_front(2.2);
18 values.push_front(3.5);
19 values.push_back(1.1);
20
21 cout << "values contains: ";
22
23 // use subscript operator to obtain elements of values
24 for (unsigned int i = 0; i < values.size(); i++)
25 cout << values[i] << ' ';
26
27 values.pop_front(); // remove first element
28 cout << "\nAfter pop_front, values contains: ";
29 std::copy(values.begin(), values.end(), output);
30
31 // use subscript operator to modify element at location 1
32 values[1] = 5.4;
33 cout << "\nAfter values[1] = 5.4, values contains: ";
34 std::copy(values.begin(), values.end(), output);
35 cout << endl;
36 return 0;
37 } // end main
```

```
values contains: 3.5 2.2 1.1
After pop_front, values contains: 2.2 1.1
After values[1] = 5.4, values contains: 2.2 5.4
```

Line 13 instantiates a deque that can store double values. Lines 1719 use functions `push_front` and `push_back` to insert elements at the beginning and end of the deque. Remember that `push_back` is available for all sequence containers, but `push_front` is available only for class `list` and class `deque`.

---

[Page 1138]

The `for` statement at lines 2425 uses the subscript operator to retrieve the value in each element of the deque for output. Note that the condition uses function `size` to ensure that we do not attempt to access an element outside the bounds of the deque.

Line 27 uses function `pop_front` to demonstrate removing the first element of the deque. Remember that `pop_front` is available only for class `list` and class `deque` (not for class `vector`).

Line 32 uses the subscript operator to create an lvalue. This enables values to be assigned directly to any element of the deque.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1139]

### 23.3.1. `multiset` Associative Container

The `multiset` associative container provides fast storage and retrieval of keys and allows duplicate keys. The ordering of the elements is determined by a **comparator function object**. For example, in an integer `multiset`, elements can be sorted in ascending order by ordering the keys with `comparator function object less< int >`. We discuss function objects in detail in [Section 23.7](#). The data type of the keys in all associative containers must support comparison properly based on the comparator function object specified. Keys sorted with `less< T >` must support comparison with `operator<`. If the keys used in the associative containers are of user-defined data types, those types must supply the appropriate comparison operators. A `multiset` supports bidirectional iterators (but not random-access iterators).

[Figure 23.19](#) demonstrates the `multiset` associative container for a `multiset` of integers sorted in ascending order. Header file `<set>` must be included to use class `multiset`. Containers `multiset` and `set` provide the same member functions.

#### Figure 23.19. Standard Library `multiset` class template.

(This item is displayed on pages 1139 - 1141 in the print version)

```

1 // Fig. 23.19: Fig23_19.cpp
2 // Testing Standard Library class multiset
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <set> // multiset class-template definition
8
9 // define short name for multiset type used in this program
10 typedef std::multiset< int, std::less< int > > Ims;
11
12 #include <algorithm> // copy algorithm
13 #include <iterator> // ostream_iterator
14
15 int main()
16 {
17 const int SIZE = 10;
18 int a[SIZE] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
19 Ims intMultiset; // Ims is typedef for "integer multiset"
20 std::ostream_iterator< int > output(cout, " ");

```

```

21
22 cout << "There are currently " << intMultiset.count(15)
23 << " values of 15 in the multiset\n";
24
25 intMultiset.insert(15); // insert 15 in intMultiset
26 intMultiset.insert(15); // insert 15 in intMultiset
27 cout << "After inserts, there are " << intMultiset.count(15)
28 << " values of 15 in the multiset\n\n";
29
30 // iterator that cannot be used to change element values
31 Ims::const_iterator result;
32
33 // find 15 in intMultiset; find returns iterator
34 result = intMultiset.find(15);
35
36 if (result != intMultiset.end()) // if iterator not at end
37 cout << "Found value 15\n"; // found search value 15
38
39 // find 20 in intMultiset; find returns iterator
40 result = intMultiset.find(20);
41
42 if (result == intMultiset.end()) // will be true hence
43 cout << "Did not find value 20\n"; // did not find 20
44
45 // insert elements of array a into intMultiset
46 intMultiset.insert(a, a + SIZE);
47 cout << "\nAfter insert, intMultiset contains:\n";
48 std::copy(intMultiset.begin(), intMultiset.end(), output);
49
50 // determine lower and upper bound of 22 in intMultiset
51 cout << "\n\nLower bound of 22: "
52 << *(intMultiset.lower_bound(22));
53 cout << "\nUpper bound of 22: " << *(intMultiset.upper_bound(22));
54
55 // p represents pair of const_iterators
56 std::pair< Ims::const_iterator, Ims::const_iterator > p;
57
58 // use equal_range to determine lower and upper bound
59 // of 22 in intMultiset
60 p = intMultiset.equal_range(22);
61
62 cout << "\n\nequal_range of 22:" << "\n Lower bound: "
63 << *(p.first) << "\n Upper bound: " << *(p.second);
64 cout << endl;
65 return 0;
66 } // end main

```

```

There are currently 0 values of 15 in the multiset
After inserts, there are 2 values of 15 in the multiset

Found value 15
Did not find value 20

After insert, intMultiset contains:
1 7 9 13 15 15 18 22 22 30 85 100

Lower bound of 22: 22
Upper bound of 22: 30

equal_range of 22:
 Lower bound: 22
 Upper bound: 30

```

Line 10 uses a `typedef` to create a new type name (alias) for a `multiset` of integers ordered in ascending order, using the function object `less< int >`. Ascending order is the default for a `multiset`, so `std::less< int >` can be omitted in line 10. This new type (`Ims`) is then used to instantiate an integer `multiset` object, `intMultiset` (line 19).

## Good Programming Practice 23.1



Use `typedefs` to make code with long type names (such as `multisets`) easier to read.

The output statement at line 22 uses function `count` (available to all associative containers) to count the number of occurrences of the value 15 currently in the `multiset`.

---

[Page 1141]

Lines 2526 use one of the three versions of function `insert` to add the value 15 to the `multiset` twice. A second version of `insert` takes an iterator and a value as arguments and begins the search for the insertion point from the iterator position specified. A third version of `insert` takes two iterators as arguments that specify a range of values to add to the `multiset` from another container.

Line 34 uses function `find` (available to all associative containers) to locate the value 15 in the `multiset`. Function `find` returns an `iterator` or a `const_iterator` pointing to the earliest location at which the value is found. If the value is not found, `find` returns an `iterator` or a `const_iterator` equal to the value returned by a call to `end`. Line 41 demonstrates this case.

Line 46 uses function `insert` to insert the elements of array `a` into the `multiset`. At line 48, the `copy` algorithm copies the elements of the `multiset` to the standard output. Note that the elements are displayed in ascending order.

Lines 52 and 53 use functions `lower_bound` and `upper_bound` (available in all associative containers) to locate the earliest occurrence of the value 22 in the `multiset` and the element after the last occurrence of the value 22 in the `multiset`. Both functions return `iterators` or `const_iterators` pointing to the appropriate location or the iterator returned by `end` if the value is not in the `multiset`.

Line 56 instantiates an instance of class `pair` called `p`. Objects of class `pair` are used to associate pairs of values. In this example, the contents of a `pair` are two `const_iterators` for our integer-based `multiset`. The purpose of `p` is to store the return value of `multiset` function `equal_range` that returns a `pair` containing the results of both a `lower_bound` and an `upper_bound` operation. Type `pair` contains two public data members called `first` and `second`.

Line 60 uses function `equal_range` to determine the `lower_bound` and `upper_bound` of 22 in the `multiset`. Line 63 uses `p.first` and `p.second`, respectively, to access the `lower_bound` and `upper_bound`. We dereferenced the iterators to output the values at the locations returned from `equal_range`.

[Page 1142]

### 23.3.2. set Associative Container

The `set` associative container is used for fast storage and retrieval of unique keys. The implementation of a `set` is identical to that of a `multiset`, except that a `set` must have unique keys. Therefore, if an attempt is made to insert a duplicate key into a `set`, the duplicate is ignored; because this is the intended mathematical behavior of a `set`, we do not identify it as a common programming error. A `set` supports bidirectional iterators (but not random-access iterators). [Figure 23.20](#) demonstrates a `set` of doubles. Header file `<set>` must be included to use class `set`.

#### Figure 23.20. Standard Library `set` class template.

(This item is displayed on pages 1142 - 1143 in the print version)

```

1 // Fig. 23.20: Fig23_20.cpp
2 // Standard Library class set test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <set>
8
9 // define short name for set type used in this program
10 typedef std::set< double, std::less< double > > DoubleSet;
11
12 #include <algorithm>
13 #include <iterator> // ostream_iterator
14
15 int main()
16 {
17 const int SIZE = 5;
18 double a[SIZE] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
19 DoubleSet doubleSet(a, a + SIZE);
20 std::ostream_iterator< double > output(cout, " ");
21
22 cout << "doubleSet contains: ";
23 std::copy(doubleSet.begin(), doubleSet.end(), output);
24
25 // p represents pair containing const_iterator and bool
26 std::pair< DoubleSet::const_iterator, bool > p;
27
28 // insert 13.8 in doubleSet; insert returns pair in which
29 // p.first represents location of 13.8 in doubleSet and
30 // p.second represents whether 13.8 was inserted
31 p = doubleSet.insert(13.8); // value not in set
32 cout << "\n\n" << *(p.first)
33 << (p.second ? " was" : " was not") << " inserted";
34 cout << "\ndoubleSet contains: ";
35 std::copy(doubleSet.begin(), doubleSet.end(), output);
36
37 // insert 9.5 in doubleSet
38 p = doubleSet.insert(9.5); // value already in set
39 cout << "\n\n" << *(p.first)
40 << (p.second ? " was" : " was not") << " inserted";
41 cout << "\ndoubleSet contains: ";
42 std::copy(doubleSet.begin(), doubleSet.end(), output);
43 cout << endl;
44 return 0;
45 } // end main

```

```

doubleSet contains: 2.1 3.7 4.2 9.5
13.8 was inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8
9.5 was not inserted
doubleSet contains: 2.1 3.7 4.2 9.5 13.8

```

[Page 1143]

Line 10 uses `typedef` to create a new type name (`DoubleSet`) for a set of `double` values ordered in ascending order, using the function object `less< double >`.

Line 19 uses the new type `DoubleSet` to instantiate object `doubleSet`. The constructor call takes the elements in array `a` between `a` and `a + SIZE` (i.e., the entire array) and inserts them into the set. Line 23 uses algorithm `copy` to output the contents of the set. Notice that the value `2.1` which appeared twice in array `a` appears only once in `doubleSet`. This is because container `set` does not allow duplicates.

Line 26 defines a `pair` consisting of a `const_iterator` for a `DoubleSet` and a `bool` value. This object stores the result of a call to `set` function `insert`.

Line 31 uses function `insert` to place the value `13.8` in the set. The returned `pair`, `p`, contains an iterator `p.first` pointing to the value `13.8` in the set and a `bool` value that is `true` if the value was inserted and `false` if the value was not inserted (because it was already in the set). In this case, `13.8` was not in the set, so it was inserted. Line 38 attempts to insert `9.5`, which is already in the set. The output of lines 3940 shows that `9.5` was not inserted.

### 23.3.3. `multimap` Associative Container

The `multimap` associative container is used for fast storage and retrieval of keys and associated values (often called key/value pairs). Many of the functions used with `multisets` and `sets` are also used with `multimaps` and `maps`. The elements of `multimaps` and `maps` are `pairs` of keys and values instead of individual values. When inserting into a `multimap` or `map`, a `pair` object that contains the key and the value is used. The ordering of the keys is determined by a comparator function object. For example, in a `multimap` that uses integers as the key type, keys can be sorted in ascending order by ordering them with comparator function object `less< int >`. Duplicate keys are allowed in a `multimap`, so multiple values can be associated with a single key. This is often called a one-to-many relationship. For example, in a credit-card transaction-processing system, one credit-card account can have many associated transactions; in a university, one student can take many courses, and one professor can teach many

students; in the military, one rank (like "private") has many people. A multimap supports bidirectional iterators, but not random-access iterators. Figure 23.21 demonstrates the multimap associative container. Header file `<map>` must be included to use class `multimap`.

**Figure 23.21. Standard Library `multimap` class template.**

(This item is displayed on pages 1144 - 1145 in the print version)

```

1 // Fig. 23.21: Fig23_21.cpp
2 // Standard Library class multimap test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <map> // map class-template definition
8
9 // define short name for multimap type used in this program
10 typedef std::multimap< int, double, std::less< int > > Mmid;
11
12 int main()
13 {
14 Mmid pairs; // declare the multimap pairs
15
16 cout << "There are currently " << pairs.count(15)
17 << " pairs with key 15 in the multimap\n";
18
19 // insert two value_type objects in pairs
20 pairs.insert(Mmid::value_type(15, 2.7));
21 pairs.insert(Mmid::value_type(15, 99.3));
22
23 cout << "After inserts, there are " << pairs.count(15)
24 << " pairs with key 15\n\n";
25
26 // insert five value_type objects in pairs
27 pairs.insert(Mmid::value_type(30, 111.11));
28 pairs.insert(Mmid::value_type(10, 22.22));
29 pairs.insert(Mmid::value_type(25, 33.333));
30 pairs.insert(Mmid::value_type(20, 9.345));
31 pairs.insert(Mmid::value_type(5, 77.54));
32
33 cout << "Multimap pairs contains:\nKey\tValue\n";
34
35 // use const_iterator to walk through elements of pairs
36 for (Mmid::const_iterator iter = pairs.begin();
37 iter != pairs.end(); ++iter)
38 cout << iter->first << '\t' << iter->second << '\n';
39

```

```

40 cout << endl;
41 return 0;
42 } // end main

```

There are currently 0 pairs with key 15 in the multimap  
 After inserts, there are 2 pairs with key 15

Multimap pairs contains:

Key	Value
5	77.54
10	22.22
15	2.7
15	99.3
20	9.345
25	33.333
30	111.11

---

[Page 1144]

### Performance Tip 23.15



A multimap is implemented to efficiently locate all values paired with a given key.

Line 10 uses `typedef` to define alias `Mmid` for a `multimap` type in which the key type is `int`, the type of a key's associated value is `double` and the elements are ordered in ascending order. Line 14 uses the new type to instantiate a `multimap` called `pairs`. Line 16 uses function `count` to determine the number of key/value pairs with a key of 15.

---

[Page 1145]

Line 20 uses function `insert` to add a new key/value pair to the `multimap`. The expression `Mmid::value_type ( 15, 2.7 )` creates a `pair` object in which `first` is the key (15) of type `int` and `second` is the value (2.7) of type `double`. The type `Mmid::value_type` is defined as part of the `typedef` for the `multimap`. Line 21 inserts another `pair` object with the key 15 and the value 99.3.

Then lines 2324 output the number of pairs with key 15.

Lines 2731 insert five additional pairs into the multimap. The for statement at lines 3638 outputs the contents of the multimap, including both keys and values. Line 38 uses the const\_iterator called iter to access the members of the pair in each element of the multimap. Notice in the output that the keys appear in ascending order.

### 23.3.4. map Associative Container

The map associative container is used for fast storage and retrieval of unique keys and associated values. Duplicate keys are not allowed in a map, so only a single value can be associated with each key. This is called a **one-to-one mapping**. For example, a company that uses unique employee numbers, such as 100, 200 and 300, might have a map that associates employee numbers with their telephone extensions 4321, 4115 and 5217, respectively. With a map you specify the key and get back the associated data quickly. A map is commonly called an **associative array**. Providing the key in a map's subscript operator [ ] locates the value associated with that key in the map. Insertions and deletions can be made anywhere in a map.

Figure 23.22 demonstrates the map associative container and uses the same features as Fig. 23.21 to demonstrate the subscript operator. Header file <map> must be included to use class map. Lines 33 and 34 use the subscript operator of class map. When the subscript is a key that is already in the map (line 33), the operator returns a reference to the associated value. When the subscript is a key that is not in the map (line 34), the operator inserts the key in the map and returns a reference that can be used to associate a value with that key. Line 33 replaces the value for the key 25 (previously 33.333 as specified in line 21) with a new value, 9999.99. Line 34 inserts a new key/value pair in the map (called **creating an association**).

---

[Page 1147]

#### Figure 23.22. Standard Library map class template.

(This item is displayed on pages 1145 - 1146 in the print version)

```

1 // Fig. 23.22: Fig23_22.cpp
2 // Standard Library class map test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <map> // map class-template definition
8
9 // define short name for map type used in this program
10 typedef std::map< int, double, std::less< int > > Mid;
11
12 int main()
13 {
14 Mid pairs;
15
16 // insert eight value_type objects in pairs
17 pairs.insert(Mid::value_type(15, 2.7));
18 pairs.insert(Mid::value_type(30, 111.11));
19 pairs.insert(Mid::value_type(5, 1010.1));
20 pairs.insert(Mid::value_type(10, 22.22));
21 pairs.insert(Mid::value_type(25, 33.333));
22 pairs.insert(Mid::value_type(5, 77.54)); // dup ignored
23 pairs.insert(Mid::value_type(20, 9.345));
24 pairs.insert(Mid::value_type(15, 99.3)); // dup ignored
25
26 cout << "pairs contains:\nKey\tValue\n";
27
28 // use const_iterator to walk through elements of pairs
29 for (Mid::const_iterator iter = pairs.begin();
30 iter != pairs.end(); ++iter)
31 cout << iter->first << '\t' << iter->second << '\n';
32
33 pairs[25] = 9999.99; // use subscripting to change value for key 25
34 pairs[40] = 8765.43; // use subscripting to insert value for key 40
35
36 cout << "\nAfter subscript operations, pairs contains:\nKey\tValue\n";
37
38 // use const_iterator to walk through elements of pairs
39 for (Mid::const_iterator iter2 = pairs.begin();
40 iter2 != pairs.end(); ++iter2)
41 cout << iter2->first << '\t' << iter2->second << '\n';
42
43 cout << endl;
44 return 0;
45 } // end main

```

pairs contains:

Key	Value
5	1010.1
10	22.22
15	2.7
20	9.345
25	33.333
30	111.11

After subscript operations, pairs contains:

Key	Value
5	1010.1
10	22.22
15	2.7
20	9.345
25	9999.99
30	111.11
40	8765.43

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

---

[Page 1148]

Function `pushElements` (lines 4956) pushes the elements onto each `stack`. Line 53 uses function `push` (available in each adapter class) to place an integer on top of the `stack`. Line 54 uses `stack` function `top` to retrieve the top element of the `stack` for output. Function `top` does not remove the top element.

Function `popElements` (lines 5966) pops the elements off each `stack`. Line 63 uses `stack` function `top` to retrieve the top element of the `stack` for output. Line 64 uses function `pop` (available in each adapter class) to remove the top element of the `stack`. Function `pop` does not return a value.

---

[Page 1149]

### 23.4.2. `queue` Adapter

Class `queue` enables insertions at the back of the underlying data structure and deletions from the front (commonly referred to as a first-in, first-out data structure). A `queue` can be implemented with STL data structure `list` or `deque`. By default, a `queue` is implemented with a `deque`. The common `queue` operations are `push` to insert an element at the back of the `queue` (implemented by calling function `push_back` of the underlying container), `pop` to remove the element at the front of the `queue` (implemented by calling function `pop_front` of the underlying container), `front` to get a reference to the first element in the `queue` (implemented by calling function `front` of the underlying container), `back` to get a reference to the last element in the `queue` (implemented by calling function `back` of the underlying container), `empty` to determine whether the `queue` is empty (implemented by calling function `empty` of the underlying container) and `size` to get the number of elements in the `queue` (implemented by calling function `size` of the underlying container).

---

[Page 1150]

#### Performance Tip 23.18



Each of the common operations of a `queue` is implemented as an `inline` function that calls the appropriate function of the underlying container. This avoids the overhead of a second function call.

#### Performance Tip 23.19



For the best performance, use class `deque` or `list` as the underlying container for a queue.

Figure 23.24 demonstrates the `queue` adapter class. Header file `<queue>` must be included to use a queue.

**Figure 23.24. Standard Library `queue` adapter class templates.**

```

1 // Fig. 23.24: Fig23_24.cpp
2 // Standard Library adapter queue test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <queue> // queue adapter definition
8
9 int main()
10 {
11 std::queue< double > values; // queue with doubles
12
13 // push elements onto queue values
14 values.push(3.2);
15 values.push(9.8);
16 values.push(5.4);
17
18 cout << "Popping from values: ";
19
20 // pop elements from queue
21 while (!values.empty())
22 {
23 cout << values.front() << ' ' ; // view front element
24 values.pop(); // remove element
25 } // end while
26
27 cout << endl;
28 return 0;
29 } // end main

```

Popping from values: 3.2 9.8 5.4

Line 11 instantiates a queue that stores double values. Lines 1416 use function `push` to add elements to the queue. The `while` statement at lines 2125 uses function `empty` (available in all containers) to determine whether the queue is empty (line 21). While there are more elements in the queue, line 23 uses `queue` function `front` to read (but not remove) the first element in the queue for output. Line 24 removes the first element in the queue with function `pop` (available in all adapter classes).

---

[Page 1151]

### 23.4.3. priority\_queue Adapter

Class `priority_queue` provides functionality that enables insertions in sorted order into the underlying data structure and deletions from the front of the underlying data structure. A `priority_queue` can be implemented with STL sequence containers `vector` or `deque`. By default, a `priority_queue` is implemented with a `vector` as the underlying container. When elements are added to a `priority_queue`, they are inserted in priority order, such that the highest-priority element (i.e., the largest value) will be the first element removed from the `priority_queue`. This is usually accomplished via a sorting technique called `heapsort` that always maintains the largest value (i.e., highest-priority element) at the front of the data structure such a data structure is called a `heap`. The comparison of elements is performed with comparator function object `less< T >` by default, but the programmer can supply a different comparator.

The common `priority_queue` operations are `push` to insert an element at the appropriate location based on priority order of the `priority_queue` (implemented by calling function `push_back` of the underlying container, then reordering the elements using `heapsort`), `pop` to remove the highest-priority element of the `priority_queue` (implemented by calling function `pop_back` of the underlying container after removing the top element of the heap), `top` to get a reference to the top element of the `priority_queue` (implemented by calling function `front` of the underlying container), `empty` to determine whether the `priority_queue` is empty (implemented by calling function `empty` of the underlying container) and `size` to get the number of elements in the `priority_queue` (implemented by calling function `size` of the underlying container).

#### Performance Tip 23.20



Each of the common operations of a `priority_queue` is implemented as an inline function that calls the appropriate function of the underlying container. This avoids the overhead of a second function call.

#### Performance Tip 23.21



For the best performance, use class `vector` or `deque` as the underlying container for a `priority_queue`.

Figure 23.25 demonstrates the `priority_queue` adapter class. Header file `<queue>` must be included to use class `priority_queue`.

**Figure 23.25. Standard Library `priority_queue` adapter class.**

(This item is displayed on pages 1151 - 1152 in the print version)

```

1 // Fig. 23.25: Fig23_25.cpp
2 // Standard Library adapter priority_queue test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <queue> // priority_queue adapter definition
8
9 int main()
10 {
11 std::priority_queue< double > priorities; // create priority_queue
12
13 // push elements onto priorities
14 priorities.push(3.2);
15 priorities.push(9.8);
16 priorities.push(5.4);
17
18 cout << "Popping from priorities: ";
19
20 // pop element from priority_queue
21 while (!priorities.empty())
22 {
23 cout << priorities.top() << ' '; // view top element
24 priorities.pop(); // remove top element
25 } // end while
26
27 cout << endl;
28 return 0;
29 } // end main

```

Popping from priorities: 9.8 5.4 3.2

[Page 1152]

Line 11 instantiates a `priority_queue` that stores `double` values and uses a `vector` as the underlying data structure. Lines 1416 use function `push` to add elements to the `priority_queue`. The `while` statement at lines 2125 uses function `empty` (available in all containers) to determine whether the `priority_queue` is empty (line 21). While there are more elements, line 23 uses `priority_queue` function `top` to retrieve the highest-priority element in the `priority_queue` for output. Line 24 removes the highest-priority element in the `priority_queue` with function `pop` (available in all adapter classes).

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1153]

## Software Engineering Observation 23.9



Algorithms can be added easily to the STL without modifying the container classes.

### 23.5.1. `fill`, `fill_n`, `generate` and `generate_n`

Figure 23.26 demonstrates algorithms `fill`, `fill_n`, `generate` and `generate_n`. Functions `fill` and `fill_n` set every element in a range of container elements to a specific value. Functions `generate` and `generate_n` use a `generator function` to create values for every element in a range of container elements. The generator function takes no arguments and returns a value that can be placed in an element of the container.

[Page 1154]

#### Figure 23.26. Algorithms `fill`, `fill_n`, `generate` and `generate_n`.

(This item is displayed on pages 1153 - 1154 in the print version)

```
1 // Fig. 23.26: Fig23_26.cpp
2 // Standard Library algorithms fill, fill_n, generate and generate_n.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // algorithm definitions
8 #include <vector> // vector class-template definition
9 #include <iterator> // ostream_iterator
10
11 char nextLetter(); // prototype of generator function
12
13 int main()
14 {
15 std::vector< char > chars(10);
16 std::ostream_iterator< char > output(cout, " ");
```

```

17 std::fill(chars.begin(), chars.end(), '5'); // fill chars with 5s
18
19 cout << "Vector chars after filling with 5s:\n";
20 std::copy(chars.begin(), chars.end(), output);
21
22 // fill first five elements of chars with As
23 std::fill_n(chars.begin(), 5, 'A');
24
25 cout << "\n\nVector chars after filling five elements with As:\n";
26 std::copy(chars.begin(), chars.end(), output);
27
28 // generate values for all elements of chars with nextLetter
29 std::generate(chars.begin(), chars.end(), nextLetter);
30
31 cout << "\n\nVector chars after generating letters A-J:\n";
32 std::copy(chars.begin(), chars.end(), output);
33
34 // generate values for first five elements of chars with nextLetter
35 std::generate_n(chars.begin(), 5, nextLetter);
36
37 cout << "\n\nVector chars after generating K-O for the"
38 << " first five elements:\n";
39 std::copy(chars.begin(), chars.end(), output);
40 cout << endl;
41 return 0;
42 } // end main
43
44 // generator function returns next letter (starts with A)
45 char nextLetter()
46 {
47 static char letter = 'A';
48 return letter++;
49 } // end function nextLetter

```

Vector chars after filling with 5s:  
 5 5 5 5 5 5 5 5 5 5

Vector chars after filling five elements with As:  
 A A A A A 5 5 5 5 5

Vector chars after generating letters A-J:  
 A B C D E F G H I J

Vector chars after generating K-O for the first five elements:  
 K L M N O F G H I J

Line 15 defines a 10-element vector that stores char values. Line 17 uses function `fill` to place the character '5' in every element of vector `chars` from `chars.begin()` up to, but not including, `chars.end()`. Note that the iterators supplied as the first and second argument must be at least forward iterators (i.e., they can be used for both input from a container and output to a container in the forward direction).

Line 23 uses function `fill_n` to place the character 'A' in the first five elements of vector `chars`. The iterator supplied as the first argument must be at least an output iterator (i.e., it can be used for output to a container in the forward direction). The second argument specifies the number of elements to fill. The third argument specifies the value to place in each element.

Line 29 uses function `generate` to place the result of a call to generator function `nextLetter` in every element of vector `chars` from `chars.begin()` up to, but not including, `chars.end()`. The iterators supplied as the first and second arguments must be at least forward iterators. Function `nextLetter` (defined at lines 4549) begins with the character 'A' maintained in a static local variable. The statement at line 48 postincrements the value of `letter` and returns the old value of `letter` each time `nextLetter` is called.

Line 35 uses function `generate_n` to place the result of a call to generator function `nextLetter` in five elements of vector `chars`, starting from `chars.begin()`. The iterator supplied as the first argument must be at least an output iterator.

### 23.5.2. `equal`, `mismatch` and `lexicographical_compare`

Figure 23.27 demonstrates comparing sequences of values for equality using algorithms `equal`, `mismatch` and `lexicographical_compare`.

**Figure 23.27. Algorithms `equal`, `mismatch` and `lexicographical_compare`.**

(This item is displayed on pages 1155 - 1156 in the print version)

```

1 // Fig. 23.27: Fig23_27.cpp
2 // Standard Library functions equal, mismatch and lexicographical_compare.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // algorithm definitions
8 #include <vector> // vector class-template definition
9 #include <iterator> // ostream_iterator
10
11 int main()
12 {
13 const int SIZE = 10;
14 int a1[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
15 int a2[SIZE] = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };
16 std::vector< int > v1(a1, a1 + SIZE); // copy of a1
17 std::vector< int > v2(a1, a1 + SIZE); // copy of a1
18 std::vector< int > v3(a2, a2 + SIZE); // copy of a2
19 std::ostream_iterator< int > output(cout, " ");
20
21 cout << "Vector v1 contains: ";
22 std::copy(v1.begin(), v1.end(), output);
23 cout << "\nVector v2 contains: ";
24 std::copy(v2.begin(), v2.end(), output);
25 cout << "\nVector v3 contains: ";
26 std::copy(v3.begin(), v3.end(), output);
27
28 // compare vectors v1 and v2 for equality
29 bool result = std::equal(v1.begin(), v1.end(), v2.begin());
30 cout << "\n\nVector v1 " << (result ? "is" : "is not")
31 << " equal to vector v2.\n";
32
33 // compare vectors v1 and v3 for equality
34 result = std::equal(v1.begin(), v1.end(), v3.begin());
35 cout << "Vector v1 " << (result ? "is" : "is not")
36 << " equal to vector v3.\n";
37
38 // location represents pair of vector iterators
39 std::pair< std::vector< int >::iterator,
40 std::vector< int >::iterator > location;
41
42 // check for mismatch between v1 and v3
43 location = std::mismatch(v1.begin(), v1.end(), v3.begin());
44 cout << "\nThere is a mismatch between v1 and v3 at location "
45 << (location.first - v1.begin()) << "\nwhere v1 contains "
46 << *location.first << " and v3 contains " << *location.second
47 << "\n\n";
48
49 char c1[SIZE] = "HELLO";

```

```

50 char c2[SIZE] = "BYE BYE";
51
52 // perform lexicographical comparison of c1 and c2
53 result = std::lexicographical_compare(c1, c1 + SIZE, c2, c2 + SIZE);
54 cout << c1 << (result ? " is less than " :
55 " is greater than or equal to ") << c2 << endl;
56
57 } // end main

```

```

Vector v1 contains: 1 2 3 4 5 6 7 8 9 10
Vector v2 contains: 1 2 3 4 5 6 7 8 9 10
Vector v3 contains: 1 2 3 4 1000 6 7 8 9 10

```

Vector v1 is equal to vector v2.  
 Vector v1 is not equal to vector v3.

There is a mismatch between v1 and v3 at location 4  
 where v1 contains 5 and v3 contains 1000

HELLO is greater than or equal to BYE BYE

Line 29 uses function `equal` to compare two sequences of values for equality. Each sequence need not necessarily contain the same number of elements. `equal` returns `false` if the sequences are not of the same length. The `== operator` (whether built-in or overloaded) performs the comparison of the elements. In this example, the elements in vector `v1` from `v1.begin()` up to, but not including, `v1.end()` are compared to the elements in vector `v2` starting from `v2.begin()`. In this example, `v1` and `v2` are equal. The three iterator arguments must be at least input iterators (i.e., they can be used for input from a sequence in the forward direction). Line 34 uses function `equal` to compare vectors `v1` and `v3`, which are not equal.

[Page 1156]

There is another version of function `equal` that takes a binary predicate function as a fourth parameter. The binary predicate function receives the two elements being compared and returns a `bool` value indicating whether the elements are equal. This can be useful in sequences that store objects or pointers to values rather than actual values, because you can define one or more comparisons. For example, you can compare `Employee` objects for age, social security number, or location rather than comparing entire objects. You can compare what pointers refer to rather than comparing the pointer values (i.e., the addresses stored in the pointers).

Lines 3943 begin by instantiating a pair of iterators called `location` for a vector of integers. This object stores the result of the call to `mismatch` (line 43). Function `mismatch` compares two sequences of values and returns a pair of iterators indicating the location in each sequence of the mismatched elements. If all the elements match, the two iterators in the pair are equal to the last iterator for each sequence. The three iterator arguments must be at least input iterators. Line 45 determines the actual location of the mismatch in the vectors with the expression `location.first - v1.begin()`. The result of this calculation is the number of elements between the iterators (this is analogous to pointer arithmetic, which we studied in [Chapter 8](#)). This corresponds to the element number in this example, because the comparison is performed from the beginning of each vector. As with function `equal`, there is another version of function `mismatch` that takes a binary predicate function as a fourth parameter.

Line 53 uses function `lexicographical_compare` to compare the contents of two character arrays. This function's four iterator arguments must be at least input iterators. As you know, pointers into arrays are random-access iterators. The first two iterator arguments specify the range of locations in the first sequence. The last two specify the range of locations in the second sequence. While iterating through the sequences, the `lexicographical_compare` checks if the element in the first sequence is less than the corresponding element in the second sequence. If so, the function returns `TRue`. If the element in the first sequence is greater than or equal to the element in the second sequence, the function returns `false`. This function can be used to arrange sequences lexicographically. Typically, such sequences contain strings.

[Page 1157]

### **23.5.3. `remove`, `remove_if`, `remove_copy` and `remove_copy_if`**

[Figure 23.28](#) demonstrates removing values from a sequence with algorithms `remove`, `remove_if`, `remove_copy` and `remove_copy_if`.

#### **Figure 23.28. Algorithms `remove`, `remove_if`, `remove_copy` and `remove_copy_if`.**

(This item is displayed on pages 1157 - 1159 in the print version)

```

1 // Fig. 23.28: Fig23_28.cpp
2 // Standard Library functions remove, remove_if,
3 // remove_copy and remove_copy_if.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // algorithm definitions
9 #include <vector> // vector class-template definition
10 #include <iterator> // ostream_iterator
11
12 bool greater9(int); // prototype
13

```

```

14 int main()
15 {
16 const int SIZE = 10;
17 int a[SIZE] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
18 std::ostream_iterator< int > output(cout, " ");
19 std::vector< int > v(a, a + SIZE); // copy of a
20 std::vector< int >::iterator newLastElement;
21
22 cout << "Vector v before removing all 10s:\n ";
23 std::copy(v.begin(), v.end(), output);
24
25 // remove all 10s from v
26 newLastElement = std::remove(v.begin(), v.end(), 10);
27 cout << "\nVector v after removing all 10s:\n ";
28 std::copy(v.begin(), newLastElement, output);
29
30 std::vector< int > v2(a, a + SIZE); // copy of a
31 std::vector< int > c(SIZE, 0); // instantiate vector c
32 cout << "\n\nVector v2 before removing all 10s and copying:\n ";
33 std::copy(v2.begin(), v2.end(), output);
34
35 // copy from v2 to c, removing 10s in the process
36 std::remove_copy(v2.begin(), v2.end(), c.begin(), 10);
37 cout << "\nVector c after removing all 10s from v2:\n ";
38 std::copy(c.begin(), c.end(), output);
39
40 std::vector< int > v3(a, a + SIZE); // copy of a
41 cout << "\n\nVector v3 before removing all elements"
42 << "\ngreater than 9:\n ";
43 std::copy(v3.begin(), v3.end(), output);
44
45 // remove elements greater than 9 from v3
46 newLastElement = std::remove_if(v3.begin(), v3.end(), greater9);
47 cout << "\nVector v3 after removing all elements"
48 << "\ngreater than 9:\n ";
49 std::copy(v3.begin(), newLastElement, output);
50
51 std::vector< int > v4(a, a + SIZE); // copy of a
52 std::vector< int > c2(SIZE, 0); // instantiate vector c2
53 cout << "\n\nVector v4 before removing all elements"
54 << "\ngreater than 9 and copying:\n ";
55 std::copy(v4.begin(), v4.end(), output);
56
57 // copy elements from v4 to c2, removing elements greater
58 // than 9 in the process
59 std::remove_copy_if(v4.begin(), v4.end(), c2.begin(), greater9);
60 cout << "\nVector c2 after removing all elements"
61 << "\ngreater than 9 from v4:\n ";
62 std::copy(c2.begin(), c2.end(), output);

```

```

63 cout << endl;
64 return 0;
65 } // end main
66
67 // determine whether argument is greater than 9
68 bool greater9(int x)
69 {
70 return x > 9;
71 } // end function greater9

```

Vector v before removing all 10s:

10 2 10 4 16 6 14 8 12 10

Vector v after removing all 10s:

2 4 16 6 14 8 12

Vector v2 before removing all 10s and copying:

10 2 10 4 16 6 14 8 12 10

Vector c after removing all 10s from v2:

2 4 16 6 14 8 12 0 0 0

Vector v3 before removing all elements  
greater than 9:

10 2 10 4 16 6 14 8 12 10

Vector v3 after removing all elements  
greater than 9:

2 4 6 8

Vector v4 before removing all elements  
greater than 9 and copying:

10 2 10 4 16 6 14 8 12 10

Vector c2 after removing all elements  
greater than 9 from v4:

2 4 6 8 0 0 0 0 0 0

Line 26 uses function **remove** to eliminate all elements with the value 10 in the range from `v.begin()` up to, but not including, `v.end()` from `v`. The first two iterator arguments must be forward iterators so that the algorithm can modify the elements in the sequence. This function does not modify the number of elements in the `vector` or destroy the eliminated elements, but it does move all elements that are not eliminated toward the beginning of the `vector`. The function returns an iterator positioned after the last `vector` element that was not deleted. Elements from the iterator position to the end of the `vector` have undefined values (in this example, each "undefined" position has value 0).

Line 36 uses function `remove_copy` to copy all elements that do not have the value 10 in the range from `v2.begin()` up to, but not including, `v2.end()` from `v2`. The elements are placed in `c`, starting at position `c.begin()`. The iterators supplied as the first two arguments must be input iterators. The iterator supplied as the third argument must be an output iterator so that the element being copied can be inserted into the copy location. This function returns an iterator positioned after the last element copied into vector `c`. Note, in line 31, the use of the vector constructor that receives the number of elements in the vector and the initial values of those elements.

---

[Page 1159]

Line 46 uses function `remove_if` to delete all those elements in the range from `v3.begin()` up to, but not including, `v3.end()` from `v3` for which our user-defined unary predicate function `greater9` returns `true`. Function `greater9` (defined at lines 6871) returns `true` if the value passed to it is greater than 9; otherwise, it returns `false`. The iterators supplied as the first two arguments must be forward iterators so that the algorithm can modify the elements in the sequence. This function does not modify the number of elements in the vector, but it does move to the beginning of the vector all elements that are not eliminated. This function returns an iterator positioned after the last element in the vector that was not deleted. All elements from the iterator position to the end of the vector have undefined values.

Line 59 uses function `remove_copy_if` to copy all those elements in the range from `v4.begin()` up to, but not including, `v4.end()` from `v4` for which the unary predicate function `greater9` returns `true`. The elements are placed in `c2`, starting at position `c2.begin()`. The iterators supplied as the first two arguments must be input iterators. The iterator supplied as the third argument must be an output iterator so that the element being copied can be inserted into the copy location. This function returns an iterator positioned after the last element copied into `c2`.

#### **23.5.4. `replace`, `replace_if`, `replace_copy` and `replace_copy_if`**

Figure 23.29 demonstrates replacing values from a sequence using algorithms `replace`, `replace_if`, `replace_copy` and `replace_copy_if`.

**Figure 23.29. Algorithms `replace`, `replace_if`, `replace_copy` and `replace_copy_if`.**

(This item is displayed on pages 1160 - 1161 in the print version)

```

1 // Fig. 23.29: Fig23_29.cpp
2 // Standard Library functions replace, replace_if,
3 // replace_copy and replace_copy_if.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm>
9 #include <vector>
10 #include <iterator> // ostream_iterator
11
12 bool greater9(int); // predicate function prototype
13
14 int main()
15 {
16 const int SIZE = 10;
17 int a[SIZE] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
18 std::ostream_iterator< int > output(cout, " ");
19
20 std::vector< int > v1(a, a + SIZE); // copy of a
21 cout << "Vector v1 before replacing all 10s:\n" ;
22 std::copy(v1.begin(), v1.end(), output);
23
24 // replace all 10s in v1 with 100
25 std::replace(v1.begin(), v1.end(), 10, 100);
26 cout << "\nVector v1 after replacing 10s with 100s:\n" ;
27 std::copy(v1.begin(), v1.end(), output);
28
29 std::vector< int > v2(a, a + SIZE); // copy of a
30 std::vector< int > c1(SIZE); // instantiate vector c1
31 cout << "\n\nVector v2 before replacing all 10s and copying:\n" ;
32 std::copy(v2.begin(), v2.end(), output);
33
34 // copy from v2 to c1, replacing 10s with 100s
35 std::replace_copy(v2.begin(), v2.end(), c1.begin(), 10, 100);
36 cout << "\nVector c1 after replacing all 10s in v2:\n" ;
37 std::copy(c1.begin(), c1.end(), output);
38
39 std::vector< int > v3(a, a + SIZE); // copy of a
40 cout << "\n\nVector v3 before replacing values greater than 9:\n" ;
41 std::copy(v3.begin(), v3.end(), output);
42
43 // replace values greater than 9 in v3 with 100
44 std::replace_if(v3.begin(), v3.end(), greater9, 100);
45 cout << "\nVector v3 after replacing all values greater"
46 << "\nthan 9 with 100s:\n" ;
47 std::copy(v3.begin(), v3.end(), output);
48
49 std::vector< int > v4(a, a + SIZE); // copy of a

```

```

50 std::vector< int > c2(SIZE); // instantiate vector c2'
51 cout << "\n\nVector v4 before replacing all values greater "
52 << "than 9 and copying:\n ";
53 std::copy(v4.begin(), v4.end(), output);
54
55 // copy v4 to c2, replacing elements greater than 9 with 100
56 std::replace_copy_if(
57 v4.begin(), v4.end(), c2.begin(), greater9, 100);
58 cout << "\nVector c2 after replacing all values greater "
59 << "than 9 in v4:\n ";
60 std::copy(c2.begin(), c2.end(), output);
61 cout << endl;
62 return 0;
63 } // end main
64
65 // determine whether argument is greater than 9
66 bool greater9(int x)
67 {
68 return x > 9;
69 } // end function greater9

```

Vector v1 before replacing all 10s:

10 2 10 4 16 6 14 8 12 10

Vector v1 after replacing 10s with 100s:

100 2 100 4 16 6 14 8 12 100

Vector v2 before replacing all 10s and copying:

10 2 10 4 16 6 14 8 12 10

Vector c1 after replacing all 10s in v2:

100 2 100 4 16 6 14 8 12 100

Vector v3 before replacing values greater than 9:

10 2 10 4 16 6 14 8 12 10

Vector v3 after replacing all values greater  
than 9 with 100s:

100 2 100 4 100 6 100 8 100 100

Vector v4 before replacing all values greater than 9 and copying:

10 2 10 4 16 6 14 8 12 10

Vector c2 after replacing all values greater than 9 in v4:

100 2 100 4 100 6 100 8 100 100

Line 25 uses function `replace` to replace all elements with the value 10 in the range from `v1.begin()` up to, but not including, `v1.end()` in `v1` with the new value 100. The iterators supplied as the first two arguments must be forward iterators so that the algorithm can modify the elements in the sequence.

---

[Page 1161]

Line 35 uses function `replace_copy` to copy all elements in the range from `v2.begin()` up to, but not including, `v2.end()` from `v2`, replacing all elements with the value 10 with the new value 100. The elements are copied into `c1`, starting at position `c1.begin()`. The iterators supplied as the first two arguments must be input iterators. The iterator supplied as the third argument must be an output iterator so that the element being copied can be inserted into the copy location. This function returns an iterator positioned after the last element copied into `c1`.

Line 44 uses function `replace_if` to replace all those elements in the range from `v3.begin()` up to, but not including, `v3.end()` in `v3` for which the unary predicate function `greater9` returns `TRue`. Function `greater9` (defined at lines 6669) returns `true` if the value passed to it is greater than 9; otherwise, it returns `false`. The value 100 replaces each value greater than 9. The iterators supplied as the first two arguments must be forward iterators so that the algorithm can modify the elements in the sequence.

---

[Page 1162]

Lines 5657 use function `replace_copy_if` to copy all elements in the range from `v4.begin()` up to, but not including, `v4.end()` from `v4`. Elements for which the unary predicate function `greater9` returns `TRue` are replaced with the value 100. The elements are placed in `c2`, starting at position `c2.begin()`. The iterators supplied as the first two arguments must be input iterators. The iterator supplied as the third argument must be an output iterator so that the element being copied can be inserted into the copy location. This function returns an iterator positioned after the last element copied into `c2`.

### 23.5.5. Mathematical Algorithms

Figure 23.30 demonstrates several common mathematical algorithms from the STL, including `random_shuffle`, `count`, `count_if`, `min_element`, `max_element`, `accumulate`, `for_each` and `Transform`.

#### Figure 23.30. Mathematical algorithms of the Standard Library.

(This item is displayed on pages 1162 - 1164 in the print version)

```

1 // Fig. 23.30: Fig23_30.cpp
2 // Mathematical algorithms of the Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // algorithm definitions
8 #include <numeric> // accumulate is defined here
9 #include <vector>
10 #include <iterator>
11
12 bool greater9(int); // predicate function prototype
13 void outputSquare(int); // output square of a value
14 int calculateCube(int); // calculate cube of a value
15
16 int main()
17 {
18 const int SIZE = 10;
19 int a1[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
20 std::vector< int > v(a1, a1 + SIZE); // copy of a1
21 std::ostream_iterator< int > output(cout, " ");
22
23 cout << "Vector v before random_shuffle: ";
24 std::copy(v.begin(), v.end(), output);
25
26 std::random_shuffle(v.begin(), v.end()); // shuffle elements of v
27 cout << "\nVector v after random_shuffle: ";
28 std::copy(v.begin(), v.end(), output);
29
30 int a2[SIZE] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
31 std::vector< int > v2(a2, a2 + SIZE); // copy of a2
32 cout << "\n\nVector v2 contains: ";
33 std::copy(v2.begin(), v2.end(), output);
34
35 // count number of elements in v2 with value 8
36 int result = std::count(v2.begin(), v2.end(), 8);
37 cout << "\nNumber of elements matching 8: " << result;
38
39 // count number of elements in v2 that are greater than 9
40 result = std::count_if(v2.begin(), v2.end(), greater9);
41 cout << "\nNumber of elements greater than 9: " << result;
42
43 // locate minimum element in v2
44 cout << "\n\nMinimum element in Vector v2 is: "
45 << *(std::min_element(v2.begin(), v2.end()));
46
47 // locate maximum element in v2
48 cout << "\nMaximum element in Vector v2 is: "
49 << *(std::max_element(v2.begin(), v2.end()));

```

```
50
51 // calculate sum of elements in v
52 cout << "\n\nThe total of the elements in Vector v is: "
53 << std::accumulate(v.begin(), v.end(), 0);
54
55 // output square of every element in v
56 cout << "\n\nThe square of every integer in Vector v is:\n";
57 std::for_each(v.begin(), v.end(), outputSquare);
58
59 std::vector< int > cubes(SIZE); // instantiate vector cubes
60
61 // calculate cube of each element in v; place results in cubes
62 std::transform(v.begin(), v.end(), cubes.begin(), calculateCube);
63 cout << "\n\nThe cube of every integer in Vector v is:\n";
64 std::copy(cubes.begin(), cubes.end(), output);
65 cout << endl;
66 return 0;
67 } // end main
68
69 // determine whether argument is greater than 9
70 bool greater9(int value)
71 {
72 return value > 9;
73 } // end function greater9
74
75 // output square of argument
76 void outputSquare(int value)
77 {
78 cout << value * value << ' ';
79 } // end function outputSquare
80
81 // return cube of argument
82 int calculateCube(int value)
83 {
84 return value * value * value;
85 } // end function calculateCube
```

```
Vector v before random_shuffle: 1 2 3 4 5 6 7 8 9 10
Vector v after random_shuffle: 5 4 1 3 7 8 9 10 6 2
```

```
Vector v2 contains: 100 2 8 1 50 3 8 8 9 10
Number of elements matching 8: 3
Number of elements greater than 9: 3
```

```
Minimum element in Vector v2 is: 1
Maximum element in Vector v2 is: 100
```

The total of the elements in Vector v is: 55

The square of every integer in Vector v is:  
25 16 1 9 49 64 81 100 36 4

The cube of every integer in Vector v is:  
125 64 1 27 343 512 729 1000 216 8

[Page 1164]

Line 26 uses function `random_shuffle` to reorder randomly the elements in the range from `v.begin()` up to, but not including, `v.end()` in `v`. This function takes two randomaccess iterator arguments.

Line 36 uses function `count` to count the elements with the value 8 in the range from `v2.begin()` up to, but not including, `v2.end()` in `v2`. This function requires its two iterator arguments to be at least input iterators.

Line 40 uses function `count_if` to count those elements in the range from `v2.begin()` up to, but not including, `v2.end()` in `v2` for which the predicate function `greater9` returns `True`. Function `count_if` requires its two iterator arguments to be at least input iterators.

Line 45 uses function `min_element` to locate the smallest element in the range from `v2.begin()` up to, but not including, `v2.end()` in `v2`. The function returns a forward iterator located at the smallest element or, if the range is empty, returns `v2.end()`. The function requires its two iterator arguments to be at least input iterators. A second version of this function takes as its third argument a binary function that compares the elements in the sequence. The binary function takes two arguments and returns a `bool` value.

Good Programming Practice 23.2



It is a good practice to check that the range specified in a call to `min_element` is not empty and that the return value is not the "past the end" iterator.

Line 49 uses function `max_element` to locate the largest element in the range from `v2.begin()` up to, but not including, `v2.end()` in `v2`. The function returns an input iterator located at the largest element. The function requires its two iterator arguments to be at least input iterators. A second version of this function takes as its third argument a binary predicate function that compares the elements in the sequence. The binary function takes two arguments and returns a `bool` value.

Line 53 uses function `accumulate` (the template of which is in header file `<numeric>`) to sum the values in the range from `v.begin()` up to, but not including, `v.end()` in `v`. The function's two iterator arguments must be at least input iterators and its third argument represents the initial value of the total. A second version of this function takes as its fourth argument a general function that determines how elements are accumulated. The general function must take two arguments and return a result. The first argument to this function is the current value of the accumulation. The second argument is the value of the current element in the sequence being accumulated.

[Page 1165]

Line 57 uses function `for_each` to apply a general function to every element in the range from `v.begin()` up to, but not including, `v.end()` in `v`. The general function should take the current element as an argument and should not modify that element. Function `for_each` requires its two iterator arguments to be at least input iterators.

Line 62 uses function `transform` to apply a general function to every element in the range from `v.begin()` up to, but not including, `v.end()` in `v`. The general function (the fourth argument) should take the current element as an argument, should not modify the element and should return the transformed value. Function `TRansform` requires its first two iterator arguments to be at least input iterators and its third argument to be at least an output iterator. The third argument specifies where the `Transformed` values should be placed. Note that the third argument can equal the first. Another version of `TRansform` accepts five arguments—the first two arguments are input iterators that specify a range of elements from one source container, the third argument is an input iterator that specifies the first element in another source container, the fourth argument is an output iterator that specifies where the transformed values should be placed and the last argument is a general function that takes two arguments. This version of `transform` takes one element from each of the two input sources and applies the general function to that pair of elements, then places the transformed value at the location specified by the fourth argument.

### 23.5.6. Basic Searching and Sorting Algorithms

Figure 23.31 demonstrates some basic searching and sorting capabilities of the Standard Library, including `find`, `find_if`, `sort` and `binary_search`.

**Figure 23.31. Basic searching and sorting algorithms of the Standard Library.**

(This item is displayed on pages 1165 - 1167 in the print version)

```

1 // Fig. 23.31: Fig23_31.cpp
2 // Standard Library search and sort algorithms.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // algorithm definitions
8 #include <vector> // vector class-template definition
9 #include <iterator>
10
11 bool greater10(int value); // predicate function prototype
12
13 int main()
14 {
15 const int SIZE = 10;
16 int a[SIZE] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
17 std::vector< int > v(a, a + SIZE); // copy of a
18 std::ostream_iterator< int > output(cout, " ");
19
20 cout << "Vector v contains: ";
21 std::copy(v.begin(), v.end(), output); // display output vector
22
23 // locate first occurrence of 16 in v
24 std::vector< int >::iterator location;
25 location = std::find(v.begin(), v.end(), 16);
26
27 if (location != v.end()) // found 16
28 cout << "\n\nFound 16 at location " << (location - v.begin());
29 else // 16 not found
30 cout << "\n\n16 not found";
31
32 // locate first occurrence of 100 in v
33 location = std::find(v.begin(), v.end(), 100);
34
35 if (location != v.end()) // found 100
36 cout << "\n\nFound 100 at location " << (location - v.begin());
37 else // 100 not found
38 cout << "\n\n100 not found";
39
40 // locate first occurrence of value greater than 10 in v
41 location = std::find_if(v.begin(), v.end(), greater10);
42
43 if (location != v.end()) // found value greater than 10
44 cout << "\n\nThe first value greater than 10 is " << *location

```

```

45 << "\nfound at location " << (location - v.begin());
46 else // value greater than 10 not found
47 cout << "\n\nNo values greater than 10 were found";
48
49 // sort elements of v
50 std::sort(v.begin(), v.end());
51 cout << "\n\nVector v after sort: ";
52 std::copy(v.begin(), v.end(), output);
53
54 // use binary_search to locate 13 in v
55 if (std::binary_search(v.begin(), v.end(), 13))
56 cout << "\n\n13 was found in v";
57 else
58 cout << "\n\n13 was not found in v";
59
60 // use binary_search to locate 100 in v
61 if (std::binary_search(v.begin(), v.end(), 100))
62 cout << "\n\n100 was found in v";
63 else
64 cout << "\n\n100 was not found in v";
65
66 cout << endl;
67 return 0;
68 } // end main
69
70 // determine whether argument is greater than 10
71 bool greater10(int value)
72 {
73 return value > 10;
74 } // end function greater10

```

Vector v contains: 10 2 17 5 16 8 13 11 20 7

Found 16 at location 4  
100 not found

The first value greater than 10 is 17  
found at location 2

Vector v after sort: 2 5 7 8 10 11 13 16 17 20

13 was found in v  
100 was not found in v

Line 25 uses function `find` to locate the value 16 in the range from `v.begin()` up to, but not including, `v.end()` in `v`. The function requires its two iterator arguments to be at least input iterators and returns an input iterator that either is positioned at the first element containing the value or indicates the end of the sequence (as is the case in line 33).

Line 41 uses function `find_if` to locate the first value in the range from `v.begin()` up to, but not including, `v.end()` in `v` for which the unary predicate function `greater10` returns `true`. Function `greater10` (defined at lines 7174) takes an integer and returns a `bool` value indicating whether the integer argument is greater than 10. Function `find_if` requires its two iterator arguments to be at least input iterators. The function returns an input iterator that either is positioned at the first element containing a value for which the predicate function returns `true` or indicates the end of the sequence.

Line 50 uses function `sort` to arrange the elements in the range from `v.begin()` up to, but not including, `v.end()` in `v` in ascending order. The function requires its two iterator arguments to be random-access iterators. A second version of this function takes a third argument that is a binary predicate function taking two arguments that are values in the sequence and returning a `bool` indicating the sorting order if the return value is `true`, the two elements being compared are in sorted order.

## Common Programming Error 23.5



Attempting to `sort` a container by using an iterator other than a random-access iterator is a compilation error. Function `sort` requires a random-access iterator.

Line 55 uses function `binary_search` to determine whether the value 13 is in the range from `v.begin()` up to, but not including, `v.end()` in `v`. The sequence of values must be sorted in ascending order first. Function `binary_search` requires its two iterator arguments to be at least forward iterators. The function returns a `bool` indicating whether the value was found in the sequence. Line 61 demonstrates a call to function `binary_search` in which the value is not found. A second version of this function takes a fourth argument that is a binary predicate function taking two arguments that are values in the sequence and returning a `bool`. The predicate function returns `true` if the two elements being compared are in sorted order.

### 23.5.7. `swap`, `iter_swap` and `swap_ranges`

Figure 23.32 demonstrates algorithms `swap`, `iter_swap` and `swap_ranges` for swapping elements. Line 20 uses function `swap` to exchange two values. In this example, the first and second elements of array `a` are exchanged. The function takes as arguments references to the two values being exchanged.

**Figure 23.32. Demonstrating swap, iter\_swap and swap\_ranges.**

```
1 // Fig. 23.32: Fig23_32.cpp
2 // Standard Library algorithms iter_swap, swap and swap_ranges.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // algorithm definitions
8 #include <iterator>
9
10 int main()
11 {
12 const int SIZE = 10;
13 int a[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14 std::ostream_iterator< int > output(cout, " ");
15
16 cout << "Array a contains:\n ";
17 std::copy(a, a + SIZE, output); // display array a
18
19 // swap elements at locations 0 and 1 of array a
20 std::swap(a[0], a[1]);
21
22 cout << "\nArray a after swapping a[0] and a[1] using swap:\n ";
23 std::copy(a, a + SIZE, output); // display array a
24
25 // use iterators to swap elements at locations 0 and 1 of array a
26 std::iter_swap(&a[0], &a[1]); // swap with iterators
27 cout << "\nArray a after swapping a[0] and a[1] using iter_swap:\n ";
28 std::copy(a, a + SIZE, output);
29
30 // swap elements in first five elements of array a with
31 // elements in last five elements of array a
32 std::swap_ranges(a, a + 5, a + 5);
33
34 cout << "\nArray a after swapping the first five elements\n"
35 << "with the last five elements:\n ";
36 std::copy(a, a + SIZE, output);
37 cout << endl;
38 return 0;
39 } // end main
```

```

Array a contains:
 1 2 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using swap:
 2 1 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using iter_swap:
 1 2 3 4 5 6 7 8 9 10
Array a after swapping the first five elements
with the last five elements:
 6 7 8 9 10 1 2 3 4 5

```

[Page 1169]

Line 26 uses function `iter_swap` to exchange the two elements. The function takes two forward iterator arguments (in this case, pointers to elements of an array) and exchanges the values in the elements to which the iterators refer.

Line 32 uses function `swap_ranges` to exchange the elements in the range from `a` up to, but not including, `a + 5` with the elements beginning at position `a + 5`. The function requires three forward iterator arguments. The first two arguments specify the range of elements in the first sequence that will be exchanged with the elements in the second sequence starting from the iterator in the third argument. In this example, the two sequences of values are in the same array, but the sequences can be from different arrays or containers.

### 23.5.8. `copy_backward`, `merge`, `unique` and `reverse`

Figure 23.33 demonstrates STL algorithms `copy_backward`, `merge`, `unique` and `reverse`. Line 28 uses function `copy_backward` to copy elements in the range from `v1.begin()` up to, but not including, `v1.end()` in `v1`, placing the elements in `results` by starting from the element before `results.end()` and working toward the beginning of the vector. The function returns an iterator positioned at the last element copied into the `results` (i.e., the beginning of `results`, because of the backward copy). The elements are placed in `results` in the same order as `v1`. This function requires three bidirectional iterator arguments (iterators that can be incremented and decremented to iterate forward and backward through a sequence, respectively). The main difference between `copy` and `copy_backward` is that the iterator returned from `copy` is positioned after the last element copied and the iterator returned from `copy_backward` is positioned at the last element copied (which is really the first element in the sequence). Also, `copy` requires two input iterators and an output iterator as argument.

### Figure 23.33. Demonstrating `copy_backward`, `merge`, `unique` and `reverse`.

(This item is displayed on pages 1169 - 1170 in the print version)

```

1 // Fig. 23.33: Fig23_33.cpp
2 // Standard Library functions copy_backward, merge, unique and reverse.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // algorithm definitions
8 #include <vector> // vector class-template definition
9 #include <iterator> // ostream_iterator
10
11 int main()
12 {
13 const int SIZE = 5;
14 int a1[SIZE] = { 1, 3, 5, 7, 9 };
15 int a2[SIZE] = { 2, 4, 5, 7, 9 };
16 std::vector< int > v1(a1, a1 + SIZE); // copy of a1
17 std::vector< int > v2(a2, a2 + SIZE); // copy of a2
18 std::ostream_iterator< int > output(cout, " ");
19
20 cout << "Vector v1 contains: ";
21 std::copy(v1.begin(), v1.end(), output); // display vector output
22 cout << "\nVector v2 contains: ";
23 std::copy(v2.begin(), v2.end(), output); // display vector output
24
25 std::vector< int > results(v1.size());
26
27 // place elements of v1 into results in reverse order
28 std::copy_backward(v1.begin(), v1.end(), results.end());
29 cout << "\n\nAfter copy_backward, results contains: ";
30 std::copy(results.begin(), results.end(), output);
31
32 std::vector< int > results2(v1.size() + v2.size());
33
34 // merge elements of v1 and v2 into results2 in sorted order
35 std::merge(v1.begin(), v1.end(), v2.begin(), v2.end(),
36 results2.begin());
37
38 cout << "\n\nAfter merge of v1 and v2 results2 contains:\n";
39 std::copy(results2.begin(), results2.end(), output);
40
41 // eliminate duplicate values from results2
42 std::vector< int >::iterator endLocation;
43 endLocation = std::unique(results2.begin(), results2.end());
44
45 cout << "\n\nAfter unique results2 contains:\n";
46 std::copy(results2.begin(), endLocation, output);

```

```

47
48 cout << "\n\nVector v1 after reverse: ";
49 std::reverse(v1.begin(), v1.end()); // reverse elements of v1
50 std::copy(v1.begin(), v1.end(), output);
51 cout << endl;
52 return 0;
53 } // end main

```

```

Vector v1 contains: 1 3 5 7 9
Vector v2 contains: 2 4 5 7 9

After copy_backward, results contains: 1 3 5 7 9

After merge of v1 and v2 results2 contains:
1 2 3 4 5 5 7 7 9 9

After unique results2 contains:
1 2 3 4 5 7 9

Vector v1 after reverse: 9 7 5 3 1

```

[Page 1170]

Lines 3536 use function `merge` to combine two sorted ascending sequences of values into a third sorted ascending sequence. The function requires five iterator arguments. The first four must be at least input iterators and the last must be at least an output iterator. The first two arguments specify the range of elements in the first sorted sequence (`v1`), the second two arguments specify the range of elements in the second sorted sequence (`v2`) and the last argument specifies the starting location in the third sequence (`results2`) where the elements will be merged. A second version of this function takes as its sixth argument a binary predicate function that specifies the sorting order.

[Page 1171]

Note that line 32 creates vector `results2` with the number of elements `v1.size() + v2.size()`. Using the `merge` function as shown here requires that the sequence where the results are stored be at least the size of the two sequences being merged. If you do not want to allocate the number of elements for the resulting sequence before the `merge` operation, you can use the following statements:

```
std::vector< int > results2();
std::merge(v1.begin(), v1.end(), v2.begin(), v2.end(),
 std::back_inserter(results2));
```

The argument `std::back_inserter( results2 )` uses function template `back_inserter` (header file `<iostream>`) for the container `results2`. A `back_inserter` calls the container's default `push_back` function to insert an element at the end of the container. More importantly, if an element is inserted into a container that has no more space available, the container grows in size. Thus, the number of elements in the container does not have to be known in advance. There are two other inserters `front_inserter` (to insert an element at the beginning of a container specified as its argument) and `inserter` (to insert an element before the iterator supplied as its second argument in the container supplied as its first argument).

Line 43 uses function `unique` on the sorted sequence of elements in the range from `results2.begin()` up to, but not including, `results2.end()` in `results2`. After this function is applied to a sorted sequence with duplicate values, only a single copy of each value remains in the sequence. The function takes two arguments that must be at least forward iterators. The function returns an iterator positioned after the last element in the sequence of unique values. The values of all elements in the container after the last unique value are undefined. A second version of this function takes as a third argument a binary predicate function specifying how to compare two elements for equality.

Line 49 uses function `reverse` to reverse all the elements in the range from `v1.begin()` up to, but not including, `v1.end()` in `v1`. The function takes two arguments that must be at least bidirectional iterators.

### 23.5.9. `inplace_merge`, `unique_copy` and `reverse_copy`

Figure 23.34 demonstrates STL algorithms `inplace_merge`, `unique_copy` and `reverse_copy`. Line 24 uses function `inplace_merge` to merge two sorted sequences of elements in the same container. In this example, the elements from `v1.begin()` up to, but not including, `v1.begin() + 5` are merged with the elements from `v1.begin() + 5` up to, but not including, `v1.end()`. This function requires its three iterator arguments to be at least bidirectional iterators. A second version of this function takes as a fourth argument a binary predicate function for comparing elements in the two sequences.

**Figure 23.34. Demonstrating `inplace_merge`, `unique_copy` and `reverse_copy`.**

(This item is displayed on pages 1171 - 1172 in the print version)

```

1 // Fig. 23.34: Fig23_34.cpp
2 // Standard Library algorithms inplace_merge,
3 // reverse_copy and unique_copy.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // algorithm definitions
9 #include <vector> // vector class-template definition
10 #include <iterator> // back_inserter definition
11
12 int main()
13 {
14 const int SIZE = 10;
15 int a1[SIZE] = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
16 std::vector< int > v1(a1, a1 + SIZE); // copy of a
17 std::ostream_iterator< int > output(cout, " ");
18
19 cout << "Vector v1 contains: ";
20 std::copy(v1.begin(), v1.end(), output);
21
22 // merge first half of v1 with second half of v1 such that
23 // v1 contains sorted set of elements after merge
24 std::inplace_merge(v1.begin(), v1.begin() + 5, v1.end());
25
26 cout << "\nAfter inplace_merge, v1 contains: ";
27 std::copy(v1.begin(), v1.end(), output);
28
29 std::vector< int > results1;
30
31 // copy only unique elements of v1 into results1
32 std::unique_copy(
33 v1.begin(), v1.end(), std::back_inserter(results1));
34 cout << "\nAfter unique_copy results1 contains: ";
35 std::copy(results1.begin(), results1.end(), output);
36
37 std::vector< int > results2;
38
39 // copy elements of v1 into results2 in reverse order
40 std::reverse_copy(
41 v1.begin(), v1.end(), std::back_inserter(results2));
42 cout << "\nAfter reverse_copy, results2 contains: ";
43 std::copy(results2.begin(), results2.end(), output);
44 cout << endl;
45 return 0;
46 } // end main

```

```

Vector v1 contains: 1 3 5 7 9 1 3 5 7 9
After inplace_merge, v1 contains: 1 1 3 3 5 5 7 7 9 9
After unique_copy results1 contains: 1 3 5 7 9
After reverse_copy, results2 contains: 9 9 7 7 5 5 3 3 1 1

```

[Page 1172]

Lines 3233 use function `unique_copy` to make a copy of all the unique elements in the sorted sequence of values from `v1.begin()` up to, but not including, `v1.end()`. The copied elements are placed into vector `results1`. The first two arguments must be at least input iterators and the last must be at least an output iterator. In this example, we did not preallocate enough elements in `results1` to store all the elements copied from `v1`. Instead, we use function `back_inserter` (defined in header file `<iterator>`) to add elements to the end of `v1`. The `back_inserter` uses class `vector`'s capability to insert elements at the end of the `vector`. Because the `back_inserter` inserts an element rather than replacing an existing element's value, the `vector` is able to grow to accommodate additional elements. A second version of the `unique_copy` function takes as a fourth argument a binary predicate function for comparing elements for equality.

[Page 1173]

Lines 4041 use function `reverse_copy` to make a reversed copy of the elements in the range from `v1.begin()` up to, but not including, `v1.end()`. The copied elements are inserted into `results2` using a `back_inserter` object to ensure that the `vector` can grow to accommodate the appropriate number of elements copied. Function `reverse_copy` requires its first two iterator arguments to be at least bidirectional iterators and its third to be at least an output iterator.

## 23.5.10. Set Operations

**Figure 23.35** demonstrates Standard Library functions `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` and `set_union` for manipulating sets of sorted values. To demonstrate that Standard Library functions can be applied to arrays and containers, this example uses only arrays (remember, a pointer into an array is a randomaccess iterator).

### Figure 23.35. set operations of the Standard Library.

(This item is displayed on pages 1173 - 1175 in the print version)

```

1 // Fig. 23.35: Fig23_35.cpp
2 // Standard Library algorithms includes, set_difference,
3 // set_intersection, set_symmetric_difference and set_union.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // algorithm definitions
9 #include <iterator> // ostream_iterator
10
11 int main()
12 {
13 const int SIZE1 = 10, SIZE2 = 5, SIZE3 = 20;
14 int a1[SIZE1] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
15 int a2[SIZE2] = { 4, 5, 6, 7, 8 };
16 int a3[SIZE2] = { 4, 5, 6, 11, 15 };
17 std::ostream_iterator< int > output(cout, " ");
18
19 cout << "a1 contains: ";
20 std::copy(a1, a1 + SIZE1, output); // display array a1
21 cout << "\na2 contains: ";
22 std::copy(a2, a2 + SIZE2, output); // display array a2
23 cout << "\na3 contains: ";
24 std::copy(a3, a3 + SIZE2, output); // display array a3
25
26 // determine whether set a2 is completely contained in a1
27 if (std::includes(a1, a1 + SIZE1, a2, a2 + SIZE2))
28 cout << "\n\na1 includes a2";
29 else
30 cout << "\n\na1 does not include a2";
31
32 // determine whether set a3 is completely contained in a1
33 if (std::includes(a1, a1 + SIZE1, a3, a3 + SIZE2))
34 cout << "\n\na1 includes a3";
35 else
36 cout << "\n\na1 does not include a3";
37
38 int difference[SIZE1];
39
40 // determine elements of a1 not in a2
41 int *ptr = std::set_difference(a1, a1 + SIZE1,
42 a2, a2 + SIZE2, difference);
43 cout << "\n\nset_difference of a1 and a2 is: ";
44 std::copy(difference, ptr, output);
45
46 int intersection[SIZE1];
47
48 // determine elements in both a1 and a2
49 ptr = std::set_intersection(a1, a1 + SIZE1,

```

```

50 a2, a2 + SIZE2, intersection);
51 cout << "\n\nset_intersection of a1 and a2 is: ";
52 std::copy(intersection, ptr, output);
53
54 int symmetric_difference[SIZE1 + SIZE2];
55
56 // determine elements of a1 that are not in a2 and
57 // elements of a2 that are not in a1
58 ptr = std::set_symmetric_difference(a1, a1 + SIZE1,
59 a3, a3 + SIZE2, symmetric_difference);
60 cout << "\n\nset_symmetric_difference of a1 and a3 is: ";
61 std::copy(symmetric_difference, ptr, output);
62
63 int unionSet[SIZE3];
64
65 // determine elements that are in either or both sets
66 ptr = std::set_union(a1, a1 + SIZE1, a3, a3 + SIZE2, unionSet);
67 cout << "\n\nset_union of a1 and a3 is: ";
68 std::copy(unionSet, ptr, output);
69 cout << endl;
70 return 0;
71 } // end main

```

```

al1 contains: 1 2 3 4 5 6 7 8 9 10
al2 contains: 4 5 6 7 8
al3 contains: 4 5 6 11 15

al1 includes al2
al1 does not include al3

set_difference of al1 and al2 is: 1 2 3 9 10

set_intersection of al1 and al2 is: 4 5 6 7 8

set_symmetric_difference of al1 and al3 is: 1 2 3 7 8 9 10 11 15

set_union of al1 and al3 is: 1 2 3 4 5 6 7 8 9 10 11 15

```

Lines 27 and 33 call function **includes** in the conditions of **if** statements. Function **includes** compares two sets of sorted values to determine whether every element of the second set is in the first set. If so, **includes** returns **True**; otherwise, it returns **false**. The first two iterator arguments must be at least input iterators and must describe the first set of values. In line 27, the first set consists of the elements

from `a1` up to, but not including, `a1 + SIZE1`. The last two iterator arguments must be at least input iterators and must describe the second set of values. In this example, the second set consists of the elements from `a2` up to, but not including, `a2 + SIZE2`. A second version of function `includes` takes a fifth argument that is a binary predicate function for comparing elements for equality.

---

[Page 1175]

Lines 4142 use function `set_difference` to find the elements from the first set of sorted values that are not in the second set of sorted values (both sets of values must be in ascending order). The elements that are different are copied into the fifth argument (in this case, the array `difference`). The first two iterator arguments must be at least input iterators for the first set of values. The next two iterator arguments must be at least input iterators for the second set of values. The fifth argument must be at least an output iterator indicating where to store a copy of the values that are different. The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points. A second version of function `set_difference` takes a sixth argument that is a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

Lines 4950 use function `set_intersection` to determine the elements from the first set of sorted values that are in the second set of sorted values (both sets of values must be in ascending order). The elements common to both sets are copied into the fifth argument (in this case, array `intersection`). The first two iterator arguments must be at least input iterators for the first set of values. The next two iterator arguments must be at least input iterators for the second set of values. The fifth argument must be at least an output iterator indicating where to store a copy of the values that are the same. The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points. A second version of function `set_intersection` takes a sixth argument that is a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

Lines 5859 use function `set_symmetric_difference` to determine the elements in the first set that are not in the second set and the elements in the second set that are not in the first set (both sets must be in ascending order). The elements that are different are copied from both sets into the fifth argument (the array `symmetric_difference`). The first two iterator arguments must be at least input iterators for the first set of values. The next two iterator arguments must be at least input iterators for the second set of values. The fifth argument must be at least an output iterator indicating where to store a copy of the values that are different. The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points. A second version of function `set_symmetric_difference` takes a sixth argument that is a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

---

[Page 1176]

Line 66 uses function `set_union` to create a set of all the elements that are in either or both of the two

sorted sets (both sets of values must be in ascending order). The elements are copied from both sets into the fifth argument (in this case the array `unionSet`). Elements that appear in both sets are only copied from the first set. The first two iterator arguments must be at least input iterators for the first set of values. The next two iterator arguments must be at least input iterators for the second set of values. The fifth argument must be at least an output iterator indicating where to store the copied elements. The function returns an output iterator positioned immediately after the last value copied into the set to which the fifth argument points. A second version of function `set_union` takes a sixth argument that is a binary predicate function indicating the order in which the elements were originally sorted. The two sequences must be sorted using the same comparison function.

### 23.5.11. `lower_bound`, `upper_bound` and `equal_range`

**Figure 23.36** demonstrates Standard Library functions `lower_bound`, `upper_bound` and `equal_range`. Line 24 uses function `lower_bound` to find the first location in a sorted sequence of values at which the third argument could be inserted in the sequence such that the sequence would still be sorted in ascending order. The first two iterator arguments must be at least forward iterators. The third argument is the value for which to determine the lower bound. The function returns a forward iterator pointing to the position at which the insert can occur. A second version of function `lower_bound` takes as a fourth argument a binary predicate function indicating the order in which the elements were originally sorted.

**Figure 23.36. Algorithms `lower_bound`, `upper_bound` and `equal_range`.**

(This item is displayed on pages 1176 - 1178 in the print version)

```

1 // Fig. 23.36: Fig23_36.cpp
2 // Standard Library functions lower_bound, upper_bound and
3 // equal_range for a sorted sequence of values.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // algorithm definitions
9 #include <vector> // vector class-template definition
10 #include <iterator> // ostream_iterator
11
12 int main()
13 {
14 const int SIZE = 10;
15 int a1[SIZE] = { 2, 2, 4, 4, 4, 6, 6, 6, 6, 8 };
16 std::vector< int > v(a1, a1 + SIZE); // copy of a1
17 std::ostream_iterator< int > output(cout, " ");
18
19 cout << "Vector v contains:\n";
20 std::copy(v.begin(), v.end(), output);
21
22 // determine lower-bound insertion point for 6 in v
23 std::vector< int >::iterator lower;
```

```

24 lower = std::lower_bound(v.begin(), v.end(), 6);
25 cout << "\n\nLower bound of 6 is element "
26 << (lower - v.begin()) << " of vector v";
27
28 // determine upper-bound insertion point for 6 in v
29 std::vector< int >::iterator upper;
30 upper = std::upper_bound(v.begin(), v.end(), 6);
31 cout << "\nUpper bound of 6 is element "
32 << (upper - v.begin()) << " of vector v";
33
34 // use equal_range to determine both the lower- and
35 // upper-bound insertion points for 6
36 std::pair< std::vector< int >::iterator,
37 std::vector< int >::iterator > eq;
38 eq = std::equal_range(v.begin(), v.end(), 6);
39 cout << "\nUsing equal_range:\n Lower bound of 6 is element "
40 << (eq.first - v.begin()) << " of vector v";
41 cout << "\n Upper bound of 6 is element "
42 << (eq.second - v.begin()) << " of vector v";
43 cout << "\n\nUse lower_bound to locate the first point\n"
44 << "at which 5 can be inserted in order";
45
46 // determine lower-bound insertion point for 5 in v
47 lower = std::lower_bound(v.begin(), v.end(), 5);
48 cout << "\n Lower bound of 5 is element "
49 << (lower - v.begin()) << " of vector v";
50 cout << "\n\nUse upper_bound to locate the last point\n"
51 << "at which 7 can be inserted in order";
52
53 // determine upper-bound insertion point for 7 in v
54 upper = std::upper_bound(v.begin(), v.end(), 7);
55 cout << "\n Upper bound of 7 is element "
56 << (upper - v.begin()) << " of vector v";
57 cout << "\n\nUse equal_range to locate the first and\n"
58 << "last point at which 5 can be inserted in order";
59
60 // use equal_range to determine both the lower- and
61 // upper-bound insertion points for 5
62 eq = std::equal_range(v.begin(), v.end(), 5);
63 cout << "\n Lower bound of 5 is element "
64 << (eq.first - v.begin()) << " of vector v";
65 cout << "\n Upper bound of 5 is element "
66 << (eq.second - v.begin()) << " of vector v" << endl;
67 return 0;
68 } // end main

```

Vector v contains:

```
2 2 4 4 4 6 6 6 6 8
```

Lower bound of 6 is element 5 of vector v

Upper bound of 6 is element 9 of vector v

Using equal\_range:

Lower bound of 6 is element 5 of vector v

Upper bound of 6 is element 9 of vector v

Use lower\_bound to locate the first point  
at which 5 can be inserted in order

Lower bound of 5 is element 5 of vector v

Use upper\_bound to locate the last point  
at which 7 can be inserted in order

Upper bound of 7 is element 9 of vector v

Use equal\_range to locate the first and  
last point at which 5 can be inserted in order

Lower bound of 5 is element 5 of vector v

Upper bound of 5 is element 5 of vector v

Line 30 uses function `upper_bound` to find the last location in a sorted sequence of values at which the third argument could be inserted in the sequence such that the sequence would still be sorted in ascending order. The first two iterator arguments must be at least forward iterators. The third argument is the value for which to determine the upper bound. The function returns a forward iterator pointing to the position at which the insert can occur. A second version of function `upper_bound` takes as a fourth argument a binary predicate function indicating the order in which the elements were originally sorted.

---

[Page 1178]

Line 38 uses function `equal_range` to return a pair of forward iterators containing the combined results of performing both a `lower_bound` and an `upper_bound` operation. The first two iterator arguments must be at least forward iterators. The third argument is the value for which to locate the equal range. The function returns a pair of forward iterators for the lower bound (`eq.first`) and upper bound (`eq.second`), respectively.

Functions `lower_bound`, `upper_bound` and `equal_range` are often used to locate insertion points in sorted sequences. Line 47 uses `lower_bound` to locate the first point at which 5 can be inserted in order in `v`. Line 54 uses `upper_bound` to locate the last point at which 7 can be inserted in order in `v`. Line 62 uses `equal_range` to locate the first and last points at which 5 can be inserted in order in `v`.

## 23.5.12. Heapsort

Figure 23.37 demonstrates the Standard Library functions for performing the **heapsort sorting algorithm**. Heapsort is a sorting algorithm in which an array of elements is arranged into a special binary tree called a **heap**. The key features of a heap are that the largest element is always at the top of the heap and the values of the children of any node in the binary tree are always less than or equal to that node's value. A heap arranged in this manner is often called a **maxheap**. Heapsort is discussed in detail in computer science courses called "Data Structures" and "Algorithms."

**Figure 23.37. Using Standard Library functions to perform a heapsort.**

(This item is displayed on pages 1178 - 1180 in the print version)

```

1 // Fig. 23.37: Fig23_37.cpp
2 // Standard Library algorithms push_heap, pop_heap,
3 // make_heap and sort_heap.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm>
9 #include <vector>
10 #include <iterator>
11
12 int main()
13 {
14 const int SIZE = 10;
15 int a[SIZE] = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
16 std::vector< int > v(a, a + SIZE); // copy of a
17 std::vector< int > v2;
18 std::ostream_iterator< int > output(cout, " ");
19
20 cout << "Vector v before make_heap:\n";
21 std::copy(v.begin(), v.end(), output);
22
23 std::make_heap(v.begin(), v.end()); // create heap from vector v
24 cout << "\nVector v after make_heap:\n";
25 std::copy(v.begin(), v.end(), output);
26
27 std::sort_heap(v.begin(), v.end()); // sort elements with sort_heap
28 cout << "\nVector v after sort_heap:\n";
29 std::copy(v.begin(), v.end(), output);
30
31 // perform the heapsort with push_heap and pop_heap
32 cout << "\n\nArray a contains: ";
33 std::copy(a, a + SIZE, output); // display array a

```

```

34 cout << endl;
35
36 // place elements of array a into v2 and
37 // maintain elements of v2 in heap
38 for (int i = 0; i < SIZE; i++)
39 {
40 v2.push_back(a[i]);
41 std::push_heap(v2.begin(), v2.end());
42 cout << "\nv2 after push_heap(a[" << i << "]): ";
43 std::copy(v2.begin(), v2.end(), output);
44 } // end for
45
46 cout << endl;
47
48 // remove elements from heap in sorted order
49 for (unsigned int j = 0; j < v2.size(); j++)
50 {
51 cout << "\nv2 after " << v2[0] << " popped from heap\n";
52 std::pop_heap(v2.begin(), v2.end() - j);
53 std::copy(v2.begin(), v2.end(), output);
54 } // end for
55
56 cout << endl;
57 return 0;
58 } // end main

```

```

Vector v before make_heap:
3 100 52 77 22 31 1 98 13 40
Vector v after make_heap:
100 98 52 77 40 31 1 3 13 22
Vector v after sort_heap:
1 3 13 22 31 40 52 77 98 100

Array a contains: 3 100 52 77 22 31 1 98 13 40

v2 after push_heap(a[0]): 3
v2 after push_heap(a[1]): 100 3
v2 after push_heap(a[2]): 100 3 52
v2 after push_heap(a[3]): 100 77 52 3
v2 after push_heap(a[4]): 100 77 52 3 22
v2 after push_heap(a[5]): 100 77 52 3 22 31
v2 after push_heap(a[6]): 100 77 52 3 22 31 1
v2 after push_heap(a[7]): 100 98 52 77 22 31 1 3
v2 after push_heap(a[8]): 100 98 52 77 22 31 1 3 13
v2 after push_heap(a[9]): 100 98 52 77 40 31 1 3 13 22

v2 after 100 popped from heap
98 77 52 22 40 31 1 3 13 100

```

```

v2 after 98 popped from heap
77 40 52 22 13 31 1 3 98 100
v2 after 77 popped from heap
52 40 31 22 13 3 1 77 98 100
v2 after 52 popped from heap
40 22 31 1 13 3 52 77 98 100
v2 after 40 popped from heap
31 22 3 1 13 40 52 77 98 100
v2 after 31 popped from heap
22 13 3 1 31 40 52 77 98 100
v2 after 22 popped from heap
13 1 3 22 31 40 52 77 98 100
v2 after 13 popped from heap
3 1 13 22 31 40 52 77 98 100
v2 after 3 popped from heap
1 3 13 22 31 40 52 77 98 100
v2 after 1 popped from heap
1 3 13 22 31 40 52 77 98 100

```

[Page 1180]

Line 23 uses function `make_heap` to take a sequence of values in the range from `v.begin()` up to, but not including, `v.end()` and create a heap that can be used to produce a sorted sequence. The two iterator arguments must be random-access iterators, so this function will work only with arrays, `vectors` and `deques`. A second version of this function takes as a third argument a binary predicate function for comparing values.

Line 27 uses function `sort_heap` to sort a sequence of values in the range from `v.begin()` up to, but not including, `v.end()` that are already arranged in a heap. The two iterator arguments must be random-access iterators. A second version of this function takes as a third argument a binary predicate function for comparing values.

Line 41 uses function `push_heap` to add a new value into a heap. We take one element of array `a` at a time, append that element to the end of vector `v2` and perform the `push_heap` operation. If the appended element is the only element in the vector, the vector is already a heap. Otherwise, function `push_heap` rearranges the elements of the vector into a heap. Each time `push_heap` is called, it assumes that the last element currently in the vector (i.e., the one that is appended before the `push_heap` function call) is the element being added to the heap and that all other elements in the vector are already arranged as a heap. The two iterator arguments to `push_heap` must be random-access iterators. A second version of this function takes as a third argument a binary predicate function for comparing values.

Line 52 uses `pop_heap` to remove the top heap element. This function assumes that the elements in the range specified by its two random-access iterator arguments are already a heap. Repeatedly removing the top heap element results in a sorted sequence of values. Function `pop_heap` swaps the first heap element (`v2.begin()`, in this example) with the last heap element (the element before `v2.end() - i`, in this example), then ensures that the elements up to, but not including, the last element still form a heap. Notice in the output that, after the `pop_heap` operations, the `vector` is sorted in ascending order. A second version of this function takes as a third argument a binary predicate function for comparing values.

### 23.5.13. min and max

Algorithms `min` and `max` determine the minimum of two elements and the maximum of two elements, respectively. Figure 23.38 demonstrates `min` and `max` for `int` and `char` values.

**Figure 23.38. Algorithms min and max.**

```
1 // Fig. 23.38: Fig23_38.cpp
2 // Standard Library algorithms min and max.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm>
8
9 int main()
10 {
11 cout << "The minimum of 12 and 7 is: " << std::min(12, 7);
12 cout << "\nThe maximum of 12 and 7 is: " << std::max(12, 7);
13 cout << "\nThe minimum of 'G' and 'Z' is: " << std::min('G', 'Z');
14 cout << "\nThe maximum of 'G' and 'Z' is: " << std::max('G', 'Z');
15 cout << endl;
16 return 0;
17 } // end main
```

```
The minimum of 12 and 7 is: 7
The maximum of 12 and 7 is: 12
The minimum of 'G' and 'Z' is: G
The maximum of 'G' and 'Z' is: Z
```

### 23.5.14. STL Algorithms Not Covered in This Chapter

Figure 23.39 summarizes the STL algorithms that are not covered in this chapter.

[Page 1182]

**Figure 23.39. Algorithms not covered in this chapter.**

[Page 1183]

Algorithm	Description
inner_product	Calculate the sum of the products of two sequences by taking corresponding elements in each sequence, multiplying those elements and adding the result to a total.
adjacent_difference	Beginning with the second element in a sequence, calculate the difference (using operator <code>-</code> ) between the current and previous elements, and store the result. The first two input iterator arguments indicate the range of elements in the container and the third indicates where the results should be stored. A second version of this algorithm takes as a fourth argument a binary function to perform a calculation between the current element and the previous element.
partial_sum	Calculate a running total (using operator <code>+</code> ) of the values in a sequence. The first two input iterator arguments indicate the range of elements in the container and the third indicates where the results should be stored. A second version of this algorithm takes as a fourth argument a binary function that performs a calculation between the current value in the sequence and the running total.
nth_element	Use three random-access iterators to partition a range of elements. The first and last arguments represent the range of elements. The second argument is the partitioning element's location. After this algorithm executes, all elements before the partitioning element are less than that element and all elements after the partitioning element are greater than or equal to that element. A second version of this algorithm takes as a fourth argument a binary comparison function.

<code>partition</code>	This algorithm is similar to <code>nth_element</code> , but it requires less powerful bidirectional iterators, making it more flexible than <code>nth_element</code> . Algorithm <code>partition</code> requires two bidirectional iterators indicating the range of elements to partition. The third element is a unary predicate function that helps partition the elements so that all elements in the sequence for which the predicate is <code>true</code> are to the left (toward the beginning of the sequence) of all elements for which the predicate is <code>false</code> . A bidirectional iterator is returned indicating the first element in the sequence for which the predicate returns <code>false</code> .
<code>stable_partition</code>	This algorithm is similar to <code>partition</code> except that elements for which the predicate function returns <code>True</code> are maintained in their original order and elements for which the predicate function returns <code>false</code> are maintained in their original order.
<code>next_permutation</code>	Next lexicographical permutation of a sequence.
<code>prev_permutation</code>	Previous lexicographical permutation of a sequence.
<code>rotate</code>	Use three forward iterator arguments to rotate the sequence indicated by the first and last argument by the number of positions indicated by subtracting the first argument from the second argument. For example, the sequence 1, 2, 3, 4, 5 rotated by two positions would be 4, 5, 1, 2, 3.
<code>rotate_copy</code>	This algorithm is identical to <code>rotate</code> except that the results are stored in a separate sequence indicated by the fourth argument an output iterator. The two sequences must have the same number of elements.
<code>adjacent_find</code>	This algorithm returns an input iterator indicating the first of two identical adjacent elements in a sequence. If there are no identical adjacent elements, the iterator is positioned at the end of the sequence.
<code>search</code>	This algorithm searches for a subsequence of elements within a sequence of elements and, if such a subsequence is found, returns a forward iterator that indicates the first element of that subsequence. If there are no matches, the iterator is positioned at the end of the sequence to be searched.
<code>search_n</code>	This algorithm searches a sequence of elements looking for a subsequence in which the values of a specified number of elements have a particular value and, if such a subsequence is found, returns a forward iterator that indicates the first element of that subsequence. If there are no matches, the iterator is positioned at the end of the sequence to be searched.
<code>partial_sort</code>	Use three random-access iterators to sort part of a sequence. The first and last arguments indicate the sequence of elements. The second argument indicates the ending location for the sorted part of the sequence. By default, elements are ordered using operator <code>&lt;</code> (a binary predicate function can also be supplied). The elements from the second argument iterator to the end of the sequence are in an undefined order.

partial_sort_copy	Use two input iterators and two random-access iterators to sort part of the sequence indicated by the two input iterator arguments. The results are stored in the sequence indicated by the two random-access iterator arguments. By default, elements are ordered using operator < (a binary predicate function can also be supplied). The number of elements sorted is the smaller of the number of elements in the result and the number of elements in the original sequence.
stable_sort	The algorithm is similar to <code>sort</code> except that all equal elements are maintained in their original order.

 PREVNEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1184]

creates bitset `b`, in which every bit is initially 0. The statement

```
b.set(bitNumber);
```

sets bit `bitNumber` of bitset `b` "on." The expression `b.set()` sets all bits in `b` "on."

The statement

```
b.reset(bitNumber);
```

sets bit `bitNumber` of bitset `b` "off." The expression `b.reset()` sets all bits in `b` "off." The statement

```
b.flip(bitNumber);
```

"flips" bit `bitNumber` of bitset `b` (e.g., if the bit is on, `flip` sets it off). The expression `b.flip()` flips all bits in `b`. The statement

```
b[bitNumber];
```

returns a reference to the bit `bitNumber` of bitset `b`. Similarly,

```
b.at(bitNumber);
```

performs range checking on `bitNumber` first. Then, if `bitNumber` is in range, `at` returns a reference to the bit. Otherwise, `at` throws an `out_of_range` exception. The statement

```
b.test(bitNumber);
```

performs range checking on `bitNumber` first. Then, if `bitNumber` is in range, `test` returns `TRUE` if the bit is on, `false` if the bit is off. Otherwise, `test` throws an `out_of_range` exception. The expression

```
b.size()
```

returns the number of bits in bitset b. The expression

```
b.count()
```

returns the number of bits that are set in bitset b. The expression

```
b.any()
```

returns TRue if any bit is set in bitset b. The expression

```
b.none()
```

returns true if none of the bits is set in bitset b. The expressions

```
b == b1
b != b1
```

compare the two bitsets for equality and inequality, respectively.

Each of the bitwise assignment operators &=, |= and ^= can be used to combine bitsets. For example,

```
b &= b1;
```

performs a bit-by-bit logical AND between bitsets b and b1. The result is stored in b. Bitwise logical OR and bitwise logical XOR are performed by

---

[Page 1185]

```
b |= b1;
b ^= b2;
```

The expression

```
b >>= n;
```

shifts the bits in `bitset b` right by `n` positions. The expression

```
b <<= n;
```

shifts the bits in `bitset b` left by `n` positions. The expressions

```
b.to_string()
b.to_ulong()
```

convert `bitset b` to a string and an `unsigned long`, respectively.

Sieve of Eratosthenes with `bitset`

Figure 23.40 revisits the Sieve of Eratosthenes for finding prime numbers that we discussed in [Exercise 7.29](#). A `bitset` is used instead of an array to implement the algorithm. The program displays all the prime numbers from 2 to 1023, then allows the user to enter a number to determine whether that number is prime.

### Figure 23.40. Class `bitset` and the Sieve of Eratosthenes.

(This item is displayed on pages 1185 - 1187 in the print version)

```

1 // Fig. 23.40: Fig23_40.cpp
2 // Using a bitset to demonstrate the Sieve of Eratosthenes.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <cmath>
12 using std::sqrt; // sqrt prototype
13
14 #include <bitset> // bitset class definition
15
16 int main()
17 {
18 const int SIZE = 1024;
19 int value;
20 std::bitset< SIZE > sieve; // create bitset of 1024 bits
21 sieve.flip(); // flip all bits in bitset sieve

```

```

22 sieve.reset(0); // reset first bit (number 0)
23 sieve.reset(1); // reset second bit (number 1)
24
25 // perform Sieve of Eratosthenes
26 int finalBit = sqrt(static_cast< double >(sieve.size())) + 1;
27
28 // determine all prime numbers from 2 to 1024
29 for (int i = 2; i < finalBit; i++)
30 {
31 if (sieve.test(i)) // bit i is on
32 {
33 for (int j = 2 * i; j < SIZE; j += i)
34 sieve.reset(j); // set bit j off
35 } // end if
36 } // end for
37
38 cout << "The prime numbers in the range 2 to 1023 are:\n";
39
40 // display prime numbers in range 2-1023
41 for (int k = 2, counter = 1; k < SIZE; k++)
42 {
43 if (sieve.test(k)) // bit k is on
44 {
45 cout << setw(5) << k;
46
47 if (counter++ % 12 == 0) // counter is a multiple of 12
48 cout << '\n';
49 } // end if
50 } // end for
51
52 cout << endl;
53
54 // get value from user to determine whether value is prime
55 cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
56 cin >> value;
57
58 // determine whether user input is prime
59 while (value != -1)
60 {
61 if (sieve[value]) // prime number
62 cout << value << " is a prime number\n";
63 else // not a prime number
64 cout << value << " is not a prime number\n";
65
66 cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
67 cin >> value;
68 } // end while
69

```

```
70 return 0;
71 } // end main
```

The prime numbers in the range 2 to 1023 are:

2	3	5	7	11	13	17	19	23	29	31	37
41	43	47	53	59	61	67	71	73	79	83	89
97	101	103	107	109	113	127	131	137	139	149	151
157	163	167	173	179	181	191	193	197	199	211	223
227	229	233	239	241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337	347	349	353	359
367	373	379	383	389	397	401	409	419	421	431	433
439	443	449	457	461	463	467	479	487	491	499	503
509	521	523	541	547	557	563	569	571	577	587	593
599	601	607	613	617	619	631	641	643	647	653	659
661	673	677	683	691	701	709	719	727	733	739	743
751	757	761	769	773	787	797	809	811	821	823	827
829	839	853	857	859	863	877	881	883	887	907	911
919	929	937	941	947	953	967	971	977	983	991	997
1009	1013	1019	1021								

Enter a value from 2 to 1023 (-1 to end): 389

389 is a prime number

Enter a value from 2 to 1023 (-1 to end): 88

88 is not a prime number

Enter a value from 2 to 1023 (-1 to end): -1

Line 20 creates a bitset of size bits (size is 1024 in this example). By default, all the bits in the bitset are set "off." Line 21 calls function **flip** to set all bits "on." Numbers 0 and 1 are not prime numbers, so lines 2223 call function **reset** to set bits 0 and 1 "off." Lines 2936 determine all the prime numbers from 2 to 1023. The integer **finalBit** (line 26) is used to determine when the algorithm is complete. The basic algorithm is that a number is prime if it has no divisors other than 1 and itself. Starting with the number 2, we can eliminate all multiples of that number. The number 2 is divisible only by 1 and itself, so it is prime. Therefore, we can eliminate 4, 6, 8 and so on. The number 3 is divisible only by 1 and itself. Therefore, we can eliminate all multiples of 3 (keep in mind that all even numbers have already been eliminated).



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1188]

## Predefined Function Objects of the Standard Template Library

Many predefined function objects can be found in the header `<functional>`. Figure 23.41 lists several of the STL function objects, which are all implemented as class templates. We used the function object `less< T >` in the `set`, `multiset` and `priority_queue` examples, to specify the sorting order for elements in a container.

**Figure 23.41. Function objects in the Standard Library.**

STL function objects	Type
<code>divides&lt; T &gt;</code>	arithmetic
<code>equal_to&lt; T &gt;</code>	relational
<code>greater&lt; T &gt;</code>	relational
<code>greater_equal&lt; T &gt;</code>	relational
<code>less&lt; T &gt;</code>	relational
<code>less_equal&lt; T &gt;</code>	relational
<code>logical_and&lt; T &gt;</code>	logical
<code>logical_not&lt; T &gt;</code>	logical
<code>logical_or&lt; T &gt;</code>	logical
<code>minus&lt; T &gt;</code>	arithmetic
<code>modulus&lt; T &gt;</code>	arithmetic
<code>negate&lt; T &gt;</code>	arithmetic
<code>not_equal_to&lt; T &gt;</code>	relational
<code>plus&lt; T &gt;</code>	arithmetic

multiplies< T >	arithmetic
-----------------	------------

## Using the STL Accumulate Algorithm

Figure 23.42 demonstrates the accumulate numeric algorithm (discussed in Fig. 23.30) to calculate the sum of the squares of the elements in a vector. The fourth argument to accumulate is a **binary function object** (that is, a function object for which `operator()` takes two arguments) or a function pointer to a **binary function** (that is, a function that takes two arguments). Function `accumulate` is demonstrated twiceonce with a function pointer and once with a function object.

### Figure 23.42. Binary function object.

(This item is displayed on pages 1188 - 1189 in the print version)

```

1 // Fig. 23.42: Fig23_42.cpp
2 // Demonstrating function objects.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <vector> // vector class-template definition
8 #include <algorithm> // copy algorithm
9 #include <numeric> // accumulate algorithm
10 #include <functional> // binary_function definition
11 #include <iterator> // ostream_iterator
12
13 // binary function adds square of its second argument and the
14 // running total in its first argument, then returns the sum
15 int sumSquares(int total, int value)
16 {
17 return total + value * value;
18 } // end function sumSquares
19
20 // binary function class template defines overloaded operator()
21 // that adds the square of its second argument and running
22 // total in its first argument, then returns sum
23 template< typename T >
24 class SumSquaresClass : public std::binary_function< T, T, T >
25 {
26 public:
27 // add square of value to total and return result
28 T operator()(const T &total, const T &value)
29 {
30 return total + value * value;

```

```

31 } // end function operator()
32 }; // end class SumSquaresClass
33
34 int main()
35 {
36 const int SIZE = 10;
37 int array[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
38 std::vector< int > integers(array, array + SIZE); // copy of array
39 std::ostream_iterator< int > output(cout, " ");
40 int result;
41
42 cout << "vector integers contains:\n";
43 std::copy(integers.begin(), integers.end(), output);
44
45 // calculate sum of squares of elements of vector integers
46 // using binary function sumSquares
47 result = std::accumulate(integers.begin(), integers.end(),
48 0, sumSquares);
49
50 cout << "\n\nSum of squares of elements in integers using "
51 << "binary\nfunction sumSquares: " << result;
52
53 // calculate sum of squares of elements of vector integers
54 // using binary function object
55 result = std::accumulate(integers.begin(), integers.end(),
56 0, SumSquaresClass< int >());
57
58 cout << "\n\nSum of squares of elements in integers using "
59 << "binary\nfunction object of type "
60 << "SumSquaresClass< int >: " << result << endl;
61 return 0;
62 } // end main

```

vector integers contains:  
 1 2 3 4 5 6 7 8 9 10

Sum of squares of elements in integers using binary  
 function sumSquares: 385

Sum of squares of elements in integers using binary  
 function object of type SumSquaresClass< int >: 385

Lines 1518 define a function `sumSquares` that squares its second argument `value`, adds that square and its first argument `total` and returns the sum. Function `accumulate` will pass each of the elements of the sequence over which it iterates as the second argument to `sumSquares` in the example. On the first call to `sumSquares`, the first argument will be the initial value of the `total` (which is supplied as the third argument to `accumulate`; 0 in this program). All subsequent calls to `sumSquares` receive as the first argument the running sum returned by the previous call to `sumSquares`. When `accumulate` completes, it returns the sum of the squares of all the elements in the sequence.

Lines 2332 define a class `SumSquaresClass` that inherits from the class template `binary_function` (in header file `<functional>`) an empty base class for creating function objects in which `operator` receives two parameters and returns a value. Class `binary_function` accepts three type parameters that represent the types of the first argument, second argument and return value of `operator`, respectively. In this example, the type of these parameters is `T` (line 24). On the first call to the function object, the first argument will be the initial value of the `total` (which is supplied as the third argument to `accumulate`; 0 in this program) and the second argument will be the first element in `vector integers`. All subsequent calls to `operator` receive as the first argument the result returned by the previous call to the function object, and the second argument will be the next element in the `vector`. When `accumulate` completes, it returns the sum of the squares of all the elements in the `vector`.

Lines 4748 call function `accumulate` with a pointer to function `sumSquares` as its last argument.

The statement at lines 5556 calls function `accumulate` with an object of class `SumSquaresClass` as the last argument. The expression `SumSquaresClass< int >()` creates an instance of class `SumSquaresClass` (a function object) that is passed to `accumulate`, which sends the object the message (invokes the function) `operator`. The statement could be written as two separate statements, as follows:

```
SumSquaresClass< int > sumSquaresObject;
result = std::accumulate(integers.begin(), integers.end(),
 0, sumSquaresObject);
```

The first line defines an object of class `SumSquaresClass`. That object is then passed to function `accumulate`.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1191]

In the next chapter, we discuss several more advanced C++ features, including cast operators, namespaces, operator keywords, pointer-to-class-member operators, multiple inheritance and `virtual` base classes.

 PREV  
page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1192]

[msdn.microsoft.com/visualc](http://msdn.microsoft.com/visualc)

This is the Microsoft Visual C++ home page. Here you can find the latest Visual C++ news, updates, technical resources, samples and downloads.

[www.borland.com/cbuilder](http://www.borland.com/cbuilder)

This is the Borland C++Builder home page. Here you can find a variety of C++ resources, including several C++ newsgroups, information on the latest product enhancements, FAQs and many other resources for programmers using C++Builder.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1193]

- To use the algorithms of the STL, you must include the header file `<algorithm>`.
- Algorithm `copy` copies each element in a container starting with the location specified by the iterator in its first argument and up to but not including the location specified by the iterator in its second argument.
- Function `front` returns a reference to the first element in a sequence container. Function `begin` returns an iterator pointing to the beginning of a sequence container.
- Function `back` returns a reference to the last element in a sequence container. Function `end` returns an iterator pointing to the element one past the end of a sequence container.
- Sequence container function `insert` inserts value(s) before the element at a specific location.
- Function `erase` (available in all first-class containers) removes specific element(s) from the container.
- Function `empty` (available in all containers and adapters) returns `true` if the container is empty.
- Function `clear` (available in all first-class containers) empties the container.
- The `list` sequence container provides an efficient implementation for insertion and deletion operations at any location in the container. Header file `<list>` must be included to use class template `list`.
- The `list` member function `push_front` inserts values at the beginning of a `list`.
- The `list` member function `sort` arranges the elements in the `list` in ascending order.
- The `list` member function `splice` removes elements in one `list` and inserts them into another `list` at a specific position.
- The `list` member function `unique` removes duplicate elements in a `list`.
- The `list` member function `assign` replaces the contents of one `list` with the contents of another.
- The `list` member function `remove` deletes all copies of a specified value from a `list`.
- Class template `deque` provides the same operations as `vector`, but adds member functions `push_front` and `pop_front` to allow insertion and deletion at the beginning of a `deque`, respectively. Header file `<deque>` must be included to use class template `deque`.
- The STL's associative containers provide direct access to store and retrieve elements via keys.
- The four associative containers are `multiset`, `set`, `multimap` and `map`.
- Class templates `multiset` and `set` provide operations for manipulating sets of values where the values are the keys there is not a separate value associated with each key. Header file `<set>` must be included to use class templates `set` and `multiset`.
- The primary difference between a `multiset` and a `set` is that a `multiset` allows duplicate keys and a `set` does not.
- Class templates `multimap` and `map` provide operations for manipulating values associated with keys.
- The primary difference between a `multimap` and a `map` is that a `multimap` allows duplicate keys with associated values to be stored and a `map` allows only unique keys with associated values.
- Function `count` (available to all associative containers) counts the number of occurrences of the specified value currently in a container.
- Function `find` (available to all associative containers) locates a specified value in a container.
- Functions `lower_bound` and `upper_bound` (available in all associative containers) locate the earliest occurrence of the specified value in a container and the element after the last occurrence of the specified value in a container, respectively.
- Function `equal_range` (available in all associative containers) returns a `pair` containing the results

of both a `lower_bound` and an `upper_bound` operation.

[Page 1194]

- The `multimap` associative container is used for fast storage and retrieval of keys and associated values (often called key/value pairs).
- Duplicate keys are allowed in a `multimap`, so multiple values can be associated with a single key. This is called a one-to-many relationship.
- Header file `<map>` must be included to use class templates `map` and `multimap`.
- Duplicate keys are not allowed in a `map`, so only a single value can be associated with each key. This is called a one-to-one mapping.
- A `map` is commonly called an associative array.
- The STL provides three container adapters `stack`, `queue` and `priority_queue`.
- Adapters are not first-class containers, because they do not provide the actual data structure implementation in which elements can be stored and they do not support iterators.
- All three adapter class templates provide member functions `push` and `pop` that properly insert an element into and remove an element from each adapter data structure, respectively.
- Class template `stack` enables insertions into and deletions from the underlying data structure at one end (commonly referred to as a last-in, first-out data structure). Header file `<stack>` must be included to use class template `stack`.
- The `stack` member function `top` returns a reference to the top element of the `stack` (implemented by calling function `back` of the underlying container).
- The `stack` member function `empty` determines whether the `stack` is empty (implemented by calling function `empty` of the underlying container).
- The `stack` member function `size` returns get the number of elements in the `stack` (implemented by calling function `size` of the underlying container).
- Class template `queue` enables insertions at the `back` of the underlying data structure and deletions from the `front` of the underlying data structure (commonly referred to as a first-in, first-out data structure). Header file `<queue>` must be included to use a `queue` or a `priority_queue`.
- The `queue` member function `front` returns a reference to the first element in the `queue` (implemented by calling function `front` of the underlying container).
- The `queue` member function `back` returns a reference to the last element in the `queue` (implemented by calling function `back` of the underlying container).
- The `queue` member function `empty` determines whether the `queue` is empty (implemented by calling function `empty` of the underlying container).
- The `queue` member function `size` returns get the number of elements in the `queue` (implemented by calling function `size` of the underlying container).
- Class template `priority_queue` provides functionality that enables insertions in sorted order into the underlying data structure and deletions from the `front` of the underlying data structure.
- The common `priority_queue` operations are `push`, `pop`, `top`, `empty` and `size`.
- Algorithms `fill` and `fill_n` set every element in a range of container elements to a specific value.
- Algorithms `generate` and `generate_n` use a generator function to create values for every element in a range of container elements.
- Algorithm `equal` compares two sequences of values for equality.
- Algorithm `mismatch` compares two sequences of values and returns a pair of iterators indicating the location in each sequence of the mismatched elements.
- Algorithm `lexicographical_compare` compares the contents of two character arrays.

- Algorithm `remove` eliminates all elements with a specific value in a certain range.
- 

[Page 1195]

- Algorithm `remove_copy` copies all elements that do not have a specific value in a certain range.
- Algorithm `remove_if` deletes all elements that satisfy the `if` condition in a certain range.
- Algorithm `remove_copy_if` copies all elements that satisfy the `if` condition in a certain range.
- Algorithm `replace` replaces all elements with a specific value in certain range.
- Algorithm `replace_copy` copies all elements with a specific value in a certain range.
- Algorithm `replace_if` replaces all elements that satisfy the `if` condition in a certain range.
- Algorithm `replace_copy_if` copies all elements that satisfy the `if` condition in a certain range.
- Algorithm `random_shuffle` reorders randomly the elements in a certain range.
- Algorithm `count` counts the elements with a specific value in a certain range.
- Algorithm `count_if` counts the elements that satisfy the `if` condition in a certain range.
- Algorithm `min_element` locates the smallest element in a certain range.
- Algorithm `max_element` locates the largest element in a certain range.
- Algorithm `accumulate` sums the values in a certain range.
- Algorithm `for_each` applies a general function to every element in a certain range.
- Algorithm `transform` applies a general function to every element in a certain range and replaces each element with the result of the function.
- Algorithm `find` locates a specific value in a certain range.
- Algorithm `find_if` locates the first value in a certain range that satisfies the `if` condition.
- Algorithm `sort` arranges the elements in a certain range in ascending order or an order specified by a predicate.
- Algorithm `binary_search` determines whether a specific value is in a certain range.
- Algorithm `swap` exchanges two values.
- Algorithm `iter_swap` exchanges the two elements.
- Algorithm `swap_ranges` exchanges the elements in a certain range.
- Algorithm `copy_backward` copies elements in a certain range and places the elements in results backward.
- Algorithm `merge` combines two sorted ascending sequences of values into a third sorted ascending sequence.
- Algorithm `unique` removes duplicated elements in a sorted sequence of elements in a certain range.
- Algorithm `reverse` reverses all the elements in a certain range.
- Algorithm `inplace_merge` merges two sorted sequences of elements in the same container.
- Algorithm `unique_copy` makes a copy of all the unique elements in the sorted sequence of values in a certain range.
- Algorithm `reverse_copy` makes a reversed copy of the elements in a certain range.
- The `set` function `includes` compares two `sets` of sorted values to determine whether every element of the second `set` is in the first `set`.
- The `set` function `set_difference` finds the elements from the first `set` of sorted values that are not in the second `set` of sorted values (both `sets` of values must be in ascending order).
- The `set` function `set_intersection` determines the elements from the first `set` of sorted values that are in the second `set` of sorted values (both `sets` of values must be in ascending order).
- The `set` function `set_symmetric_difference` determines the elements in the first `set` that are not in the second `set` and the elements in the second `set` that are not in the first `set` (both `sets` of values must be in ascending order).

- The `set` function `set_union` creates a `set` of all the elements that are in either or both of the two sorted `sets` (both `sets` of values must be in ascending order).
- Algorithm `lower_bound` finds the first location in a sorted sequence of values at which the third argument could be inserted in the sequence such that the sequence would still be sorted in ascending order.
- Algorithm `upper_bound` finds the last location in a sorted sequence of values at which the third argument could be inserted in the sequence such that the sequence would still be sorted in ascending order.
- Algorithm `make_heap` takes a sequence of values in a certain range and creates a heap that can be used to produce a sorted sequence.
- Algorithm `sort_heap` sorts a sequence of values in a certain range that are already arranged in a heap.
- Algorithm `pop_heap` removes the top heap element.
- Algorithms `min` and `max` determine the minimum of two elements and the maximum of two elements, respectively.
- Class template `bitset` makes it easy to create and manipulate bit sets, which are useful for representing a set of bit flags.
- A function object is an instance of a class that overloads `operator()`.
- STL provides many predefined function objects, which can be found in header `<functional>`.
- Binary function objects are function objects that take two arguments and return a value. Class template `binary_function` is an empty base class for creating binary function objects.

 PREVNEXT 

page footer

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1197]

heap

heapsort sorting algorithm

includes algorithm

inplace\_merge algorithm

input iterator

input sequence

insert member function of containers

inserter function template

istream\_iterator

iterator

iter\_swap algorithm

key/value pair

less< int >

lexicographical\_compare algorithm

<list> header file

list sequence container

lower\_bound algorithm

`lower_bound` function of associative container

`make_heap` algorithm

`<map>` header file

`map` associative container

`max` algorithm

`max_element` algorithm

`merge` algorithm

`min` algorithm

`min_element` algorithm

`mismatch` algorithm

`multimap` associative container

`multiset` associative container

mutating-sequence algorithm

near container

`<numeric>` header file

one-to-one mapping

`ostream_iterator`

output iterator

output sequence

`pop_back` function

`pop_front` function

`pop_heap` algorithm

`pop` member function of container adapters

`priority_queue` adapter class template

`push_heap` algorithm

`push` member function of container adapters

`queue` adapter class template

`<queue>` header file

random-access iterator

`random_shuffle` algorithm

range

`rbegin` member function of `vector`

`remove` algorithm

`remove` member function of `list`

`remove_copy` algorithm

`remove_copy_if` algorithm

`remove_if` algorithm

`rend` member function of containers

`replace` algorithm

`replace_copy` algorithm

replace\_copy\_if algorithm

replace\_if algorithm

reset function of bitset

reverse algorithm

reverse\_copy algorithm

reverse\_iterator

search key

second data member of pair

sequence

sequence container

set associative container

set\_difference algorithm

<set> header file

set\_intersection algorithm

set\_symmetric\_difference algorithm

set\_union algorithm

size member function of containers

sort algorithm

sort\_heap algorithm

sort member function of list

splice member function of list

stack adapter class template

<stack> header file

Standard Template Library (STL)

swap algorithm

swap member function of list

swap\_range algorithm

top member function of container adapters

unique algorithm

unique\_copy algorithm

unique member function of list

upper\_bound algorithm

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1198]

### 23.3

The five main iterator types are \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ and \_\_\_\_\_.

### 23.4

(T/F) An iterator acts like a pointer to an element.

### 23.5

(T/F) STL algorithms can operate on C-like pointer-based arrays.

### 23.6

(T/F) STL algorithms are encapsulated as member functions within each container class.

### 23.7

(T/F) The `remove` algorithm does not decrease the `size` of the `vector` from which elements are being removed.

### 23.8

The three STL container adapters are \_\_\_\_\_, \_\_\_\_\_ and \_\_\_\_\_.

### 23.9

(T/F) Container member function `end` yields the position of the last element of the container.

### 23.10

STL algorithms operate on container elements indirectly, using \_\_\_\_\_.

### 23.11

The `sort` algorithm requires a \_\_\_\_\_ iterator.

 PREV

NEXT 

## page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1198 (continued)]

## Answers to Self-Review Exercises

- 23.1** False. These were avoided for performance reasons.
- 23.2** Associative.
- 23.3** Input, output, forward, bidirectional, random access.
- 23.4** False. It is actually vice versa.
- 23.5** True.
- 23.6** False. STL algorithms are not member functions. They operate indirectly on containers, through iterators.
- 23.7** True.
- 23.8** stack, queue, priority\_queue.
- 23.9** False. It actually yields the position just after the end of the container.
- 23.10** Iterators.
- 23.11** Random-access.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1198 (continued)]

## Exercises

- 23.12** Write a function template `palindrome` that takes a `vector` parameter and returns `TRUE` or `false` according to whether the `vector` does or does not read the same forward as backward (e.g., a `vector` containing `1, 2, 3, 2, 1` is a palindrome, but a `vector` containing `1, 2, 3, 4` is not).
- 23.13** Modify [Fig. 23.40](#), the Sieve of Eratosthenes, so that, if the number the user inputs into the program is not prime, the program displays the prime factors of the number. Remember that a prime number's factors are only 1 and the prime number itself. Every nonprime number has a unique prime factorization. For example, the factors of 54 are 2, 3, 3 and 3. When these values are multiplied together, the result is 54. For the number 54, the prime factors output should be 2 and 3.
- 23.14** Modify [Exercise 23.13](#) so that, if the number the user inputs into the program is not prime, the program displays the prime factors of the number and the number of times each prime factor appears in the unique prime factorization. For example, the output for the number 54 should be

The unique prime factorization of 54 is: `2 * 3 * 3 * 3`

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1199]

## Recommended Reading

Ammeraal, L. *STL for C++ Programmers*. New York: John Wiley, 1997.

Austern, M. H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Reading, MA: Addison-Wesley, 1998

Glass, G., and B. Schuchert. *The STL <Primer>*. Upper Saddle River, NJ: Prentice Hall PTR, 1995.

Henricson, M., and E. Nyquist. *Industrial Strength C++: Rules and Recommendations*. Upper Saddle River, NJ: Prentice Hall, 1997.

Josuttis, N. *The C++ Standard Library: A Tutorial and Handbook*. Reading, MA: Addison-Wesley, 1999.

Koenig, A., and B. Moo. *Ruminations on C++*. Reading, MA: Addison-Wesley, 1997.

Meyers, S. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Reading, MA: Addison-Wesley, 2001.

Musser, D. R., and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Reading, MA: Addison-Wesley, 1996.

Musser, D. R., and A. A. Stepanov. "Algorithm-Oriented Generic Libraries," *Software Practice and Experience*, Vol. 24, No. 7, July 1994.

Nelson, M. *C++ Programmer's Guide to the Standard Template Library*. Foster City, CA: Programmer's Press, 1995.

Pohl, I. *C++ Distilled: A Concise ANSI/ISO Reference and Style Guide*. Reading, MA: Addison-Wesley, 1997.

Pohl, I. *Object-Oriented Programming Using C++*, Second Edition. Reading, MA: Addison-Wesley, 1997.

Robson, R. *Using the STL: The C++ Standard Template Library*. New York: Springer Verlag, 2000.

Schildt, H. *STL Programming from the Ground Up*, New York: Osborne McGraw-Hill, 1999.

Stepanov, A., and M. Lee. "The Standard Template Library," Internet Distribution 31 October 1995 <[www.cs.rpi.edu/~musser/doc.ps](http://www.cs.rpi.edu/~musser/doc.ps)>.

Stroustrup, B. "Making a `vector` Fit for a Standard," The C++ Report, October 1994.

Stroustrup, B. The Design and Evolution of C++. Reading, MA: Addison-Wesley, 1994.

Stroustrup, B. The C++ Programming Language, Third Edition. Reading, MA: Addison-Wesley, 1997.

Vilot, M. J. "An Introduction to the Standard Template Library," The C++ Report, Vol. 6, No. 8, October 1994.

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1201]

## Outline

[24.1 Introduction](#)

[24.2 const\\_cast Operator](#)

[24.3 namespaces](#)

[24.4 Operator Keywords](#)

[24.5 mutable Class Members](#)

[24.6 Pointers to Class Members \(. \\* and ->\\*\)](#)

[24.7 Multiple Inheritance](#)

[24.8 Multiple Inheritance and virtual Base Classes](#)

[24.9 Wrap-Up](#)

[24.10 Closing Remarks](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

[Exercises](#)

[PREV](#)

[NEXT](#)

page footer



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1201 (continued)]

## 24.1. Introduction

We now consider several advanced C++ features. First, you will learn about the `const_cast` operator, which allows programmers to add or remove the `const` qualification of a variable. Next, we discuss namespaces, which can be used to ensure that every identifier in a program has a unique name and can help resolve naming conflicts caused by using libraries that have the same variable, function or class names. We then present several operator keywords that are useful for programmers who have keyboards that do not support certain characters used in operator symbols, such as `!`, `&`, `^`, `~` and `|`. We continue our discussion with the `mutable` storage-class specifier, which enables a programmer to indicate that a data member should always be modifiable, even when it appears in an object that is currently being treated as a `const` object by the program. Next we introduce two special operators that we can use with pointers to class members to access a data member or member function without knowing its name in advance. Finally, we introduce multiple inheritance, which enables a derived class to inherit the members of several base classes. As part of this introduction, we discuss potential problems with multiple inheritance and how `virtual` inheritance can be used to solve those problems.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1202]

Similarly, you could pass non-const data to a function that treats the data as if it were constant, then returns that data as a constant. In such cases, you might need to cast away the const-ness of the returned data, as we demonstrate in Fig. 24.1.

**Figure 24.1. Demonstrating operator const\_cast.**

```
1 // Fig. 24.1: fig24_01.cpp
2 // Demonstrating const_cast.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // contains prototypes for functions strcmp and strlen
8 #include <cctype> // contains prototype for function toupper
9
10 // returns the larger of two C-style strings
11 const char *maximum(const char *first, const char *second)
12 {
13 return (strcmp(first, second) >= 0 ? first : second);
14 } // end function maximum
15
16 int main()
17 {
18 char s1[] = "hello"; // modifiable array of characters
19 char s2[] = "goodbye"; // modifiable array of characters
20
21 // const_cast required to allow the const char * returned by maximum
22 // to be assigned to the char * variable maxPtr
23 char *maxPtr = const_cast< char * >(maximum(s1, s2));
24
25 cout << "The larger string is: " << maxPtr << endl;
26
27 for (size_t i = 0; i < strlen(maxPtr); i++)
28 maxPtr[i] = toupper(maxPtr[i]);
29
30 cout << "The larger string capitalized is: " << maxPtr << endl;
31 return 0;
32 } // end main
```

```
The larger string is: hello
The larger string capitalized is: HELLO
```

In this program, function `maximum` (lines 1114) receives two C-style strings as `const char *` parameters and returns a `const char *` that points to the larger of the two strings. Function `main` declares the two C-style strings as non-`const char` arrays (lines 1819); thus, these arrays are modifiable. In `main`, we wish to output the larger of the two C-style strings, then modify that C-style string by converting it to uppercase letters.

Function `maximum`'s two parameters are of type `const char *`, so the function's return type also must be declared as `const char *`. If the return type is specified as only `char *`, the compiler issues an error message indicating that the value being returned cannot be converted from `const char *` to `char *`; a dangerous conversion, because it attempts to treat data that the function believes to be `const` as if it were is `non-const` data.

[Page 1203]

Even though function `maximum` believes the data to be constant, we know that the original arrays in `main` do not contain constant data. Therefore, `main` should be able to modify the contents of those arrays as necessary. Since we know these arrays are modifiable, we use `const_cast` (line 23) to cast away the `const-ness` of the pointer returned by `maximum`, so we can then modify the data in the array representing the larger of the two C-style strings. We can then use the pointer as the name of a character array in the `for` statement (lines 2728) to convert the contents of the larger string to uppercase letters. Without the `const_cast` in line 23, this program will not compile, because you are not allowed to assign a pointer of type `const char *` to a pointer of type `char *`.

#### Error-Prevention Tip 24.1



In general, a `const_cast` should be used only when it is known in advance that the original data is not constant. Otherwise, unexpected results may occur.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1204]

## Error-Prevention Tip 24.2



Precede a member with its namespace name and the scope resolution operator (`::`) if the possibility exists of a naming conflict.

Not all namespaces are guaranteed to be unique. Two third-party vendors might inadvertently use the same identifiers for their namespace names. [Figure 24.2](#) demonstrates the use of namespaces.

**Figure 24.2. Demonstrating the use of namespaces.**

(This item is displayed on pages 1204 - 1205 in the print version)

```
1 // Fig. 24.2: fig24_02.cpp
2 // Demonstrating namespaces.
3 #include <iostream>
4 using namespace std; // use std namespace
5
6 int integer1 = 98; // global variable
7
8 // create namespace Example
9 namespace Example
10 {
11 // declare two constants and one variable
12 const double PI = 3.14159;
13 const double E = 2.71828;
14 int integer1 = 8;
15
16 void printValues(); // prototype
17
18 // nested namespace
19 namespace Inner
20 {
21 // define enumeration
22 enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
23 } // end Inner namespace
24 } // end Example namespace
```

```

25
26 // create unnamed namespace
27 namespace
28 {
29 double doubleInUnnamed = 88.22; // declare variable
30 } // end unnamed namespace
31
32 int main()
33 {
34 // output value doubleInUnnamed of unnamed namespace
35 cout << "doubleInUnnamed = " << doubleInUnnamed;
36
37 // output global variable
38 cout << "\n(global) integer1 = " << integer1;
39
40 // output values of Example namespace
41 cout << "\nPI = " << Example::PI << "E = " << Example::E
42 << "\ninteger1 = " << Example::integer1 << "\nFISCAL3 = "
43 << Example::Inner::FISCAL3 << endl;
44
45 Example::printValues(); // invoke printValues function
46 return 0;
47 } // end main
48
49 // display variable and constant values
50 void Example::printValues()
51 {
52 cout << "\nIn printValues:\ninteger1 = " << integer1 << "PI = "
53 << PI << "\nE = " << E << "\ndoubleInUnnamed = "
54 << doubleInUnnamed << "\n(global) integer1 = " << ::integer1
55 << "\nFISCAL3 = " << Inner::FISCAL3 << endl;
56 } // end printValues

```

```

doubleInUnnamed = 88.22
(global) integer1 = 98
PI = 3.14159
E = 2.71828
integer1 = 8
FISCAL3 = 1992

In printValues:
integer1 = 8
PI = 3.14159
E = 2.71828
doubleInUnnamed = 88.22
(global) integer1 = 98
FISCAL3 = 1992

```

---

[Page 1205]

## Using the std Namespace

Line 4 informs the compiler that namespace `std` is being used. The contents of header file `<iostream>` are all defined as part of namespace `std`. [Note: Most C++ programmers consider it poor practice to write a `using` directive such as line 4 because the entire contents of the namespace are included, thus increasing the likelihood of a naming conflict.]

The `using namespace` directive specifies that the members of a namespace will be used frequently throughout a program. This allows the programmer to access all the members of the namespace and to write more concise statements such as

```
cout << "double1 = " << double1;
```

rather than

```
std::cout << "double1 = " << double1;
```

Without line 4, either every `cout` and `endl` in Fig. 24.2 would have to be qualified with `std::`, or individual `using` declarations must be included for `cout` and `endl` as in:

```
using std::cout;
using std::endl;
```

The `using namespace` directive can be used for predefined namespaces (e.g., `std`) or programmer-defined namespaces.

---

[Page 1206]

## Defining Namespaces

Lines 924 use the keyword `namespace` to define namespace `Example`. The body of a namespace is

delimited by braces ({}). Namespace Example's members consist of two constants (PI and E at lines 1213), an int (integer1 at line 14), a function (printValues at line 16) and a **nested namespace** (Inner at lines 1923). Notice that member integer1 has the same name as global variable integer1 (line 6). Variables that have the same name must have different scopes otherwise compilation errors occur. A namespace can contain constants, data, classes, nested namespaces, functions, etc. Definitions of namespaces must occupy the global scope or be nested within other namespaces.

Lines 2730 create an **unnamed namespace** containing the member doubleInUnnamed. The unnamed namespace has an implicit using directive, so its members appear to occupy the **global namespace**, are accessible directly and do not have to be qualified with a namespace name. Global variables are also part of the global namespace and are accessible in all scopes following the declaration in the file.

## Software Engineering Observation 24.2



Each separate compilation unit has its own unique unnamed namespace; i.e., the unnamed namespace replaces the static linkage specifier.

## Accessing Namespace Members with Qualified Names

Line 35 outputs the value of variable doubleInUnnamed, which is directly accessible as part of the unnamed namespace. Line 38 outputs the value of global variable integer1. For both of these variables, the compiler first attempts to locate a local declaration of the variables in main. Since there are no local declarations, the compiler assumes those variables are in the global namespace.

Lines 4143 output the values of PI, E, integer1 and FISCAL3 from namespace Example. Notice that each must be qualified with Example:: because the program does not provide any using directive or declarations indicating that it will use members of namespace Example. In addition, member integer1 must be qualified, because a global variable has the same name. Otherwise, the global variable's value is output. Notice that FISCAL3 is a member of nested namespace Inner, so it must be qualified with Example::Inner::.

Function printValues (defined at lines 5056) is a member of Example, so it can access other members of the Example namespace directly without using a namespace qualifier. The output statement in lines 5255 outputs integer1, PI, E, doubleInUnnamed, global variable integer1 and FISCAL3. Notice that PI and E are not qualified with Example. Variable doubleInUnnamed is still accessible, because it is in the unnamed namespace and the variable name does not conflict with any other members of namespace Example. The global version of integer1 must be qualified with the unary scope resolution operator (::), because its name conflicts with a member of namespace Example. Also, FISCAL3 must be qualified with Inner::. When accessing members of a nested namespace, the members must be qualified with the namespace name (unless the member is being used inside the nested namespace).

## Common Programming Error 24.1



Placing `main` in a namespace is a compilation error.

---

[Page 1207]

## Aliases for Namespace Names

Namespaces can be aliased. For example the statement

```
namespace CPPHTTP5E = CPlusPlusHowToProgram5E;
```

creates the alias `CPPHTTP5E` for `CPlusPlusHowToProgram5E`.

PREV

NEXT

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1209]

The program declares and initializes two `bool` variables and two integer variables (lines 1215). Logical operations (lines 2430) are performed with `bool` variables `a` and `b` using the various logical operator keywords. Bitwise operations (lines 3339) are performed with the `int` variables `c` and `d` using the various bitwise operator keywords. The result of each operation is output.

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1210]

## Software Engineering Observation 24.3



`mutable` members are useful in classes that have "secret" implementation details that do not contribute to the logical value of an object.

### Mechanical Demonstration of a `mutable` Data Member

Figure 24.5 demonstrates using a `mutable` member. The program defines class `TestMutable` (lines 822), which contains a constructor, function `getValue` and a `private` data member `value` that is declared `mutable`. Lines 1619 define function `getValue` as a `const` member function that returns a copy of `value`. Notice that the function increments `mutable` data member `value` in the return statement. Normally, a `const` member function cannot modify data members unless the object on which the function operates i.e., the one to which `this` points is cast (using `const_cast`) to a non-`const` type. Because `value` is `mutable`, this `const` function is able to modify the data.

#### Figure 24.5. Demonstrating a `mutable` data member.

(This item is displayed on pages 1210 - 1211 in the print version)

```

1 // Fig. 24.5: fig24_05.cpp
2 // Demonstrating storage-class specifier mutable.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // class TestMutable definition
8 class TestMutable
9 {
10 public:
11 TestMutable(int v = 0)
12 {
13 value = v;
14 } // end TestMutable constructor
15
16 int getValue() const

```

```

17 {
18 return value++; // increments value
19 } // end function getValue
20 private:
21 mutable int value; // mutable member
22 } // end class TestMutable
23
24 int main()
25 {
26 const TestMutable test(99);
27
28 cout << "Initial value: " << test.getValue();
29 cout << "\nModified value: " << test.getValue() << endl;
30 return 0;
31 } // end main

```

Initial value: 99  
Modified value: 100

[Page 1211]

Line 26 declares `const TestMutable` object `test` and initializes it to 99. Line 28 calls the `const` member function `getValue`, which adds one to `value` and returns its previous contents. Notice that the compiler allows the call to member function `getValue` on the object `test` because it is a `const` object and `getValue` is a `const` member function. However, `getValue` modifies variable `value`. Thus, when line 29 invokes `getValue` again, the new value (100) is output to prove that the `mutable` data member was indeed modified.

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1212]

The program declares class `Test` (lines 817), which provides public member function `test` and public data member `value`. Lines 1920 provide prototypes for the functions `arrowStar` (defined at lines 3236) and `dotStar` (defined at lines 3943), which demonstrate the `->*` and `.*` operators, respectively. Lines 24 creates object `test`, and line 25 assigns 8 to its data member `value`. Lines 2627 call functions `arrowStar` and `dotStar` with the address of the object `test`.

Line 34 in function `arrowStar` declares and initializes variable `memPtr` as a pointer to a member function. In this declaration, `Test:::*` indicates that the variable `memPtr` is a pointer to a member of class `Test`. To declare a pointer to a function, enclose the pointer name preceded by `*` in parentheses, as in `(Test:::*memPtr)`. A pointer to a function must specify, as part of its type, both the return type of the function it points to and the parameter list of that function. The return type of the function appears to the left of the left parenthesis and the parameter list appears in a separate set of parentheses to the right of the pointer declaration. In this case, the function has a `void` return type and no parameters. The pointer `memPtr` is initialized with the address of class `Test`'s member function named `test`. Note that the header of the function must match the function pointer's declaration i.e., function `test` must have a `void` return type and no parameters. Notice that the right side of the assignment uses the address operator (`&`) to get the address of the member function `test`. Also, notice that neither the left side nor the right side of the assignment in line 34 refers to a specific object of class `Test`. Only the class name is used with the binary scope resolution operator (`:::`). Line 35 invokes the member function stored in `memPtr` (i.e., `test`), using the `->*` operator. Because `memPtr` is a pointer to a member of a class, the `->*` operator must be used rather than the `->` operator to invoke the function.

Line 41 declares and initializes `vPtr` as a pointer to an `int` data member of class `Test`. The right side of the assignment specifies the address of the data member `value`. Line 42 dereferences the pointer `testPtr2`, then uses the `.*` operator to access the member to which `vPtr` points. Note that the client code can create pointers to class members for only those class members that are accessible to the client code. In this example, both member function `test` and data member `value` are publicly accessible.

[Page 1213]

## Common Programming Error 24.2



Declaring a member-function pointer without enclosing the pointer name in parentheses is a syntax error.

## Common Programming Error 24.3



Declaring a member-function pointer without preceding the pointer name with a class name followed by the scope resolution operator (::) is a syntax error.

## Common Programming Error 24.4



Attempting to use the -> or \* operator with a pointer to a class member generates syntax errors.

[◀ PREV](#)

page footer

[NEXT ▶](#)

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1214]

Class `Base2` ([Fig. 24.8](#)) is similar to class `Base1`, except that its `protected` data is a `char` named `letter` (line 20). Like class `Base1`, `Base2` has a `public` member function `getdata`, but this function returns the value of `char` data member `letter`.

Class `Derived` ([Figs. 24.924.10](#)) inherits from both class `Base1` and class `Base2` through multiple inheritance. Class `Derived` has a `private` data member of type `double` named `real` (line 21), a constructor to initialize all the data of class `Derived` and a `public` member function `getreal` that returns the value of `double` variable `real`.

---

[Page 1215]

Notice how straightforward it is to indicate multiple inheritance by following the colon (:) after `class Derived` with a comma-separated list of base classes (line 14). In [Fig. 24.10](#), notice that constructor `Derived` explicitly calls base-class constructors for each of its base classes `Base1` and `Base2` using the member-initializer syntax (line 9). The base-class constructors are called in the order that the inheritance is specified, not in the order in which their constructors are mentioned; also, if the base-class constructors are not explicitly called in the member-initializer list, their default constructors will be called implicitly.

The overloaded stream insertion operator ([Fig. 24.10](#), lines 1823) uses its second parameter a reference to a `Derived` object to display a `Derived` object's data. This operator function is a friend of `Derived`, so `operator<<` can directly access all of class `Derived`'s `protected` and `private` members, including the `protected` data member `value` (inherited from class `Base1`), `protected` data member `letter` (inherited from class `Base2`) and `private` data member `real` (declared in class `Derived`).

Now let us examine the `main` function ([Fig. 24.11](#)) that tests the classes in [Figs. 24.724.10](#). Line 13 creates `Base1` object `base1` and initializes it to the `int` value 10, then creates the pointer `base1Ptr` and initializes it to the null pointer (i.e., 0). Line 14 creates `Base2` object `base2` and initializes it to the `char` value 'z', then creates the pointer `base2Ptr` and initializes it to the null pointer. Line 15 creates `Derived` object `derived` and initializes it to contain the `int` value 7, the `char` value 'A' and the `double` value 3.5.

---

[Page 1217]

Lines 1820 display each object's data values. For objects `base1` and `base2`, we invoke each object's `getdata` member function. Even though there are two `getdata` functions in this example, the calls are not ambiguous. In line 18, the compiler knows that `base1` is an object of class `Base1`, so class `Base1`'s

version of `getTData` is called. In line 19, the compiler knows that `base2` is an object of class `Base2` so class `Base2`'s version of `getTData` is called. Line 20 displays the contents of object derived using the overloaded stream insertion operator.

## Resolving Ambiguity Issues That Arise When a Derived Class Inherits Member Functions of the Same Name from Multiple Base Classes

Lines 2427 output the contents of object derived again by using the get member functions of class `Derived`. However, there is an ambiguity problem, because this object contains two `getData` functions, one inherited from class `Base1` and one inherited from class `Base2`. This problem is easy to solve by using the binary scope resolution operator. The expression `derived.Base1::getData()` gets the value of the variable inherited from class `Base1` (i.e., the `int` variable named `value`) and `derived.Base2::getData()` gets the value of the variable inherited from class `Base2` (i.e., the `char` variable named `letter`). The `double` value in `real` is printed without ambiguity with the call `derived.getReal()` there are no other member functions with that name in the hierarchy.

---

[Page 1218]

## Demonstrating the Is-A Relationships in Multiple Inheritance

The is-a relationships of single inheritance also apply in multiple-inheritance relationships. To demonstrate this, line 31 assigns the address of object derived to the `Base1` pointer `base1Ptr`. This is allowed because an object of class `Derived` is an object of class `Base1`. Line 32 invokes `Base1` member function `getData` via `base1Ptr` to obtain the value of only the `Base1` part of the object derived. Line 35 assigns the address of object derived to the `Base2` pointer `base2Ptr`. This is allowed because an object of class `Derived` is an object of class `Base2`. Line 36 invokes `Base2` member function `getTData` via `base2Ptr` to obtain the value of only the `Base2` part of the object derived.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1219]

**Figure 24.13. Attempting to call a multiply inherited function polymorphically.**

(This item is displayed on pages 1219 - 1220 in the print version)

```
1 // Fig. 24.13: fig24_13.cpp
2 // Attempting to polymorphically call a function that is
3 // multiply inherited from two base classes.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 // class Base definition
9 class Base
10 {
11 public:
12 virtual void print() const = 0; // pure virtual
13 }; // end class Base
14
15 // class DerivedOne definition
16 class DerivedOne : public Base
17 {
18 public:
19 // override print function
20 void print() const
21 {
22 cout << "DerivedOne\n";
23 } // end function print
24 }; // end class DerivedOne
25
26 // class DerivedTwo definition
27 class DerivedTwo : public Base
28 {
29 public:
30 // override print function
31 void print() const
32 {
33 cout << "DerivedTwo\n";
34 } // end function print
35 }; // end class DerivedTwo
36
```

```

37 // class Multiple definition
38 class Multiple : public DerivedOne, public DerivedTwo
39 {
40 public:
41 // qualify which version of function print
42 void print() const
43 {
44 DerivedTwo::print();
45 } // end function print
46 }; // end class Multiple
47
48 int main()
49 {
50 Multiple both; // instantiate Multiple object
51 DerivedOne one; // instantiate DerivedOne object
52 DerivedTwo two; // instantiate DerivedTwo object
53 Base *array[3]; // create array of base-class pointers
54
55 array[0] = &both; // ERROR--ambiguous
56 array[1] = &one;
57 array[2] = &two;
58
59 // polymorphically invoke print
60 for (int i = 0; i < 3; i++)
61 array[i] -> print();
62
63 return 0;
64 } // end main

```

C:\Projects\cpphttp5\examples\ch24\Fig24\_20\Fig24\_20.cpp(55): error C2594:  
'=' : ambiguous conversions from 'Multiple \*' to 'Base \*'

[Page 1220]

Class `Multiple` (lines 3846) inherits from both classes `DerivedOne` and `DerivedTwo`. In class `Multiple`, function `print` is overridden to call `DerivedTwo`'s `print` (line 44). Notice that we must qualify the `print` call with the class name `DerivedTwo` to specify which version of `print` to call.

Function `main` (lines 4864) declares objects of classes `Multiple` (line 50), `DerivedOne` (line 51) and

DerivedTwo (line 52). Line 53 declares an array of `Base *` pointers. Each array element is initialized with the address of an object (lines 5557). An error occurs when the address of both an object of class Multiple is assigned to `array[ 0 ]`. The object both actually contains two subobjects of type `Base`, so the compiler does not know which subobject the pointer `array[ 0 ]` should point to, and it generates a compilation error indicating an ambiguous conversion.

## Eliminating Duplicate Subobjects with virtual Base-Class Inheritance

The problem of duplicate subobjects is resolved with `virtual` inheritance. When a base class is inherited as `virtual`, only one subobject will appear in the derived class—a process called **`virtual` base-class inheritance**. Figure 24.14 revises the program of Fig. 24.13 to use a `virtual` base class.

**Figure 24.14. Using `virtual` base classes.**

(This item is displayed on pages 1220 - 1222 in the print version)

```

1 // Fig. 24.14: fig24_14.cpp
2 // Using virtual base classes.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // class Base definition
8 class Base
9 {
10 public:
11 virtual void print() const = 0; // pure virtual
12 }; // end class Base
13
14 // class DerivedOne definition
15 class DerivedOne : virtual public Base
16 {
17 public:
18 // override print function
19 void print() const
20 {
21 cout << "DerivedOne\n";
22 } // end function print
23 }; // end DerivedOne class
24
25 // class DerivedTwo definition
26 class DerivedTwo : virtual public Base
27 {
28 public:
29 // override print function
30 void print() const

```

```

31 {
32 cout << "DerivedTwo\n";
33 } // end function print
34 }; // end DerivedTwo class
35
36 // class Multiple definition
37 class Multiple : public DerivedOne, public DerivedTwo
38 {
39 public:
40 // qualify which version of function print
41 void print() const
42 {
43 DerivedTwo::print();
44 } // end function print
45 }; // end Multiple class
46
47 int main()
48 {
49 Multiple both; // instantiate Multiple object
50 DerivedOne one; // instantiate DerivedOne object
51 DerivedTwo two; // instantiate DerivedTwo object
52
53 // declare array of base-class pointers and initialize
54 // each element to a derived-class type
55 Base *array[3];
56 array[0] = &both;
57 array[1] = &one;
58 array[2] = &two;
59
60 // polymorphically invoke function print
61 for (int i = 0; i < 3; i++)
62 array[i]->print();
63
64 return 0;
65 } // end main

```

DerivedTwo  
DerivedOne  
DerivedTwo

The key change in the program is that classes `DerivedOne` (line 15) and `DerivedTwo` (line 26) each inherit from class `Base` by specifying `virtual public Base`. Since both of these classes inherit from `Base`, they each contain a `Base` subobject. The benefit of virtual inheritance is not clear until class `Multiple` inherits from both `DerivedOne` and `DerivedTwo` (line 37). Since each of the base classes used virtual inheritance to inherit class `Base`'s members, the compiler ensures that only one subobject of type `Base` is inherited into class `Multiple`. This eliminates the ambiguity error generated by the compiler in Fig. 24.13. The compiler now allows the implicit conversion of the derived-class pointer (`&both`) to the base-class pointer `array[ 0 ]` at line 56 in `main`. The `for` statement at lines 6162 polymorphically calls `print` for each object.

## Constructors in Multiple-Inheritance Hierarchies with virtual Base Classes

Implementing hierarchies with virtual base classes is simpler if default constructors are used for the base classes. The examples in Figs. 24.13 and 24.14 use compiler-generated default constructors. If a virtual base class provides a constructor that requires arguments, the implementation of the derived classes becomes more complicated, because the **most derived class** must explicitly invoke the virtual base class's constructor to initialize the members inherited from the virtual base class.

Software Engineering Observation 24.5



Providing a default constructor for virtual base classes simplifies hierarchy design.

## Additional Information on Multiple Inheritance

Multiple inheritance is a complex topic typically covered in more advanced C++ texts. The following URLs provide additional information about multiple inheritance.

[cplus.about.com/library/weekly/aa121302a.htm](http://cplus.about.com/library/weekly/aa121302a.htm)

A tutorial on multiple inheritance with a detailed example.

[cpptips.hyperformix.com/MultipleInher.html](http://cpptips.hyperformix.com/MultipleInher.html)

Provides technical tips that explain several issues regarding multiple inheritance.

[www.parashift.com/c++-faq-lite/multiple-inheritance.html](http://www.parashift.com/c++-faq-lite/multiple-inheritance.html)

Part of the C++ FAQ Lite. Provides a detailed technical explanation of multiple inheritance and virtual inheritance.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1223]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1223 (continued)]

## 24.10. Closing Remarks

We sincerely hope you have enjoyed learning C++ and object-oriented programming with C++ How to Program, 5/e. We would greatly appreciate your comments, criticisms, corrections and suggestions for improving the text. Please address all correspondence to our e-mail address:

[deitel@deitel.com](mailto:deitel@deitel.com)

Good luck!

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1224]

- A `using namespace` directive can be used for predefined namespaces (e.g., `std`) or programmer-defined namespaces.
- A namespace can contain constants, data, classes, nested namespaces, functions, etc. Definitions of namespaces must occupy the global scope or be nested within other namespaces.
- An unnamed namespace has an implicit `using` directive, so its members appear to occupy the global namespace, are accessible directly and do not have to be qualified with a namespace name. Global variables are also part of the global namespace.
- When accessing members of a nested namespace, the members must be qualified with the namespace name (unless the member is being used inside the nested namespace).
- Namespaces can be aliased.
- The C++ standard provides operator keywords that can be used in place of several C++ operators. Operator keywords are useful for programmers who have keyboards that do not support certain characters such as `!`, `&`, `^`, `~`, `|`, etc.
- If a data member should always be modifiable, C++ provides the storage-class specifier `mutable` as an alternative to `const_cast`. A `mutable` data member is always modifiable, even in a `const` member function or `const` object. This reduces the need to cast away "const-ness."
- Both `mutable` and `const_cast` allow a data member to be modified; they are used in different contexts. For a `const` object with no `mutable` data members, operator `const_cast` must be used every time a member is to be modified. This greatly reduces the chance of a member being accidentally modified because the member is not permanently modifiable.
- Operations involving `const_cast` are typically hidden in a member function's implementation. The user of a class might not be aware that a member is being modified.
- C++ provides the `.*` and `->*` operators for accessing class members via pointers. This is rarely used capability that is used primarily by advanced C++ programmers.
- Declaring a pointer to a function requires that you enclose the pointer name preceded by an `*` in parentheses. A pointer to a function must specify, as part of its type, both the return type of the function it points to and the parameter list of that function.
- In C++, a class may be derived from more than one base class—a technique known as multiple inheritance, in which a derived class inherits the members of two or more base classes.
- A common problem with multiple inheritance is that each of the base classes might contain data members or member functions that have the same name. This can lead to ambiguity problems when you attempt to compile.
- The is-a relationships of single inheritance also apply in multiple-inheritance relationships.
- Multiple inheritance is used, for example, in the C++ Standard Library to form class `basic_iostream`. Class `basic_ios` is the base class for both `basic_istream` and `basic_ostream`, each of which is formed with single inheritance. Class `basic_iostream` inherits from both `basic_istream` and `basic_ostream`. This enables objects of class `basic_iostream` to provide the functionality of both `basic_istreams` and `basic_ostreams`. In multiple-inheritance hierarchies, the situation described here is referred to as diamond inheritance.
- Because classes `basic_istream` and `basic_ostream` each inherit from `basic_ios`, a potential problem exists for `basic_iostream`. If not implemented correctly, class `basic_iostream` could contain two copies of the members of class `basic_ios` one inherited via class `basic_istream` and one

inherited via class `basic_ostream`). Such a situation would be ambiguous and would result in a compilation error, because the compiler would not know which version of the members from class `basic_ios` to use.

- The ambiguity in diamond inheritance occurs when a derived-class object inherits two or more base-class subobjects. The problem of duplicate subobjects is resolved with `virtual` inheritance. When a base class is inherited as `virtual`, only one subobject will appear in the derived class a process called `virtual` base-class inheritance.

---

[Page 1225]

- Implementing hierarchies with `virtual` base classes is simpler if default constructors are used for the base classes. If a `virtual` base class provides a constructor that requires arguments, the implementation of the derived classes becomes more complicated, because the most derived class must explicitly invoke the `virtual` base class's constructor to initialize the members inherited from the `virtual` base class.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1225 (continued)]

## Terminology

. \* operator

->\* operator

and operator keyword

and\_eq operator keyword

base-class subobject

bitand operator keyword

bitor operator keyword

bitwise assignment operator keywords

bitwise operator keywords

cast away const-ness

comma-separated list of base classes

compl operator keyword

const\_cast operator

diamond inheritance

global namespace

inequality operator keywords

logical operator keywords

most derived class

multiple inheritance

mutable data member

namespace

namespace alias

namespace keyword

naming conflict

nested namespace

not operator keyword

not\_eq operator keyword

operator keyword

or operator keyword

or\_eq operator keyword

pointer-to-member operators

unnamed namespace

using declaration

using namespace declaration

virtual base class

[virtual inheritance](#)

[volatile qualifier](#)

[xor operator keyword](#)

[xor\\_eq operator keyword](#)

[◀ PREV](#)

[NEXT ▶](#)

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1225 (continued)]

## Self-Review Exercises

**24.1** Fill in the blanks for each of the following:

a.

The \_\_\_\_\_ operator qualifies a member with its namespace.

b.

The \_\_\_\_\_ operator allows an object's "const-ness" to be cast away.

c.

Because an unnamed namespace has an implicit `using` directive, its members appear to occupy the \_\_\_\_\_, are accessible directly and do not have to be qualified with a namespace name.

d.

Operator \_\_\_\_\_ is the operator keyword for inequality.

e.

A class may be derived from more than one base class; such derivation is called \_\_\_\_\_.

f.

When a base class is inherited as \_\_\_\_\_, only one subobject of the base class will appear in the derived class.

**24.2** State which of the following are true and which are false. If a statement is false, explain why.

a.

When passing a non-const argument to a const function, the const\_cast operator should be used to cast away the "const-ness" of the function.

b.

namespaces are guaranteed to be unique.

c.

Like class bodies, namespace bodies also end in semicolons.

d.

namespaces cannot have namespaces as members.

e.

A mutable data member cannot be modified in a const member function.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1226]

## Answers to Self-Review Exercises

- 24.1** a) binary scope resolution (>::). b) `const_cast`. c) global namespace. d) `not_eq`. e) multiple inheritance. f) `virtual`.

**24.2**

a. False. It is legal to pass a non-const argument to a `const` function. However, when passing a `const` reference or pointer to a non-const function, the `const_cast` operator should be used to cast away the "const-ness" of the reference or pointer

b.

False. Programmers might inadvertently choose the namespace already in use.

c.

False. namespace bodies do not end in semicolons.

d.

False. namespaces can be nested.

e.

False. A mutable data member is always modifiable, even in a `const` member function.

 PREV

NEXT 

page footer



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1227]

- Variable `kilometers` is visible within namespace `Data`.
- Object `string1` is visible within namespace `Data`.
- Constant `POLAND` is not visible within namespace `Data`.
- Constant `GERMANY` is visible within namespace `Data`.
- Function `function` is visible to namespace `Data`.
- Namespace `Data` is visible to namespace `CountryInformation`.
- Object `map` is visible to namespace `CountryInformation`.
- Object `string1` is visible within namespace `RegionalInformation`.

## 24.6

Compare and contrast `mutable` and `const_cast`. Give at least one example of when one might be preferred over the other. [Note: This exercise does not require any code to be written.]

**24.7**

Write a program that uses `const_cast` to modify a `const` variable. [Hint: Use a pointer in your solution to point to the `const` identifier.]

**24.8**

What problem do virtual base classes solve?

**24.9**

Write a program that uses virtual base classes. The class at the top of the hierarchy should provide a constructor that takes at least one argument (i.e., do not provide a default constructor). What challenges does this present for the inheritance hierarchy?

**24.10**

Find the error(s) in each of the following. When possible, explain how to correct each error.

**a.**

```
namespace Name {
 int x;
 int y;
 mutable int z;
};
```

**b.**

```
int integer = const_cast< int >(double);
```

**c.**

```
namespace PCM(111, "hello"); // construct namespace
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1228]

## Appendix A. Operator Precedence and Associativity Chart

### Section A.1. Operator Precedence

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Pages 1229 - 1230]

Operator

Type

Associativity

::

binary scope resolution

left to right

::

unary scope resolution

( )

parentheses

left to right

[ ]

array subscript

.

member selection via object

->

member selection via pointer

`++`

unary postfix increment

`--`

unary postfix decrement

`typeid`

runtime type information

`dynamic_cast< type >`

runtime type-checked cast

`static_cast< type >`

compile-time type-checked cast

`reinterpret_cast< type >`

cast for nonstandard conversions

`const_cast< type >`

cast away `const`-ness

`++`

unary prefix increment

right to left

`--`

unary prefix decrement

`+`

unary plus

`-`

unary minus

`!`

unary logical negation

`~`

unary bitwise complement

`sizeof`

determine size in bytes

&

address

\*

dereference

`new`

dynamic memory allocation

`new[ ]`

dynamic array allocation

`delete`

dynamic memory deallocation

`delete[ ]`

dynamic array deallocation

( type )

C-style unary cast

right to left

. \*

pointer to member via object

left to right

- > \*

pointer to member via pointer

\*

multiplication

left to right

/

division

%

modulus

+

addition

left to right

-

subtraction

<<

bitwise left shift

left to right

>>

bitwise right shift

<

relational less than

left to right

<=

relational less than or equal to

>

relational greater than

>=

relational greater than or equal to

==

relational is equal to

left to right

!=

relational is not equal to

&

bitwise AND

left to right

^

bitwise exclusive OR

left to right

|

bitwise inclusive OR

left to right

&&

logical AND

left to right

||

logical OR

left to right

? :

ternary conditional

right to left

=

assignment

right to left

+ =

addition assignment

- =

subtraction assignment

\* =

multiplication assignment

/ =

division assignment

% =

modulus assignment

& =

bitwise AND assignment

^ =

bitwise exclusive OR assignment

| =

bitwise inclusive OR assignment

<<=

bitwise left-shift assignment

>>=

bitwise right-shift assignment

,

comma

left to right

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)
[NEXT ▶](#)

[Page 1231]

## Appendix B. ASCII Character Set

**Figure B.1. ASCII character set.**

ASCII character set										
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

The digits at the left of the table are the left digits of the decimal equivalent (0127) of the character code, and the digits at the top of the table are the right digits of the character code. For example, the character code for "F" is 70, and the character code for "&" is 38.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1233]

### **Figure C.1. C++ fundamental types.**

(This item is displayed on page 1232 in the print version)

Integral Types	Floating-Point Types
bool	float
char	double
signed char	long double
unsigned char	
short int	
unsigned short int	
int	
unsigned int	
long int	
unsigned long int	
wchar_t	

The exact sizes and ranges of values for the fundamental types are implementation dependent. The header files `<climits>` (for the integral types) and `<cfloat>` (for the floating-point types) specify the ranges of values supported on your system.

The range of values a type supports depends on the number of bytes that are used to represent that type. For example, consider a system with 4 byte (32 bits) ints. For the signed int type, the nonnegative values are in the range 0 to 2,147,483,647 ( $2^{31} - 1$ ). The negative values are in the range -2,147,483,648 ( $-2^{31}$ ). This is a total of  $2^{32}$  possible values. An unsigned int on the same system would use the same number of bits to represent data, but would not represent any negative values. This results

in values in the range 0 to 4,294,967,295 ( $2^{32} - 1$ ). On the same system, a `short int` could not use more than 32 bits to represent its data and a `long int` must use at least 32 bits.

C++ provides the data type `bool` for variables that can hold only the values `TRUE` and `false`.

 PREV

NEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1235]

## Outline

[D.1 Introduction](#)

[D.2 Abbreviating Binary Numbers as Octal and Hexadecimal Numbers](#)

[D.3 Converting Octal and Hexadecimal Numbers to Binary Numbers](#)

[D.4 Converting from Binary, Octal or Hexadecimal to Decimal](#)

[D.5 Converting from Decimal to Binary, Octal or Hexadecimal](#)

[D.6 Negative Binary Numbers: Two's Complement Notation](#)

## Summary

## Terminology

## Self-Review Exercises

## Answers to Self-Review Exercises

## Exercises



[page footer](#)

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1236]

**Figure D.3. Positional values in the decimal number system.**

**Positional values in the decimal number system**

Decimal digit	9	3	7
Position name	Hundreds	Tens	Ones
Positional value	100	10	1
Positional value as a power of the base (10)	$10^2$	$10^1$	$10^0$

For longer decimal numbers, the next positions to the left would be the thousands position (10 to the 3rd power), the ten-thousands position (10 to the 4th power), the hundred-thousands position (10 to the 5th power), the millions position (10 to the 6th power), the ten-millions position (10 to the 7th power) and so on.

[Page 1237]

In the binary number 101, the rightmost 1 is written in the ones position, the 0 is written in the twos position and the leftmost 1 is written in the fours position. Note that each position is a power of the base (base 2) and that these powers begin at 0 and increase by 1 as we move left in the number ([Fig. D.4](#)).

$$\text{So, } 101 = 2^2 + 2^0 = 4 + 1 = 5.$$

**Figure D.4. Positional values in the binary number system.**

**Positional values in the binary number system**

Binary digit	1	0	1
Position name	Fours	Twos	Ones

Positional value	4	2	1
Positional value as a power of the base (2)	$2^2$	$2^1$	$2^0$

For longer binary numbers, the next positions to the left would be the eights position (2 to the 3rd power), the sixteens position (2 to the 4th power), the thirty-twos position (2 to the 5th power), the sixty-fourths position (2 to the 6th power) and so on.

In the octal number 425, we say that the 5 is written in the ones position, the 2 is written in the eights position and the 4 is written in the sixty-fourths position. Note that each of these positions is a power of the base (base 8) and that these powers begin at 0 and increase by 1 as we move left in the number ([Fig. D.5](#)).

**Figure D.5. Positional values in the octal number system.**

<b>Positional values in the octal number system</b>			
Decimal digit	4	2	5
Position name	Sixty-fourths	Eights	Ones
Positional value	64	8	1
Positional value as a power of the base (8)	$8^2$	$8^1$	$8^0$

For longer octal numbers, the next positions to the left would be the five-hundred-and-twelves position (8 to the 3rd power), the four-thousand-and-ninety-sixes position (8 to the 4th power), the thirty-two-thousand-seven-hundred-and-sixty-eights position (8 to the 5th power) and so on.

In the hexadecimal number 3DA, we say that the A is written in the ones position, the D is written in the sixteens position and the 3 is written in the two-hundred-and-fifty-sixes position. Note that each of these positions is a power of the base (base 16) and that these powers begin at 0 and increase by 1 as we move left in the number ([Fig. D.6](#)).

**Figure D.6. Positional values in the hexadecimal number system.**

(This item is displayed on page 1238 in the print version)

## Positional values in the hexadecimal number system

Decimal digit	3	D	A
Position name	Two-hundred-and-fifty-sixes	Sixteens	Ones
Positional value	256	16	1
Positional value as a power of the base (16)	$16^2$	$16^1$	$16^0$

For longer hexadecimal numbers, the next positions to the left would be the four-thousand-and-ninety-sixes position (16 to the 3rd power), the sixty-five-thousand-five-hundred-and-thirty-sixes position (16 to the 4th power) and so on.

[PREV](#)
[NEXT](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1239]

Binary number    Octal equivalent    Hexadecimal equivalent

100011010001	4321	8D1
--------------	------	-----

To see how the binary number converts easily to octal, simply break the 12-digit binary number into groups of three consecutive bits each, starting from the right, and write those groups over the corresponding digits of the octal number as follows:

100	011	010	001
4	3	2	1

Note that the octal digit you have written under each group of three bits corresponds precisely to the octal equivalent of that 3-digit binary number, as shown in [Fig. D.7](#).

The same kind of relationship can be observed in converting from binary to hexadecimal. Break the 12-digit binary number into groups of four consecutive bits each, starting from the right, and write those groups over the corresponding digits of the hexadecimal number as follows:

1000	1101	0001
8	D	1

Notice that the hexadecimal digit you wrote under each group of four bits corresponds precisely to the hexadecimal equivalent of that 4-digit binary number as shown in [Fig. D.7](#).

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1239 (continued)]

## D.3. Converting Octal and Hexadecimal Numbers to Binary Numbers

In the previous section, we saw how to convert binary numbers to their octal and hexadecimal equivalents by forming groups of binary digits and simply rewriting them as their equivalent octal digit values or hexadecimal digit values. This process may be used in reverse to produce the binary equivalent of a given octal or hexadecimal number.

For example, the octal number 653 is converted to binary simply by writing the 6 as its 3-digit binary equivalent 110, the 5 as its 3-digit binary equivalent 101 and the 3 as its 3-digit binary equivalent 011 to form the 9-digit binary number 110101011.

The hexadecimal number FAD5 is converted to binary simply by writing the F as its 4-digit binary equivalent 1111, the A as its 4-digit binary equivalent 1010, the D as its 4-digit binary equivalent 1101 and the 5 as its 4-digit binary equivalent 0101 to form the 16-digit 1111101011010101.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1240]

To convert octal 7614 to decimal 3980, we use the same technique, this time using appropriate octal positional values, as shown in [Fig. D.9](#).

**Figure D.9. Converting an octal number to decimal.**

#### Converting an octal number to decimal

Positional values:	512	64	8	1
Symbol values:	7	6	1	4
Products	$7 * 512 = 3584$	$6 * 64 = 384$	$1 * 8 = 8$	$4 * 1 = 4$
Sum:	$= 3584 + 384 + 8 + 4 = 3980$			

To convert hexadecimal AD3B to decimal 44347, we use the same technique, this time using appropriate hexadecimal positional values, as shown in [Fig. D.10](#).

**Figure D.10. Converting a hexadecimal number to decimal.**

#### Converting a hexadecimal number to decimal

Positional values:	4096	256	16	1
Symbol values:	A	D	3	B
Products	$A * 4096 = 40960$	$D * 256 = 3328$	$3 * 16 = 48$	$B * 1 = 11$
Sum:	$= 40960 + 3328 + 48 + 11 = 44347$			



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1241]

Next we work from the leftmost column to the right. We divide 32 into 57 and observe that there is one 32 in 57 with a remainder of 25, so we write 1 in the 32 column. We divide 16 into 25 and observe that there is one 16 in 25 with a remainder of 9 and write 1 in the 16 column. We divide 8 into 9 and observe that there is one 8 in 9 with a remainder of 1. The next two columns each produce quotients of 0 when their positional values are divided into 1, so we write 0s in the 4 and 2 columns. Finally, 1 into 1 is 1, so we write 1 in the 1 column. This yields:

Positional values: 32 16 8 4 2 1

Symbol values: 1 1 1 0 0 1

and thus decimal 57 is equivalent to binary 111001.

To convert decimal 103 to octal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values: 512 64 8 1

Then we discard the column with positional value 512, yielding:

Positional values: 64 8 1

Next we work from the leftmost column to the right. We divide 64 into 103 and observe that there is one 64 in 103 with a remainder of 39, so we write 1 in the 64 column. We divide 8 into 39 and observe that there are four 8s in 39 with a remainder of 7 and write 4 in the 8 column. Finally, we divide 1 into 7 and observe that there are seven 1s in 7 with no remainder, so we write 7 in the 1 column. This yields:

Positional values: 64 8 1

Symbol values: 1 4 7

and thus decimal 103 is equivalent to octal 147.

To convert decimal 375 to hexadecimal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values: 4096 256 16 1

Then we discard the column with positional value 4096, yielding:

Positional values: 256 16 1

Next we work from the leftmost column to the right. We divide 256 into 375 and observe that there is one 256 in 375 with a remainder of 119, so we write 1 in the 256 column. We divide 16 into 119 and observe that there are seven 16s in 119 with a remainder of 7 and write 7 in the 16 column. Finally, we divide 1 into 7 and observe that there are seven 1s in 7 with no remainder, so we write 7 in the 1 column. This yields:

Positional values: 256 16 1

Symbol values: 1 7 7

and thus decimal 375 is equivalent to hexadecimal 177.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1243]

Suppose `a` is 27 and `value` is 13 as before. If the two's complement of `value` is actually the negative of `value`, then adding the two's complement of `value` to `a` should produce the result 14. Let us try this:

<code>a</code> (i.e., 27)	00000000 00000000 00000000 00011011
<code>+(~value + 1)</code>	+11111111 11111111 11111111 11110011
<hr/>	
	00000000 00000000 00000000 00001110

which is indeed equal to 14.

 PREV

NEXT 

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1243 (continued)]

## Summary

- An integer such as 19 or 227 or 63 in a C++ program is assumed to be in the decimal (base 10) number system. The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The lowest digit is 0 and the highest is one less than the base of 10.
- Internally, computers use the binary (base 2) number system. The binary number system has only two digits, namely 0 and 1. Its lowest digit is 0 and its highest is one less than the base of 2.
- The octal number system (base 8) and the hexadecimal number system (base 16) are popular primarily because they make it convenient to abbreviate binary numbers.
- The digits of the octal number system range from 0 to 7.
- The hexadecimal number system poses a problem because it requires 16 digits a lowest digit of 0 and a highest digit with a value equivalent to decimal 15 (one less than the base of 16). By convention, we use the letters A through F to represent the hexadecimal digits corresponding to decimal values 10 through 15.
- Each number system uses positional notation each position in which a digit is written has a different positional value.
- A particularly important relationship of both the octal and the hexadecimal number systems to the binary system is that their bases (8 and 16 respectively) are powers of the base of the binary number system (base 2).
- To convert an octal to a binary number, replace each octal digit with its three-digit binary equivalent.
- To convert a hexadecimal to a binary number, simply replace each hexadecimal digit with its four-digit binary equivalent.
- Because we are accustomed to working in decimal, it is convenient to convert a binary, octal or hexadecimal number to decimal to get a sense of the number's "real" worth.
- To convert a number to decimal from another base, multiply the decimal equivalent of each digit by its positional value and sum the products.
- Computers represent negative numbers using two's complement notation.
- To form the negative of a value in binary, first form its one's complement by applying C++'s bitwise complement operator (~). This reverses the bits of the value. To form the two's complement of a value, simply add one to the value's one's complement.

 PREV

NEXT 

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1244]

decimal number system

digit

hexadecimal number system

negative value

octal number system

one's complement notation

positional notation

positional value

symbol value

two's complement notation

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1244 (continued)]

## Self-Review Exercises

- D.1 The bases of the decimal, binary, octal and hexadecimal number systems are \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ and \_\_\_\_\_ respectively.
- D.2 In general, the decimal, octal and hexadecimal representations of a given binary number contain (more/fewer) digits than the binary number contains.
- D.3 (True/False) A popular reason for using the decimal number system is that it forms a convenient notation for abbreviating binary numbers simply by substituting one decimal digit per group of four binary bits.
- D.4 The (octal / hexadecimal / decimal) representation of a large binary value is the most concise (of the given alternatives).
- D.5 (True/False) The highest digit in any base is one more than the base.
- D.6 (True/False) The lowest digit in any base is one less than the base.
- D.7 The positional value of the rightmost digit of any number in either binary, octal, decimal or hexadecimal is always \_\_\_\_\_.
- D.8 The positional value of the digit to the left of the rightmost digit of any number in binary, octal, decimal or hexadecimal is always equal to \_\_\_\_\_.
- D.9 Fill in the missing values in this chart of positional values for the rightmost four positions in each of the indicated number systems:

decimal	1000	100	10	1
hexadecimal	...	256	...	...
binary	...	...	...	...
octal	512	...	8	...

- D.10** Convert binary 110101011000 to octal and to hexadecimal.
- D.11** Convert hexadecimal FACE to binary.
- D.12** Convert octal 7316 to binary.
- D.13** Convert hexadecimal 4FEC to octal. [Hint: First convert 4FEC to binary, then convert that binary number to octal.]
- D.14** Convert binary 1101110 to decimal.
- D.15** Convert octal 317 to decimal.
- D.16** Convert hexadecimal EFD4 to decimal.
- D.17** Convert decimal 177 to binary, to octal and to hexadecimal.
- D.18** Show the binary representation of decimal 417. Then show the one's complement of 417 and the two's complement of 417.
- D.19** What is the result when a number and its two's complement are added to each other?

 PREVNEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1246]

512 256 128 64 32 16 8 4 2 1 256 128 64 32 16 8 4 2 1  $(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)$   
 $+(0*4)+(0*2)+(1*1)$  110100001

One's complement: 001011110

Two's complement: 001011111

Check: Original binary number + its two's complement

110100001  
001011111  
-----  
000000000

## D.19

Zero.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1246 (continued)]

## Exercises

- D.20** Some people argue that many of our calculations would be easier in the base 12 number system because 12 is divisible by so many more numbers than 10 (for base 10). What is the lowest digit in base 12? What would be the highest symbol for the digit in base 12? What are the positional values of the rightmost four positions of any number in the base 12 number system?
- D.21** Complete the following chart of positional values for the rightmost four positions in each of the indicated number systems:

decimal	1000	100	10	1
base 6	...	...	6	...
base 13	...	169	...	...
base 3	27	...	...	...

- D.22** Convert binary 100101111010 to octal and to hexadecimal.
- D.23** Convert hexadecimal 3A7D to binary.
- D.24** Convert hexadecimal 765F to octal. [Hint: First convert 765F to binary, then convert that binary number to octal.]
- D.25** Convert binary 1011110 to decimal.
- D.26** Convert octal 426 to decimal.
- D.27** Convert hexadecimal FFFF to decimal.
- D.28** Convert decimal 299 to binary, to octal and to hexadecimal.

- D.29** Show the binary representation of decimal 779. Then show the one's complement of 779 and the two's complement of 779.
- D.30** Show the two's complement of integer value 1 on a machine with 32-bit integers.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1248]

## Outline

[E.1 Introduction](#)

[E.2 Redirecting Input/Output on UNIX/LINUX/Mac OS X and Windows Systems](#)

[E.3 Variable-Length Argument Lists](#)

[E.4 Using Command-Line Arguments](#)

[E.5 Notes on Compiling Multiple-Source-File Programs](#)

[E.6 Program Termination with `exit` and `atexit`](#)

[E.7 The `volatile` Type Qualifier](#)

[E.8 Suffixes for Integer and Floating-Point Constants](#)

[E.9 Signal Handling](#)

[E.10 Dynamic Memory Allocation with `calloc` and `realloc`](#)

[E.11 The Unconditional Branch: `goto`](#)

[E.12 Unions](#)

[E.13 Linkage Specifications](#)

[E.14 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

## Answers to Self-Review Exercises

### Exercises

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1248 (continued)]

## E.1. Introduction

This chapter presents several topics not ordinarily covered in introductory courses. Many of the capabilities discussed here are specific to particular operating systems, especially UNIX/LINUX/Mac OS X and/or Windows. Much of the material is for the benefit of C++ programmers who will need to work with older C legacy code.

[◀ PREV](#)[NEXT ▶](#)

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1249]

Note that \$ represents the UNIX command-line prompt. (UNIX prompts vary from system to system and between shells on a single system.) Redirection is an operating-system function, not another C++ feature.

The second method of redirecting input is **piping**. A **pipe (|)** causes the output of one program to be redirected as the input to another program. Suppose program `random` outputs a series of random integers; the output of `random` can be "piped" directly to program `sum` using the UNIX command line

```
$ random | sum
```

This causes the sum of the integers produced by `random` to be calculated. Piping can be performed in UNIX, LINUX, Mac OS X and Windows.

Program output can be redirected to a file by using the **redirect output symbol (>)**. (The same symbol is used for UNIX, LINUX, Mac OS X and Windows.) For example, to redirect the output of program `random` to a new file called `out`, use

```
$ random > out
```

Finally, program output can be appended to the end of an existing file by using the **append output symbol (>>)**. (The same symbol is used for UNIX, LINUX, Mac OS X and Windows.) For example, to append the output from program `random` to file `out` created in the preceding command line, use the command line

```
$ random >> out
```

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1251]

Variable-length argument lists promote variables of type `float` to type `double`. These argument lists also promote integral variables that are smaller than `int` to type `int` (variables of type `int`, `unsigned`, `long` and `unsigned long` are left alone).

### Software Engineering Observation E.1



Variable-length argument lists can be used only with fundamental type arguments and with arguments of C-style struct types that do not contain C++ specific features such as virtual functions, constructors, destructors, references, const data members and virtual base classes.

[Page 1252]

### Common Programming Error E.1



Placing an ellipsis in the middle of a function parameter list is a syntax error. An ellipsis may be placed only at the end of the parameter list.

[PREV](#)

[NEXT](#)

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1253]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1254]

prior to the variable's use in that file. In the preceding declaration, the storage class-specifier `extern` indicates to the compiler that variable `flag` is defined either later in the same file or in a different file. The compiler informs the linker that unresolved references to variable `flag` appear in the file. (The compiler does not know where `flag` is defined, so it lets the linker attempt to find `flag`.) If the linker cannot locate a definition of `flag`, a linker error is reported. If a proper global definition is located, the linker resolves the references by indicating where `flag` is located.

### Performance Tip E.1



Global variables increase performance because they can be accessed directly by any functionthe overhead of passing data to functions is eliminated.

### Software Engineering Observation E.2



Global variables should be avoided unless application performance is critical or the variable represents a shared global resource such as `cin`, because they violate the principle of least privilege, and they make software difficult to maintain.

Just as `extern` declarations can be used to declare global variables to other program files, function prototypes can be used to declare functions in other program files. (The `extern` specifier is not required in a function prototype.) This is accomplished by including the function prototype in each file in which the function is invoked, then compiling each source file and linking the resulting object code files together. Function prototypes indicate to the compiler that the specified function is defined either later in the same file or in a different file. The compiler does not attempt to resolve references to such a functionthat task is left to the linker. If the linker cannot locate a function definition, an error is generated.

As an example of using function prototypes to extend the scope of a function, consider any program containing the preprocessor directive `#include <cstring>`. This directive includes in a file the function prototypes for functions such as `strcmp` and `strcat`. Other functions in the file can use `strcmp` and `strcat` to accomplish their tasks. The `strcmp` and `strcat` functions are defined for us separately. We do not need to know where they are defined. We are simply reusing the code in our programs. The linker resolves our references to these functions. This process enables us to use the functions in the standard library.

## Software Engineering Observation E.3



Creating programs in multiple source files facilitates software reusability and good software engineering. Functions may be common to many applications. In such instances, those functions should be stored in their own source files, and each source file should have a corresponding header file containing function prototypes. This enables programmers of different applications to reuse the same code by including the proper header file and compiling their application with the corresponding source file.

### Portability Tip E.1



Some systems do not support global variable names or function names of more than six characters. This should be considered when writing programs that will be ported to multiple platforms.

[Page 1255]

It is possible to restrict the scope of a global variable or function to the file in which it is defined. The storage-class specifier `static`, when applied to a file scope variable or a function, prevents it from being used by any function that is not defined in the same file. This is referred to as **internal linkage**. Global variables (except those that are `const`) and functions that are not preceded by `static` in their definitions have **external linkage**; they can be accessed in other files if those files contain proper declarations and/or function prototypes.

#### The global variable declaration

```
static double pi = 3.14159;
```

creates variable `pi` of type `double`, initializes it to `3.14159` and indicates that `pi` is known only to functions in the file in which it is defined.

The `static` specifier is commonly used with utility functions that are called only by functions in a particular file. If a function is not required outside a particular file, the principle of least privilege should be enforced by using `static`. If a function is defined before it is used in a file, `static` should be applied to the function definition. Otherwise, `static` should be applied to the function prototype.

When building large programs from multiple source files, compiling the program becomes tedious if making small changes to one file means that the entire program must be recompiled. Many systems provide special utilities that recompile only source files dependant on the modified program file. On UNIX

systems, the utility is called **make**. Utility **make** reads a file called **Makefile** that contains instructions for compiling and linking the program. Systems such as Borland C++ and Microsoft Visual C++ for PCs provide **make** utilities and "projects." For more information on **make** utilities, see the manual for your particular system.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1256]

**Figure E.4. Using functions `exit` and `atexit`.**

(This item is displayed on pages 1256 - 1257 in the print version)

```
1 // Fig. E.4: figE_04.cpp
2 // Using the exit and atexit functions
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::cin;
7
8 #include <cstdlib>
9 using std::atexit;
10 using std::exit;
11
12 void print();
13
14 int main()
15 {
16 atexit(print); // register function print
17
18 cout << "Enter 1 to terminate program with function exit"
19 << "\nEnter 2 to terminate program normally\n";
20
21 int answer;
22 cin >> answer;
23
24 // exit if answer is 1
25 if (answer == 1)
26 {
27 cout << "\nTerminating program with function exit\n";
28 exit(EXIT_SUCCESS);
29 } // end if
30
31 cout << "\nTerminating program by reaching the end of main"
32 << endl;
33
34 return 0;
35 } // end main
36
```

```
37 // display message before termination
38 void print()
39 {
40 cout << "Executing function print at program termination\n"
41 << "Program terminated" << endl;
42 } // end function print
```

Enter 1 to terminate program with function exit  
Enter 2 to terminate program normally  
2

Terminating program by reaching the end of main  
Executing function print at program termination  
Program terminated

Enter 1 to terminate program with function exit  
Enter 2 to terminate program normally  
1

Terminating program with function exit  
Executing function print at program termination  
Program terminated

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1257]

## E.7. The volatile Type Qualifier

The **volatile** type qualifier is applied to a definition of a variable that may be altered from outside the program (i.e., the variable is not completely under the control of the program). Thus, the compiler cannot perform optimizations (such as speeding program execution or reducing memory consumption, for example) that depend on "knowing that a variable's behavior is influenced only by program activities the compiler can observe."

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1257 (continued)]

## E.8. Suffixes for Integer and Floating-Point Constants

C++ provides integer and floating-point suffixes for specifying the types of integer and floating-point constants. The integer suffixes are: `u` or `U` for an `unsigned` integer, `l` or `L` for a `long` integer, and `ul` or `UL` for an `unsigned long` integer. The following constants are of type `unsigned`, `long` and `unsigned long`, respectively:

`174u`  
`8358L`  
`28373ul`

If an integer constant is not suffixed, its type is `int`; if the constant cannot be stored in an `int`, it is stored in a `long`.

The floating-point suffixes are `f` or `F` for a `float` and `l` or `L` for a `long double`. The following constants are of type `long double` and `float`, respectively:

`3.14159L`  
`1.28f`

A floating-point constant that is not suffixed is of type `double`. A constant with an improper suffix results in either a compiler warning or an error.

[◀ PREV](#)[NEXT ▶](#)

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1258]

**Figure E.5. Signals defined in header <csignal>.**

Signal	Explanation
SIGABRT	Abnormal termination of the program (such as a call to abort).
SIGFPE	An erroneous arithmetic operation, such as a divide by zero or an operation resulting in overflow.
SIGILL	Detection of an illegal instruction.
SIGINT	Receipt of an interactive attention signal.
SIGSEGV	An invalid access to storage.
SIGTERM	A termination request sent to the program.

[Figure E.6](#) traps an interactive signal (SIGINT) with function `signal`. The program calls `signal` with SIGINT and a pointer to function `signalHandler`. (Remember that the name of a function is a pointer to the function.) Now, when a signal of type SIGINT occurs, function `signalHandler` is called, a message is printed and the user is given the option to continue normal execution of the program. If the user wishes to continue execution, the signal handler is reinitialized by calling `signal` again (some systems require the signal handler to be reinitialized), and control returns to the point in the program at which the signal was detected. In this program, function `raise` is used to simulate an interactive signal. A random number between 1 and 50 is chosen. If the number is 25, then `raise` is called to generate the signal. Normally, interactive signals are initiated outside the program. For example, pressing Ctrl+C during program execution on a UNIX, LINUX, Mac OS X or Windows system generates an interactive signal that terminates program execution. Signal handling can be used to trap the interactive signal and prevent the program from terminating.

**Figure E.6. Using signal handling.**

(This item is displayed on pages 1258 - 1260 in the print version)

```
1 // Fig. E.6: figE_06.cpp
2 // Using signal handling
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <csignal>
12 using std::raise;
13 using std::signal;
14
15 #include <cstdlib>
16 using std::exit;
17 using std::rand;
18 using std::srand;
19
20 #include <ctime>
21 using std::time;
22
23 void signalHandler(int);
24
25 int main()
26 {
27 signal(SIGINT, signalHandler);
28 srand(time(0));
29
30 // create and output random numbers
31 for (int i = 1; i <= 100; i++)
32 {
33 int x = 1 + rand() % 50;
34
35 if (x == 25)
36 raise(SIGINT); // raise SIGINT when x is 25
37
38 cout << setw(4) << i;
39
40 if (i % 10 == 0)
41 cout << endl; // output endl when i is a multiple of 10
42 } // end for
43
44 return 0;
45 } // end main
46
47 // handles signal
48 void signalHandler(int signalValue)
```

```

49 {
50 cout << "\nInterrupt signal (" << signalValue
51 << ") received.\n"
52 << "Do you wish to continue (1 = yes or 2 = no)? ";
53
54 int response;
55
56 cin >> response;
57
58 // check for invalid responses
59 while (response != 1 && response != 2)
60 {
61 cout << "(1 = yes or 2 = no)? ";
62 cin >> response;
63 } // end while
64
65 // determine if it is time to exit
66 if (response != 1)
67 exit(EXIT_SUCCESS);
68
69 // call signal and pass it SIGINT and address of signalHandler
70 signal(SIGINT, signalHandler);
71 } // end function signalHandler

```

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	

Interrupt signal (2) received.

Do you wish to continue (1 = yes or 2 = no)? 1

100

1 2 3 4

Interrupt signal (2) received.

Do you wish to continue (1 = yes or 2 = no)? 2

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1261]

```
void *realloc(void *ptr, size_t size);
```

Function `realloc` takes two arguments a pointer to the original object (`ptr`) and the new size of the object (`size`). If `ptr` is 0, `realloc` works identically to `malloc`. If `size` is 0 and `ptr` is not 0, the memory for the object is freed. Otherwise, if `ptr` is not 0 and `size` is greater than zero, `realloc` tries to allocate a new block of memory. If the new space cannot be allocated, the object pointed to by `ptr` is unchanged. Function `realloc` returns either a pointer to the reallocated memory or a null pointer.

## Common Programming Error E.2



Using the `delete` operator on a pointer resulting from `malloc`, `calloc` and `realloc`. Using `realloc` or `free` on a pointer resulting from the `new` operator.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1262]

### Figure E.7. Using goto.

(This item is displayed on pages 1261 - 1262 in the print version)

```
1 // Fig. E.7: figE_07.cpp
2 // Using goto.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::left;
9 using std::setw;
10
11 int main()
12 {
13 int count = 1;
14
15 start: // label
16 // goto end when count exceeds 10
17 if (count > 10)
18 goto end;
19
20 cout << setw(2) << left << count;
21 ++count;
22
23 // goto start on line 17
24 goto start;
25
26 end: // label
27 cout << endl;
28
29 return 0;
30 } // end main
```

1 2 3 4 5 6 7 8 9 10

In Chapters 45, we stated that only three control structures are required to write any program sequence, selection and repetition. When the rules of structured programming are followed, it is possible to create deeply nested control structures from which it is difficult to escape efficiently. Some programmers use `goto` statements in such situations as a quick exit from a deeply nested structure. This eliminates the need to test multiple conditions to escape from a control structure.

### Performance Tip E.2



The `goto` statement can be used to exit deeply nested control structures efficiently but can make code more difficult to read and maintain.

### Error-Prevention Tip E.1



The `goto` statement should be used only in performance-oriented applications. The `goto` statement is unstructured and can lead to programs that are more difficult to debug, maintain and modify.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1263]

## Portability Tip E.2



If data are stored in a `union` as one type and referenced as another type, the results are implementation dependent.

At different times during a program's execution, some objects might not be relevant, while one other object is so a `union` shares the space instead of wasting storage on objects that are not being used. The number of bytes used to store a `union` will be at least enough to hold the largest member.

## Performance Tip E.3



Using `unions` conserves storage.

## Portability Tip E.3



The amount of storage required to store a `union` is implementation dependent.

A `union` is declared in the same format as a `struct` or a `class`. For example,

```
union Number
{
 int x;
 double y;
};
```

indicates that `Number` is a `union` type with members `int x` and `double y`. The `union` definition must precede all functions in which it will be used.

## Software Engineering Observation E.4



Like a `struct` or a `class` declaration, a `union` declaration simply creates a new type. Placing a `union` or `struct` declaration outside any function does not create a global variable.

The only valid built-in operations that can be performed on a `union` are assigning a `union` to another `union` of the same type, taking the address (`&`) of a `union` and accessing `union` members using the structure member operator (`.`) and the structure pointer operator (`->`). `unions` cannot be compared.

## Common Programming Error E.4



Comparing `unions` is a compilation error, because the compiler does not know which member of each is active and hence which member of one to compare to which member of the other.

A `union` is similar to a `class` in that it can have a constructor to initialize any of its members. A `union` that has no constructor can be initialized with another `union` of the same type, with an expression of the type of the first member of the `union` or with an initializer (enclosed in braces) of the type of the first member of the `union`. `unions` can have other member functions, such as destructors, but a `union`'s member functions cannot be declared `virtual`. The members of a `union` are `public` by default.

## Common Programming Error E.5



Initializing a `union` in a declaration with a value or an expression whose type is different from the type of the `union`'s first member is a compilation error.

---

[Page 1264]

A `union` cannot be used as a base class in inheritance (i.e., classes cannot be derived from `unions`). `unions` can have objects as members only if these objects do not have a constructor, a destructor or an overloaded assignment operator. None of a `union`'s data members can be declared `static`.

[Figure E.8](#) uses the variable `value` of type `union Number` to display the value stored in the `union` as both an `int` and a `double`. The program output is implementation dependent. The program output shows that the internal representation of a `double` value can be quite different from the representation

of an int.

### Figure E.8. Printing the value of a union in both member data types.

(This item is displayed on pages 1264 - 1265 in the print version)

```
1 // Fig. E.8: figE_08.cpp
2 // An example of a union.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // define union Number
8 union Number
9 {
10 int integer1;
11 double double1;
12 } // end union Number
13
14 int main()
15 {
16 Number value; // union variable
17
18 value.integer1 = 100; // assign 100 to member integer1
19
20 cout << "Put a value in the integer member\n"
21 << "and print both members.\nint: "
22 << value.integer1 << "\ndouble: " << value.double1
23 << endl;
24
25 value.double1 = 100.0; // assign 100.0 to member double1
26
27 cout << "Put a value in the floating member\n"
28 << "and print both members.\nint: "
29 << value.integer1 << "\ndouble: " << value.double1
30 << endl;
31
32 return 0;
33 } // end main
```

```

Put a value in the integer member
and print both members.
int: 100
double: -9.25596e+061
Put a value in the floating member
and print both members.
int: 0
double: 100

```

An **anonymous union** is a union without a type name that does not attempt to define objects or pointers before its terminating semicolon. Such a union does not create a type but does create an unnamed object. An anonymous union's members may be accessed directly in the scope in which the anonymous union is declared just as are any other local variables there is no need to use the dot (.) or arrow (->) operators.

---

[Page 1265]

Anonymous unions have some restrictions. Anonymous unions can contain only data members. All members of an anonymous union must be public. And an anonymous union declared globally (i.e., at file scope) must be explicitly declared static. [Figure E.9](#) illustrates the use of an anonymous union.

**Figure E.9. Using an anonymous union.**

(This item is displayed on pages 1265 - 1266 in the print version)

```

1 // Fig. E.9: figE_09.cpp
2 // Using an anonymous union.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 // declare an anonymous union
10 // members integer1, double1 and charPtr share the same space
11 union
12 {
13 int integer1;
14 double double1;

```

```
15 char *charPtr;
16 } // end anonymous union
17
18 // declare local variables
19 int integer2 = 1;
20 double double2 = 3.3;
21 char *char2Ptr = "Anonymous";
22
23 // assign value to each union member
24 // successively and print each
25 cout << integer2 << ' ';
26 integer1 = 2;
27 cout << integer1 << endl;
28
29 cout << double2 << ' ';
30 double1 = 4.4;
31 cout << double1 << endl;
32
33 cout << char2Ptr << ' ';
34 charPtr = "union";
35 cout << charPtr << endl;
36
37 return 0;
38 } // end main
```

```
1 2
3.3 4.4
Anonymous union
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1266]

To inform the compiler that one or several functions have been compiled in C, write the function prototypes as follows:

```
extern "C" function prototype // single function

extern "C" // multiple functions
{
 function prototypes
}
```

These declarations inform the compiler that the specified functions are not compiled in C++, so name encoding should not be performed on the functions listed in the linkage specification. These functions can then be linked properly with the program. C++ environments normally include the standard C libraries and do not require the programmer to use linkage specifications for those functions.

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1266 (continued)]

## E.14. Wrap-Up

This appendix introduced a number of C legacy code topics. We discussed redirecting keyboard input to come from a file and redirecting screen output to a file. We also introduced variable-length argument lists, command-line arguments and processing of unexpected events. You also learned about allocating and resizing memory dynamically. In the next appendix, you will learn about using the preprocessor to include other files, define symbolic constants and define macros.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1268]

- Function `atexit` registers a function in a program to be called upon normal termination of the program i.e., either when the program terminates by reaching the end of `main`, or when `exit` is invoked.
- Function `atexit` takes a pointer to a function (e.g., a function name) as an argument. Functions called at program termination cannot have arguments and cannot return a value.
- Function `exit` takes one argument normally the symbolic constant `EXIT_SUCCESS` or the symbolic constant `EXIT_FAILURE`. If `exit` is called with `EXIT_SUCCESS`, the implementation-defined value for successful termination is returned to the calling environment. If `exit` is called with `EXIT_FAILURE`, the implementation-defined value for unsuccessful termination is returned.
- When function `exit` is invoked, any functions registered with `atexit` are invoked in the reverse order of their registration, all streams associated with the program are flushed and closed and control returns to the host environment.
- The `volatile` qualifier is used to prevent optimizations of a variable, because it can be modified from outside the program's scope.
- C++ provides integer and floating-point suffixes for specifying the types of integer and floating-point constants. The integer suffixes are `u` or `U` for an `unsigned` integer, `l` or `L` for a `long` integer and `ul` or `UL` for an `unsigned long` integer. If an integer constant is not suffixed, its type is determined by the first type capable of storing a value of that size (first `int`, then `long int`). The floating-point suffixes are `f` or `F` for a `float` and `l` or `L` for a `long double`. A floating-point constant that is not suffixed is of type `double`.
- The signal-handling library provides the capability to register a function to trap unexpected events with function `signal`. Function `signal` receives two arguments an integer signal number and a pointer to the signal-handling function.
- Signals can also be generated with function `raise` and an integer argument.
- The general-utilities library (`cstdlib`) provides functions `calloc` and `realloc` for dynamic memory allocation. These functions can be used to create dynamic arrays.
- Function `calloc` receives two arguments the number of elements (`nmemb`) and the size of each element (`size`) and initializes the elements of the array to zero. The function returns a pointer to the allocated memory or if the memory is not allocated, the function returns a null pointer.
- Function `realloc` changes the size of an object allocated by a previous call to `malloc`, `calloc` or `realloc`. The original object's contents are not modified, provided that the amount of memory allocated is larger than the amount allocated previously.
- Function `realloc` takes two arguments a pointer to the original object (`ptr`) and the new size of the object (`size`). If `ptr` is null, `realloc` works identically to `malloc`. If `size` is 0 and the pointer received is not null, the memory for the object is freed. Otherwise, if `ptr` is not null and `size` is greater than zero, `realloc` tries to allocate a new block of memory for the object. If the new space cannot be allocated, the object pointed to by `ptr` is unchanged. Function `realloc` returns either a pointer to the reallocated memory or a null pointer.
- The result of the `goto` statement is a change in the program's flow of control. Program execution continues at the first statement after the label in the `goto` statement.
- A label is an identifier followed by a colon. A label must appear in the same function as the `goto` statement that refers to it.

- A union is a data type whose members share the same storage space. The members can be almost any type. The storage reserved for a union is large enough to store its largest member. In most cases, unions contain two or more data types. Only one member, and thus one data type, can be referenced at a time.
- A union is declared in the same format as a structure.

---

[Page 1269]

- A union can be initialized only with a value of the type of its first member or another object of the same union type.
- C++ enables the programmer to provide linkage specifications to inform the compiler that a function was compiled on a C compiler and to prevent the name of the function from being encoded by the C++ compiler.
- To inform the compiler that one or several functions have been compiled in C, write the function prototypes as follows:

```
extern "C" function prototype // single function

extern "C" // multiple functions
{
 function prototypes
}
```

These declarations inform the compiler that the specified functions are not compiled in C++, so name encoding should not be performed on the functions listed in the linkage specification. These functions can then be linked properly with the program.

- C++ environments normally include the standard C libraries and do not require the programmer to use linkage specifications for those functions.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1269 (continued)]

## Terminology

append output symbol >>

argv

atexit

calloc

command-line arguments

const

<csignal>

<cstdarg>

dynamic arrays

event

exit

EXIT\_FAILURE

EXIT\_SUCCESS

extern "C"

extern storage-class specifier

external linkage

float suffix (f or F)

floating-point exception

free

goto statement

I/O redirection

illegal instruction

internal linkage

interrupt

long double suffix (l or L)

long integer suffix (l or L)

make

Makefile

malloc

pipe |

piping

raise

realloc

redirect input symbol <

redirect output symbol >

segmentation violation

`signal`

signal-handling library

`static` storage-class specifier

`trap`

`union`

`unsigned` integer suffix (`u` or `U`)

`unsigned long` integer suffix (`ul` or `UL`)

`va_arg`

`va_end`

`va_list`

`va_start`

variable-length argument list

`volatile`

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1270]

•

The \_\_\_\_\_ symbol is used to append the output of a program to the end of a file.

•

A(n) \_\_\_\_\_ is used to direct the output of a program as the input of another program.

•

A(n) \_\_\_\_\_ in the parameter list of a function indicates that the function can receive a variable number of arguments.

•

Macro \_\_\_\_\_ must be invoked before the arguments in a variable-length argument list can be accessed.

•

Macro \_\_\_\_\_ is used to access the individual arguments of a variable-length argument list.

•

Macro \_\_\_\_\_ performs termination housekeeping in a function whose variable argument list was referred to by macro `va_start`.

•

Argument \_\_\_\_\_ of `main` receives the number of arguments in a command line.

•

Argument \_\_\_\_\_ of `main` stores command-line arguments as character strings.

•

The UNIX utility \_\_\_\_\_ reads a file called \_\_\_\_\_ that contains instructions for compiling

and linking a program consisting of multiple source files. The utility recompiles a file only if the file (or a header it uses) has been modified since it was last compiled.

• Function \_\_\_\_\_ forces a program to terminate execution.

• Function \_\_\_\_\_ registers a function to be called upon normal termination of the program.

• An integer or floating-point \_\_\_\_\_ can be appended to an integer or floating-point constant to specify the exact type of the constant.

• Function \_\_\_\_\_ can be used to register a function to trap unexpected events.

• Function \_\_\_\_\_ generates a signal from within a program.

• Function \_\_\_\_\_ dynamically allocates memory for an array and initializes the elements to zero.

• Function \_\_\_\_\_ changes the size of a block of dynamically allocated memory.

• A(n) \_\_\_\_\_ is an entity containing a collection of variables that occupy the same memory, but at different times.

• The \_\_\_\_\_ keyword is used to introduce a union definition.

 PREV

NEXT 

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1270 (continued)]

## Answers to Self-Review Exercises

- E.1** a) redirect input (<). b) redirect output (>). c) append output (>>). d) pipe (|). e) ellipsis (...). f) va\_start. g) va\_arg. h) va\_end. i) argc. j) argv. k) make, Makefile. l) exit. m) atexit. n) suffix. o) signal. p) raise. q) calloc. r) realloc. s) union. t) union.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1271]

### E.6

Write a program that dynamically allocates an array of integers using a function from <cstdlib>, not the new operator. The size of the array should be input from the keyboard. The elements of the array should be assigned values input from the keyboard. Print the values of the array. Next, reallocate the memory for the array to half of the current number of elements. Print the values remaining in the array to confirm that they match the first half of the values in the original array.

### E.7

Write a program that takes two file names as command-line arguments, reads the characters from the first file one at a time and writes the characters in reverse order to the second file.

### E.8

Write a program that uses goto statements to simulate a nested looping structure that prints a square of asterisks as shown in Fig. E.10. The program should use only the following three output statements:

```
cout << '*';
cout << ' ';
cout << endl;
```

### E.9

Provide the definition for union Data containing char character1, short short1, long long1, float float1 and double double1.

### E.10

Create union Integer with members char character1, short short1, int integer1 and long long1. Write a program that inputs values of type char, short, int and long and stores the values in union variables of type union Integer. Each union variable should be printed as a char, a short, an int and a long. Do the values always print correctly?

### E.11

Create union FloatingPoint with members float float1, double double1 and long double longDouble. Write a program that inputs values of type float, double and long double and stores

the values in union variables of type union FloatingPoint. Each union variable should be printed as a float, a double and a long double. Do the values always print correctly?

## E.12

Given the union

```
union A
{
 double y;
 char *zPtr;
};
```

which of the following are correct statements for initializing the union?

a.

```
A p = b; // b is of type A
```

b.

```
A q = x; // x is a double
```

c.

```
A r = 3.14159;
```

d.

```
A s = { 79.63 };
```

e.

```
A t = { "Hi There!" };
```

f.

```
A u = { 3.14159, "Pi" };
```

g.

```
A v = { Y = 7.843 , zPtr = &x };
```

## Figure E.10. Example for Exercise E.8.

```
* * * * *
* *
* *
* *
* * * * *
```

[!\[\]\(447dc77693e0f38f88876637c0fa6504\_img.jpg\) PREV](#)

[\*\*NEXT\*\* !\[\]\(91c420a1de3cd67b171aa1a664969a5a\_img.jpg\)](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1273]

## Outline

[F.1 Introduction](#)

[F.2 The #include Preprocessor Directive](#)

[F.3 The #define Preprocessor Directive: Symbolic Constants](#)

[F.4 The #define Preprocessor Directive: Macros](#)

[F.5 Conditional Compilation](#)

[F.6 The #error and #pragma Preprocessor Directives](#)

[F.7 The # and ## Operators](#)

[F.8 Predefined Symbolic Constants](#)

[F.9 Assertions](#)

[F.10 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

[Exercises](#)



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 1273 (continued)]

## F.1. Introduction

This chapter introduces the **preprocessor**. Preprocessing occurs before a program is compiled. Some possible actions are inclusion of other files in the file being compiled, definition of **symbolic constants** and **macros**, **conditional compilation** of program code and **conditional execution of preprocessor directives**. All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;). Preprocessor directives are processed fully before compilation begins.

### Common Programming Error F.1



Placing a semicolon at the end of a preprocessor directive can lead to a variety of errors, depending on the type of preprocessor directive.

### Software Engineering Observation F.1



Many preprocessor features (especially macros) are more appropriate for C programmers than for C++ programmers. C++ programmers should familiarize themselves with the preprocessor, because they might need to work with C legacy code.

[◀ PREV](#)

[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1274]

The #include directive is used to include standard header files such as <iostream> and <iomanip>. The #include directive is also used with programs consisting of several source files that are to be compiled together. A header file containing declarations and definitions common to the separate program files is often created and included in the file. Examples of such declarations and definitions are classes, structures, unions, enumerations and function prototypes, constants and stream objects (e.g., cin).

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 1274 (continued)]

## F.3. The #define Preprocessor Directive: Symbolic Constants

The `#define` preprocessor directive creates symbolic constants represented as symbols and macros operations defined as symbols. The `#define` preprocessor directive format is

`#define identifier replacement-text`

When this line appears in a file, all subsequent occurrences (except those inside a string) of identifier in that file will be replaced by replacement-text before the program is compiled. For example,

```
#define PI 3.14159
```

replaces all subsequent occurrences of the symbolic constant `PI` with the numeric constant `3.14159`. Symbolic constants enable the programmer to create a name for a constant and use the name throughout the program. Later, if the constant needs to be modified throughout the program, it can be modified once in the `#define` preprocessor directive and when the program is recompiled, all occurrences of the constant in the program will be modified. [Note: Everything to the right of the symbolic constant name replaces the symbolic constant. For example, `#define PI = 3.14159` causes the preprocessor to replace every occurrence of `PI` with `= 3.14159`. Such replacement is the cause of many subtle logic and syntax errors.] Redefining a symbolic constant with a new value without first undefining it is also an error. Note that `const` variables in C++ are preferred over symbolic constants. Constant variables have a specific data type and are visible by name to a debugger. Once a symbolic constant is replaced with its replacement text, only the replacement text is visible to a debugger. A disadvantage of `const` variables is that they might require a memory location of their data type size symbolic constants do not require any additional memory.

### Common Programming Error F.2



Using symbolic constants in a file other than the file in which the symbolic constants are defined is a compilation error (unless they are `#included` from a header file).

### Good Programming Practice F.1



Using meaningful names for symbolic constants helps make programs more self-documenting.

[◀ PREV](#)

page footer

[NEXT ▶](#)

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

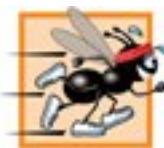
[Page 1276]

Macro CIRCLE\_AREA could be defined as a function. Function circleArea, as in

```
double circleArea(double x) { return 3.14159 * x * x; }
```

performs the same calculation as CIRCLE\_AREA, but the overhead of a function call is associated with function circleArea. The advantages of CIRCLE\_AREA are that macros insert code directly in the program avoiding function overhead and the program remains readable because CIRCLE\_AREA is defined separately and named meaningfully. A disadvantage is that its argument is evaluated twice. Also, every time a macro appears in a program, the macro is expanded. If the macro is large, this produces an increase in program size. Thus, there is a trade-off between execution speed and program size (if disk space is low). Note that `inline` functions (see [Chapter 3](#)) are preferred to obtain the performance of macros and the software engineering benefits of functions.

### Performance Tip F.1



Macros can sometimes be used to replace a function call with `inline` code prior to execution time. This eliminates the overhead of a function call. `Inline` functions are preferable to macros because they offer the type-checking services of functions.

The following is a macro definition with two arguments for the area of a rectangle:

```
#define RECTANGLE_AREA(x, y) ((x) * (y))
```

Wherever `RECTANGLE_AREA( a, b )` appears in the program, the values of `a` and `b` are substituted in the macro replacement text, and the macro is expanded in place of the macro name. For example, the statement

```
rectArea = RECTANGLE_AREA(a + 4, b + 7);
```

is expanded to

```
rectArea = ((a + 4) * (b + 7));
```

The value of the expression is evaluated and assigned to variable rectArea.

The replacement text for a macro or symbolic constant is normally any text on the line after the identifier in the #define directive. If the replacement text for a macro or symbolic constant is longer than the remainder of the line, a backslash (\) must be placed at the end of each line of the macro (except the last line), indicating that the replacement text continues on the next line.

Symbolic constants and macros can be discarded using the **#undef preprocessor directive**. Directive #undef "undefines" a symbolic constant or macro name. The scope of a symbolic constant or macro is from its definition until it is either undefined with #undef or the end of the file is reached. Once undefined, a name can be redefined with #define.

Note that expressions with side effects (e.g., variable values are modified) should not be passed to a macro, because macro arguments may be evaluated more than once.

#### Common Programming Error F.4



Macros often end up replacing a name that wasn't intended to be a use of the macro but just happened to be spelled the same. This can lead to exceptionally mysterious compilation and syntax errors.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1278]

## Common Programming Error F.5



Inserting conditionally compiled output statements for debugging purposes in locations where C++ currently expects a single statement can lead to syntax errors and logic errors. In this case, the conditionally compiled statement should be enclosed in a compound statement. Thus, when the program is compiled with debugging statements, the flow of control of the program is not altered.

[PREV](#)

page footer

[NEXT](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1278 (continued)]

## F.6. The #error and #pragma Preprocessor Directives

### The #error directive

#### #error tokens

prints an implementation-dependent message including the tokens specified in the directive. The tokens are sequences of characters separated by spaces. For example,

```
#error 1 - Out of range error
```

contains six tokens. In one popular C++ compiler, for example, when a #error directive is processed, the tokens in the directive are displayed as an error message, preprocessing stops and the program does not compile.

### The #pragma directive

#### #pragma tokens

causes an implementation-defined action. A pragma not recognized by the implementation is ignored. A particular C++ compiler, for example, might recognize pragmas that enable the programmer to take advantage of that compiler's specific capabilities. For more information on #error and #pragma, see the documentation for your C++ implementation.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1279]

```
#define TOKENCONCAT(x, y) x ## y
```

When TOKENCONCAT appears in the program, its arguments are concatenated and used to replace the macro. For example, TOKENCONCAT( o, k ) is replaced by ok in the program. The ## operator must have two operands.

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)
[NEXT ▶](#)

[Page 1279 (continued)]

## F.8. Predefined Symbolic Constants

There are six **predefined symbolic constants** (Fig. F.1). The identifiers for each of these begin and (and, except for `__cplusplus`, end) with two underscores. These identifiers and the defined preprocessor operator (Section F.5) cannot be used in `#define` or `#undef` directives.

**Figure F.1. The predefined symbolic constants.**

Symbolic constant	Description
<code>__LINE__</code>	The line number of the current source code line (an integer constant).
<code>__FILE__</code>	The presumed name of the source file (a string).
<code>__DATE__</code>	The date the source file is compiled (a string of the form " <code>Mmm dd yyyy</code> " such as " <code>Aug 19 2002</code> ").
<code>__STDC__</code>	Indicates whether the program conforms to the ANSI/ISO C standard. Contains value 1 if there is full conformance and is undefined otherwise.
<code>__TIME__</code>	The time the source file is compiled (a string literal of the form " <code>hh:mm:ss</code> ").
<code>__cplusplus</code>	Contains the value <code>199711L</code> (the date the ISO C++ standard was approved) if the file is being compiled by a C++ compiler, undefined otherwise. Allows a file to be set up to be compiled as either C or C++.

[◀ PREV](#)
[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1280]

in the program file rather than deleting each assertion manually. As with the `DEBUG` symbolic constant, `NDEBUG` is often set by compiler command-line options or through a setting in the IDE.

Most C++ compilers now include exception handling. C++ programmers prefer using exceptions rather than assertions. But assertions are still valuable for C++ programmers who work with C legacy code.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1280 (continued)]

## F.10. Wrap-Up

This appendix discussed the `#include` directive, which is used to develop larger programs. You also learned about the `#define` directive, which is used to create macros. We introduced conditional compilation, displaying error messages and using assertions. In the next appendix, you will implement the design of the ATM system from the "software engineering case study" found in [Chapters 17, 9 and 13](#).

[◀ PREV](#)[NEXT ▶](#)

**page footer**

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1281]

- The # operator causes the following replacement text token to be converted to a string surrounded by quotes. The # operator must be used in a macro with arguments, because the operand of # must be an argument of the macro.
- The ## operator concatenates two tokens. The ## operator must have two operands.
- There are six predefined symbolic constants. Constant \_\_LINE\_\_ is the line number of the current source code line (an integer). Constant \_\_FILE\_\_ is the presumed name of the file (a string). Constant \_\_DATE\_\_ is the date the source file is compiled (a string). Constant \_\_TIME\_\_ is the time the source file is compiled (a string). Note that each of the predefined symbolic constants begins (and, with the exception of \_\_cplusplus, ends) with two underscores.
- The assert macrodefined in the <cassert> header filetests the value of an expression. If the value of the expression is 0 (false), then assert prints an error message and calls function abort to terminate program execution.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1281 (continued)]

## Terminology

\ (backslash) continuation character

abort

argument

assert

<cassert>

concatenation preprocessor operator ##

conditional compilation

conditional execution of preprocessor

convert-to-string preprocessor directive

\_\_cplusplus

<cstdio>

<cstdlib>

\_\_DATE\_\_

debugger

#define

directives

#elif

#else

#endif

#error

expand a macro

\_\_FILE\_\_

header file

#if

#ifdef

#ifndef

#include "filename"

#include <filename>

\_\_LINE\_\_

macro

macro with arguments

operator #

#pragma

predefined symbolic constants

preprocessing directive

preprocessor

replacement text

scope of a symbolic constant or macro

standard library header files

symbolic constant

TIME

#undef

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1282]

- \_\_\_\_\_ enables the programmer to control the execution of preprocessor directives and the compilation of program code.

- 
- The \_\_\_\_\_ macro prints a message and terminates program execution if the value of the expression the macro evaluates is 0.

- 
- The \_\_\_\_\_ directive inserts a file in another file.

- 
- The \_\_\_\_\_ operator concatenates its two arguments.

- 
- The \_\_\_\_\_ operator converts its operand to a string.

- 
- The character \_\_\_\_\_ indicates that the replacement text for a symbolic constant or macro continues on the next line.

## F.2

Write a program to print the values of the predefined symbolic constants `_LINE_`, `_FILE_`, `_DATE_` and `_TIME_` listed in Fig. F.1.

## F.3

Write a preprocessor directive to accomplish each of the following:

a.

Define symbolic constant YES to have the value 1.

b.

Define symbolic constant NO to have the value 0.

c.

Include the header file common.h. The header is found in the same directory as the file being compiled.

d.

If symbolic constant TRUE is defined, undefine it, and redefine it as 1. Do not use #ifdef.

e.

If symbolic constant trUE is defined, undefine it, and redefine it as 1. Use the #ifdef preprocessor directive.

f.

If symbolic constant ACTIVE is not equal to 0, define symbolic constant INACTIVE as 0. Otherwise, define INACTIVE as 1.

g.

Define macro CUBE\_VOLUME that computes the volume of a cube (takes one argument).

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1283]

```
__LINE__ = 9
__FILE__ = c:\cpp4e\ch19\ex19_02.CPP
__DATE__ = Jul 17 2002
__TIME__ = 09:55:58
__cplusplus = 199711L
```

### F.3

a.

```
#define YES 1
```

b.

```
#define NO 0
```

c.

```
#include "common.h"
```

d.

```
#if defined(TRUE)
 #undef TRUE
 #define TRUE 1
#endif
```

e.

```
#ifdef TRUE
 #undef TRUE
 #define TRUE 1
#endif
```

```
#if ACTIVE
 #define INACTIVE 0
#else
 #define INACTIVE 1
#endif
```

g.

```
#define CUBE_VOLUME(x) ((x) * (x) * (x))
```

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1284]

## F.10

Write a program that uses macro SUMARRAY to sum the values in a numeric array. The macro should receive the array and the number of elements in the array as arguments.

## F.11

Rewrite the solutions to [Exercise F.4](#) to [Exercise F.10](#) as inline functions.

## F.12

For each of the following macros, identify the possible problems (if any) when the preprocessor expands the macros:

a.

```
#define SQR(x) x * x
```

b.

```
#define SQR(x) (x * x)
```

c.

```
#define SQR(x) (x) * (x)
```

d.

```
#define SQR(x) ((x) * (x))
```

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1285]

## Appendix G. ATM Case Study Code

[Section G.1. ATM Case Study Implementation](#)

[Section G.2. Class ATM](#)

[Section G.3. Class Screen](#)

[Section G.4. Class Keypad](#)

[Section G.5. Class CashDispenser](#)

[Section G.6. Class DepositSlot](#)

[Section G.7. Class Account](#)

[Section G.8. Class BankDatabase](#)

[Section G.9. Class Transaction](#)

[Section G.10. Class BalanceInquiry](#)

[Section G.11. Class Withdrawal](#)

[Section G.12. Class Deposit](#)

[Section G.13. Test Program ATMCaseStudy.cpp](#)

[Section G.14. Wrap-Up](#)

 PREV

NEXT 



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1286]

We conclude the discussion by presenting a C++ program (`ATMCaseStudy.cpp`) that starts the ATM and puts the other classes in the system in use. Recall that we are developing a first version of the ATM system that runs on a personal computer and uses the computer's keyboard and monitor to approximate the ATM's keypad and screen. We also only simulate the actions of the ATM's cash dispenser and deposit slot. We attempt to implement the system, however, so that real hardware versions of these devices could be integrated without significant changes in the code.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1287]

**Figure G.1. Definition of class ATM, which represents the ATM.**

(This item is displayed on page 1286 in the print version)

```
1 // ATM.h
2 // ATM class definition. Represents an automated teller machine.
3 #ifndef ATM_H
4 #define ATM_H
5
6 #include "Screen.h" // Screen class definition
7 #include "Keypad.h" // Keypad class definition
8 #include "CashDispenser.h" // CashDispenser class definition
9 #include "DepositSlot.h" // DepositSlot class definition
10 #include "BankDatabase.h" // BankDatabase class definition
11 class Transaction; // forward declaration of class Transaction
12
13 class ATM
14 {
15 public:
16 ATM(); // constructor initializes data members
17 void run(); // start the ATM
18 private:
19 bool userAuthenticated; // whether user is authenticated
20 int currentAccountNumber; // current user's account number
21 Screen screen; // ATM's screen
22 Keypad keypad; // ATM's keypad
23 CashDispenser cashDispenser; // ATM's cash dispenser
24 DepositSlot depositSlot; // ATM's deposit slot
25 BankDatabase bankDatabase; // account information database
26
27 // private utility functions
28 void authenticateUser(); // attempts to authenticate user
29 void performTransactions(); // performs transactions
30 int displayMainMenu() const; // displays main menu
31
32 // return object of specified Transaction derived class
33 Transaction *createTransaction(int);
34 }; // end class ATM
35
36 #endif // ATM_H
```

Lines 1925 of Fig. G.1 implement the class's attributes as private data members. We determine all but one of these attributes from the UML class diagrams of Fig. 13.28 and Fig. 13.29. Note that we implement the UML Boolean attribute `userAuthenticated` in Fig. 13.29 as a `bool` data member in C++ (line 19). Line 20 declares a data member not found in our UML designan `int` data member `currentAccountNumber` that keeps track of the account number of the current authenticated user. We will soon see how the class uses this data member.

Lines 2124 create objects to represent the parts of the ATM. Recall from the class diagram of Fig. 13.28 that class `ATM` has composition relationships with classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`, so class `ATM` is responsible for their creation. Line 25 creates a `BankDatabase`, with which the `ATM` interacts to access and manipulate bank account information. [Note: If this were a real ATM system, the `ATM` class would receive a reference to an existing database object created by the bank. However, in this implementation we are only simulating the bank's database, so class `ATM` creates the `BankDatabase` object with which it interacts.] Note that lines 610 `#include` the class definitions of `Screen`, `Keypad`, `CashDispenser`, `DepositSlot` and `BankDatabase` so that the `ATM` can store objects of these classes.

Lines 2830 and 33 contain function prototypes for private utility functions that the class uses to perform its tasks. We will see how these functions serve the class shortly. Note that member function `createTransaction` (line 33) returns a `transaction` pointer. To include the class name `transaction` in this file, we must at least include a forward declaration of class `transaction` (line 11). Recall that a forward declaration tells the compiler that a class exists, but that the class is defined elsewhere. A forward declaration is sufficient here, as we are using `transaction` only as a return typeif we were creating an actual `TTransaction` object, we would need to `#include` the full `transaction` header file.

## ATM Class Member-Function Definitions

Figure G.2 contains the member-function definitions for class `ATM`. Lines 37 `#include` the header files required by the implementation file `ATM.cpp`. Note that including the `ATM` header file allows the compiler to ensure that the class's member functions are defined correctly. This also allows the member functions to use the class's data members.

Line 10 declares an `enum` named `MenuOption` that contains constants corresponding to the four options in the ATM's main menu (i.e., balance inquiry, withdrawal, deposit and exit). Note that setting `BALANCE_INQUIRY` to 1 causes the subsequent enumeration constants to be assigned the values 2, 3 and 4, as enumeration constant values increment by 1.

**Figure G.2. ATM class member-function definitions.**

(This item is displayed on pages 1287 - 1290 in the print version)

```

1 // ATM.cpp
2 // Member-function definitions for class ATM.
3 #include "ATM.h" // ATM class definition
4 #include "Transaction.h" // Transaction class definition
5 #include "BalanceInquiry.h" // BalanceInquiry class definition
6 #include "Withdrawal.h" // Withdrawal class definition
7 #include "Deposit.h" // Deposit class definition
8
9 // enumeration constants represent main menu options
10 enum MenuOption { BALANCE_INQUIRY = 1, WITHDRAWAL, DEPOSIT, EXIT } ;
11
12 // ATM default constructor initializes data members
13 ATM::ATM()
14 : userAuthenticated(false), // user is not authenticated to start
15 currentAccountNumber(0) // no current account number to start
16 {
17 // empty body
18 } // end ATM default constructor
19
20 // start ATM
21 void ATM::run()
22 {
23 // welcome and authenticate user; perform transactions
24 while (true)
25 {
26 // loop while user is not yet authenticated
27 while (!userAuthenticated)
28 {
29 screen.displayMessageLine("\nWelcome! ");
30 authenticateUser(); // authenticate user
31 } // end while
32
33 performTransactions(); // user is now authenticated
34 userAuthenticated = false; // reset before next ATM session
35 currentAccountNumber = 0; // reset before next ATM session
36 screen.displayMessageLine("\nThank you! Goodbye! ");
37 } // end while
38 } // end function run
39
40 // attempt to authenticate user against database
41 void ATM::authenticateUser()
42 {
43 screen.displayMessage("\nPlease enter your account number: ");

```

```

44 int accountNumber = keypad.getInput(); // input account number
45 screen.displayMessage("\nEnter your PIN: "); // prompt for PIN
46 int pin = keypad.getInput(); // input PIN
47
48 // set userAuthenticated to bool value returned by database
49 userAuthenticated =
50 bankDatabase.authenticateUser(accountNumber, pin);
51
52 // check whether authentication succeeded
53 if (userAuthenticated)
54 {
55 currentAccountNumber = accountNumber; // save user's account #
56 } // end if
57 else
58 screen.displayMessageLine(
59 "Invalid account number or PIN. Please try again.");
60 } // end function authenticateUser
61
62 // display the main menu and perform transactions
63 void ATM::performTransactions()
64 {
65 // local pointer to store transaction currently being processed
66 Transaction *currentTransactionPtr;
67
68 bool userExited = false; // user has not chosen to exit
69
70 // loop while user has not chosen option to exit system
71 while (!userExited)
72 {
73 // show main menu and get user selection
74 int mainMenuSelection = displayMainMenu();
75
76 // decide how to proceed based on user's menu selection
77 switch (mainMenuSelection)
78 {
79 // user chose to perform one of three transaction types
80 case BALANCE_INQUIRY:
81 case WITHDRAWAL:
82 case DEPOSIT:
83 // initialize as new object of chosen type
84 currentTransactionPtr =
85 createTransaction(mainMenuSelection);
86
87 currentTransactionPtr->execute(); // execute transaction
88
89 // free the space for the dynamically allocated Transaction
90 delete currentTransactionPtr;
91

```

```

92 break;
93 case EXIT: // user chose to terminate session
94 screen.displayMessageLine("\nExiting the system...");
95 userExited = true; // this ATM session should end
96 break;
97 default: // user did not enter an integer from 1-4
98 screen.displayMessageLine(
99 "\nYou did not enter a valid selection. Try again.");
100 break;
101 } // end switch
102 } // end while
103 } // end function performTransactions
104
105 // display the main menu and return an input selection
106 int ATM::displayMainMenu() const
107 {
108 screen.displayMessageLine("\nMain menu:");
109 screen.displayMessageLine("1 - View my balance");
110 screen.displayMessageLine("2 - Withdraw cash");
111 screen.displayMessageLine("3 - Deposit funds");
112 screen.displayMessageLine("4 - Exit\n");
113 screen.displayMessage("Enter a choice: ");
114 return keypad.getInput(); // return user's selection
115 } // end function displayMainMenu
116
117 // return object of specified Transaction derived class
118 Transaction *ATM::createTransaction(int type)
119 {
120 Transaction *tempPtr; // temporary Transaction pointer
121
122 // determine which type of Transaction to create
123 switch (type)
124 {
125 case BALANCE_INQUIRY: // create new BalanceInquiry transaction
126 tempPtr = new BalanceInquiry(
127 currentAccountNumber, screen, bankDatabase);
128 break;
129 case WITHDRAWAL: // create new Withdrawal transaction
130 tempPtr = new Withdrawal(currentAccountNumber, screen,
131 bankDatabase, keypad, cashDispenser);
132 break;
133 case DEPOSIT: // create new Deposit transaction
134 tempPtr = new Deposit(currentAccountNumber, screen,
135 bankDatabase, keypad, depositSlot);
136 break;
137 } // end switch
138
139 return tempPtr; // return the newly created object

```

```
140 } // end function createTransaction
```

Lines 1318 define class ATM's constructor, which initializes the class's data members. When an ATM object is first created, no user is authenticated, so line 14 uses a member initializer to set userAuthenticated to false. Likewise, line 15 initializes currentAccountNumber to 0 because there is no current user yet.

ATM member function run (lines 2138) uses an infinite loop (lines 2437) to repeatedly welcome a user, attempt to authenticate the user and, if authentication succeeds, allow the user to perform transactions. After an authenticated user performs the desired transactions and chooses to exit, the ATM resets itself, displays a goodbye message to the user and restarts the process. We use an infinite loop here to simulate the fact that an ATM appears to run continuously until the bank turns it off (an action beyond the user's control). An ATM user has the option to exit the system, but does not have the ability to turn off the ATM completely.

Inside member function run's infinite loop, lines 2731 cause the ATM to repeatedly welcome and attempt to authenticate the user as long as the user has not been authenticated (i.e., !userAuthenticated is true). Line 29 invokes member function displayMessageLine of the ATM's screen to display a welcome message. Like Screen member function displayMessage designed in the case study, member function displayMessageLine (declared in line 13 of Fig. G.3 and defined in lines 2023 of Fig. G.4) displays a message to the user, but this member function also outputs a newline after displaying the message. We have added this member function during implementation to give class Screen's clients more control over the placement of displayed messages. Line 30 of Fig. G.2 invokes class ATM's private utility function authenticateUser (lines 4160) to attempt to authenticate the user.

---

[Page 1291]

We refer to the requirements document to determine the steps necessary to authenticate the user before allowing transactions to occur. Line 43 of member function authenticateUser invokes member function displayMessage of the ATM's screen to prompt the user to enter an account number. Line 44 invokes member function getInput of the ATM's keypad to obtain the user's input, then stores the integer value entered by the user in a local variable accountNumber. Member function authenticateUser next prompts the user to enter a PIN (line 45), and stores the PIN input by the user in a local variable pin (line 46). Next, lines 4950 attempt to authenticate the user by passing the accountNumber and pin entered by the user to the bankDatabase's authenticateUser member function. Class ATM sets its userAuthenticated data member to the bool value returned by this function. userAuthenticated becomes true if authentication succeeds (i.e., accountNumber and pin match those of an existing Account in bankDatabase) and remains false otherwise. If userAuthenticated is true, line 55 saves the account number entered by the user (i.e., accountNumber) in the ATM data member currentAccountNumber. The other member functions of class ATM use this variable whenever an ATM session requires access to the user's account number. If userAuthenticated is false, lines 5859 use the screen's displayMessageLine member function to indicate that an invalid account number and/or PIN was entered and the user must try again. Note that

we set `currentAccountNumber` only after authenticating the user's account number and the associated PIN if the database could not authenticate the user, `currentAccountNumber` remains 0.

After member function `run` attempts to authenticate the user (line 30), if `userAuthenticated` is still `false`, the `while` loop in lines 2731 executes again. If `userAuthenticated` is now `TRUE`, the loop terminates and control continues with line 33, which calls class ATM's utility function `performTransactions`.

Member function `performTransactions` (lines 63103) carries out an ATM session for an authenticated user. Line 66 declares a local `TTransaction` pointer, which we aim at a `BalanceInquiry`, `Withdrawal` or `Deposit` object representing the ATM transaction currently being processed. Note that we use a `TTransaction` pointer here to allow us to take advantage of polymorphism. Also note that we use the role name included in the class diagram of Fig. 3.20 `currentTransaction` naming this pointer. As per our pointer-naming convention, we append "`Ptr`" to the role name to form the variable name `currentTransactionPtr`. Line 68 declares another local variable a `bool` called `userExited` that keeps track of whether the user has chosen to exit. This variable controls a `while` loop (lines 71102) that allows the user to execute an unlimited number of transactions before choosing to exit. Within this loop, line 74 displays the main menu and obtains the user's menu selection by calling an ATM utility function `displayMainMenu` (defined in lines 106115). This member function displays the main menu by invoking member functions of the ATM's screen and returns a menu selection obtained from the user through the ATM's keypad. Note that this member function is `const` because it does not modify the contents of the object. Line 74 stores the user's selection returned by `displayMainMenu` in local variable `mainMenuSelection`.

[Page 1292]

After obtaining a main menu selection, member function `performTransactions` uses a `switch` statement (lines 77101) to respond to the selection appropriately. If `mainMenuSelection` is equal to any of the three enumeration constants representing transaction types (i.e., if the user chose to perform a transaction), lines 8485 call utility function `createTransaction` (defined in lines 118140) to return a pointer to a newly instantiated object of the type that corresponds to the selected transaction. Pointer `currentTransactionPtr` is assigned the pointer returned by `createTransaction`. Line 87 then uses `currentTransactionPtr` to invoke the new object's `execute` member function to execute the transaction. We will discuss `TTransaction` member function `execute` and the three `TTransaction` derived classes shortly. Finally, when the `transaction` derived class object is no longer needed, line 90 releases the memory dynamically allocated for it.

Note that we aim the `transaction` pointer `currentTransactionPtr` at an object of one of the three `TTransaction` derived classes so that we can execute transactions polymorphically. For example, if the user chooses to perform a balance inquiry, `mainMenuSelection` equals `BALANCE_INQUIRY`, leading `createTransaction` to return a pointer to a `BalanceInquiry` object. Thus, `currentTransactionPtr` points to a `BalanceInquiry`, and invoking `currentTransactionPtr->execute()` results in `BalanceInquiry`'s version of `execute` being called.

Member function `createTransaction` (lines 118–140) uses a `switch` statement (lines 123–137) to instantiate a new transaction derived class object of the type indicated by the parameter `type`. Recall that member function `performTransactions` passes `mainMenuSelection` to this member function only when `mainMenuSelection` contains a value corresponding to one of the three transaction types. Therefore `type` equals either `BALANCE_INQUIRY`, `WITHDRAWAL` or `DEPOSIT`. Each case in the `switch` statement aims the temporary pointer `tempPtr` at a newly created object of the appropriate `TRansaction` derived class. Note that each constructor has a unique parameter list, based on the specific data required to initialize the derived class object. A `BalanceInquiry` requires only the account number of the current user and references to the ATM's screen and the `bankDatabase`. In addition to these parameters, a `Withdrawal` requires references to the ATM's keypad and `cashDispenser`, and a `Deposit` requires references to the ATM's keypad and `depositSlot`. Note that, as you will soon see, the `BalanceInquiry`, `Withdrawal` and `Deposit` constructors each specify reference parameters to receive the objects representing the required parts of the ATM. Thus, when member function `createTransaction` passes objects in the ATM (e.g., `screen` and `keypad`) to the initializer for each newly created `TRansaction` derived class object, the new object actually receives references to the ATM's composite objects. We discuss the transaction classes in more detail in [Section G.9](#)[Section G.12](#).

After executing a transaction (line 87 in `performTransactions`), `userExited` remains `false` and the `while` loop in lines 71–102 repeats, returning the user to the main menu. However, if a user does not perform a transaction and instead selects the main menu option to exit, line 95 sets `userExited` to `TRUE`, causing the condition of the `while` loop (`!userExited`) to become `false`. This `while` is the final statement of member function `performTransactions`, so control returns to the calling function `run`. If the user enters an invalid main menu selection (i.e., not an integer from 14), lines 98–99 display an appropriate error message, `userExited` remains `false` and the user returns to the main menu to try again.

When `performTransactions` returns control to member function `run`, the user has chosen to exit the system, so lines 34–35 reset the ATM's data members `userAuthenticated` and `currentAccountNumber` to prepare for the next ATM user. Line 36 displays a goodbye message before the ATM starts over and welcomes the next user.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 1293]

## G.3. Class Screen

Class Screen (Figs. G.3G.4) represents the screen of the ATM and encapsulates all aspects of displaying output to the user. Class Screen approximates a real ATM's screen with a computer monitor and outputs text messages using cout and the stream insertion operator (<<). In this case study, we designed class Screen to have one operation `displayMessage`. For greater flexibility in displaying messages to the Screen, we now declare three Screen member functions `displayMessage`, `displayMessageLine` and `displayDollarAmount`. The prototypes for these member functions appear in lines 1214 of Fig. G.3.

**Figure G.3. Screen class definition.**

```

1 // Screen.h
2 // Screen class definition. Represents the screen of the ATM.
3 #ifndef SCREEN_H
4 #define SCREEN_H
5
6 #include <string>
7 using std::string;
8
9 class Screen
10 {
11 public:
12 void displayMessage(string) const; // output a message
13 void displayMessageLine(string) const; // output message with newline
14 void displayDollarAmount(double) const; // output a dollar amount
15 }; // end class Screen
16
17 #endif // SCREEN_H

```

**Figure G.4. Screen class member-function definitions.**

(This item is displayed on pages 1293 - 1294 in the print version)

```

1 // Screen.cpp
2 // Member-function definitions for class Screen.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include "Screen.h" // Screen class definition
12
13 // output a message without a newline
14 void Screen::displayMessage(string message) const
15 {
16 cout << message;
17 } // end function displayMessage
18
19 // output a message with a newline
20 void Screen::displayMessageLine(string message) const
21 {
22 cout << message << endl;
23 } // end function displayMessageLine
24
25 // output a dollar amount
26 void Screen::displayDollarAmount(double amount) const
27 {
28 cout << fixed << setprecision(2) << "$" << amount;
29 } // end function displayDollarAmount

```

## Screen Class Member-Function Definitions

Figure G.4 contains the member-function definitions for class Screen. Line 11 `#includes` the Screen class definition. Member function `displayMessage` (lines 1417) takes a `string` as an argument and prints it to the console using `cout` and the stream insertion operator (`<<`). The cursor stays on the same line, making this member function appropriate for displaying prompts to the user. Member function `displayMessageLine` (lines 2023) also prints a `string`, but outputs a newline to move the cursor to the next line. Finally, member function `displayDollarAmount` (lines 2629) outputs a properly formatted dollar amount (e.g., `$123.45`). Line 28 uses stream manipulators `fixed` and `setprecision` to output a value formatted with two decimal places. See Chapter 15, Stream Input/Output, for more information about formatting output.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1295]

## Keypad Class Member-Function Definition

In the Keypad implementation file ([Fig. G.6](#)), member function `getInput` (defined in lines 914) uses the standard input stream `cin` and the stream extraction operator (`>>`) to obtain input from the user. Line 11 declares a local variable to store the user's input. Line 12 reads input into local variable `input`, then line 13 returns this value. Recall that `getInput` obtains all the input used by the ATM. Keypad's `getInput` member function simply returns the integer input by the user. If a client of class `Keypad` requires input that satisfies some particular criteria (i.e., a number corresponding to a valid menu option), the client must perform the appropriate error checking. [Note: Using the standard input stream `cin` and the stream extraction operator (`>>`) allows noninteger input to be read from the user. Because the real ATM's keypad permits only integer input, however, we assume that the user enters an integer and do not attempt to fix problems caused by noninteger input.]

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1297]

Member function `isSufficientCashAvailable` (lines 2028) has a parameter `amount` that specifies the amount of cash in question. Lines 2427 return `TRUE` if the `CashDispenser`'s count is greater than or equal to `billsRequired` (i.e., enough bills are available) and `false` otherwise (i.e., not enough bills). For example, if a user wishes to withdraw \$80 (i.e., `billsRequired` is 4), but only three bills remain (i.e., `count` is 3), the member function returns `false`.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1298]

Recall from the requirements document that the ATM allows the user up to two minutes to insert an envelope. The current version of member function `isEnvelopeReceived` simply returns `true` immediately (line 9 of [Fig. G.10](#)), because this is only a software simulation, and we assume that the user has inserted an envelope within the required time frame. If an actual hardware deposit slot were connected to our system, member function `isEnvelopeReceived` might be implemented to wait for a maximum of two minutes to receive a signal from the hardware deposit slot indicating that the user has indeed inserted a deposit envelope. If `isEnvelopeReceived` were to receive such a signal within two minutes, the member function would return `TRue`. If two minutes elapsed and the member function still had not received a signal, then the member function would return `false`.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1299]

Member function `validatePIN` (lines 1723) determines whether a user-specified PIN (i.e., parameter `userPIN`) matches the PIN associated with the account (i.e., data member `pin`). Recall that we modeled this member function's parameter `userPIN` in the UML class diagram of [Fig. 6.37](#). If the two PINs match, the member function returns `TRue` (line 20); otherwise, it returns `false` (line 22).

Member functions `getAvailableBalance` (lines 2629) and `getTotalBalance` (lines 3235) are get functions that return the values of double data members `availableBalance` and `totalBalance`, respectively.

[Page 1300]

Member function `credit` (lines 3841) adds an amount of money (i.e., parameter `amount`) to an `Account` as part of a deposit transaction. Note that this member function adds the amount only to data member `totalBalance` (line 40). The money credited to an account during a deposit does not become available immediately, so we modify only the total balance. We assume that the bank updates the available balance appropriately at a later time. Our implementation of class `Account` includes only member functions required for carrying out ATM transactions. Therefore, we omit the member functions that some other bank system would invoke to add to data member `availableBalance` (to confirm a deposit) or subtract from data member `totalBalance` (to reject a deposit).

Member function `debit` (lines 4448) subtracts an amount of money (i.e., parameter `amount`) from an `Account` as part of a withdrawal transaction. This member function subtracts the amount from both data member `availableBalance` (line 46) and data member `totalBalance` (line 47), because a withdrawal affects both measures of an account balance.

Member function `getAccountNumber` (lines 5154) provides access to an `Account`'s `accountNumber`. We include this member function in our implementation so that a client of the class (i.e., `BankDatabase`) can identify a particular `Account`. For example, `BankDatabase` contains many `Account` objects, and it can invoke this member function on each of its `Account` objects to locate the one with a specific account number.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1301]

**Figure G.13. BankDatabase class definition.**

```
1 // BankDatabase.h
2 // BankDatabase class definition. Represents the bank's database.
3 #ifndef BANK_DATABASE_H
4 #define BANK_DATABASE_H
5
6 #include <vector> // class uses vector to store Account objects
7 using std::vector;
8
9 #include "Account.h" // Account class definition
10
11 class BankDatabase
12 {
13 public:
14 BankDatabase(); // constructor initializes accounts
15
16 // determine whether account number and PIN match those of an Account
17 bool authenticateUser(int, int); // returns true if Account authentic
18
19 double getAvailableBalance(int); // get an available balance
20 double getTotalBalance(int); // get an Account's total balance
21 void credit(int, double); // add amount to Account balance
22 void debit(int, double); // subtract amount from Account balance
23 private:
24 vector< Account > accounts; // vector of the bank's Accounts
25
26 // private utility function
27 Account * getAccount(int); // get pointer to Account object
28 }; // end class BankDatabase
29
30 #endif // BANK_DATABASE_H
```

**Figure G.14. BankDatabase class member-function definitions.**

(This item is displayed on pages 1301 - 1303 in the print version)

```

1 // BankDatabase.cpp
2 // Member-function definitions for class BankDatabase.
3 #include "BankDatabase.h" // BankDatabase class definition
4
5 // BankDatabase default constructor initializes accounts
6 BankDatabase::BankDatabase()
7 {
8 // create two Account objects for testing
9 Account account1(12345, 54321, 1000.0, 1200.0);
10 Account account2(98765, 56789, 200.0, 200.0);
11
12 // add the Account objects to the vector accounts
13 accounts.push_back(account1); // add account1 to end of vector
14 accounts.push_back(account2); // add account2 to end of vector
15 } // end BankDatabase default constructor
16
17 // retrieve Account object containing specified account number
18 Account * BankDatabase::getAccount(int accountNumber)
19 {
20 // loop through accounts searching for matching account number
21 for (size_t i = 0; i < accounts.size(); i++)
22 {
23 // return current account if match found
24 if (accounts[i].getAccountNumber() == accountNumber)
25 return &accounts[i];
26 } // end for
27
28 return NULL; // if no matching account was found, return NULL
29 } // end function getAccount
30
31 // determine whether user-specified account number and PIN match
32 // those of an account in the database
33 bool BankDatabase::authenticateUser(int userAccountNumber,
34 int userPIN)
35 {
36 // attempt to retrieve the account with the account number
37 Account * const userAccountPtr = getAccount(userAccountNumber);
38
39 // if account exists, return result of Account function validatePIN
40 if (userAccountPtr != NULL)
41 return userAccountPtr->validatePIN(userPIN);
42 else
43 return false; // account number not found, so return false
44 } // end function authenticateUser
45
46 // return available balance of Account with specified account number
47 double BankDatabase::getAvailableBalance(int userAccountNumber)
48 {
49 Account * const userAccountPtr = getAccount(userAccountNumber);

```

```

50 return userAccountPtr->getAvailableBalance();
51 } // end function getAvailableBalance
52
53 // return total balance of Account with specified account number
54 double BankDatabase::getTotalBalance(int userAccountNumber)
55 {
56 Account * const userAccountPtr = getAccount(userAccountNumber);
57 return userAccountPtr->getTotalBalance();
58 } // end function getTotalBalance
59
60 // credit an amount to Account with specified account number
61 void BankDatabase::credit(int userAccountNumber, double amount)
62 {
63 Account * const userAccountPtr = getAccount(userAccountNumber);
64 userAccountPtr->credit(amount);
65 } // end function credit
66
67 // debit an amount from of Account with specified account number
68 void BankDatabase::debit(int userAccountNumber, double amount)
69 {
70 Account * const userAccountPtr = getAccount(userAccountNumber);
71 userAccountPtr->debit(amount);
72 } // end function debit

```

## BankDatabase Class Member-Function Definitions

Figure G.14 contains the member-function definitions for class `BankDatabase`. We implement the class with a default constructor (lines 615) that adds `Account` objects to data member `accounts`. For the sake of testing the system, we create two new `Account` objects with test data (lines 910), then add them to the end of the `vector` (lines 1314). Note that the `Account` constructor has four parameters—the account number, the PIN assigned to the account, the initial available balance and the initial total balance.

---

[Page 1303]

Recall that class `BankDatabase` serves as an intermediary between class `ATM` and the actual `Account` objects that contain users' account information. Thus, the member functions of class `BankDatabase` do nothing more than invoke the corresponding member functions of the `Account` object belonging to the current `ATM` user.

We include private utility function `getAccount` (lines 1829) to allow the `BankDatabase` to obtain a pointer to a particular `Account` within `vector accounts`. To locate the user's `Account`, the `BankDatabase` compares the value returned by member function `getAccountNumber` for each element of `accounts` to a specified account number until it finds a match. Lines 2126 traverse the `accounts`

vector. If the account number of the current Account (i.e., `accounts[ i ]`) equals the value of parameter `accountNumber`, the member function immediately returns the address of the current Account (i.e., a pointer to the current Account). If no account has the given account number, then line 28 returns `NULL`. Note that this member function must return a pointer, as opposed to a reference, because there is the possibility that the return value could be `NULL`; a reference cannot be `NULL`, but a pointer can.

Note that `vector` function `size` (invoked in the loop-continuation condition in line 21) returns the number of elements in a `vector` as a value of type `size_t` (which is usually `unsigned int`). As a result, we declare the control variable `i` to be of type `size_t`, too. On some compilers, declaring `i` as an `int` would cause the compiler to issue a warning message, because the loop-continuation condition would compare a signed value (i.e., an `int`) and an `unsigned` value (i.e., a value of type `size_t`).

Member function `authenticateUser` (lines 3344) proves or disproves the identity of an ATM user. This member function takes a user-specified account number and user-specified PIN as arguments and indicates whether they match the account number and PIN of an Account in the database. Line 37 calls utility function `getAccount`, which returns either a pointer to an Account with `userAccountNumber` as its account number or `NULL` to indicate that `userAccountNumber` is invalid. We declare `userAccountPtr` to be a `const` pointer because, once the member function aims this pointer at the user's Account, the pointer should not change. If `getAccount` returns a pointer to an Account object, line 41 returns the `bool` value returned by that object's `validatePIN` member function. Note that `BankDatabase`'s `authenticateUser` member function does not perform the PIN comparison itself rather, it forwards `userPIN` to the Account object's `validatePIN` member function to do so. The value returned by Account member function `validatePIN` indicates whether the user-specified PIN matches the PIN of the user's Account, so member function `authenticateUser` simply returns this value to the client of the class (i.e., ATM).

[Page 1304]

`BankDatabase` trusts the ATM to invoke member function `authenticateUser` and receive a return value of `true` before allowing the user to perform transactions. `BankDatabase` also trusts that each transaction object created by the ATM contains the valid account number of the current authenticated user and that this is the account number passed to the remaining `BankDatabase` member functions as argument `userAccountNumber`. Member functions `getAvailableBalance` (lines 4751), `getTotalBalance` (lines 5458), `credit` (lines 6165) and `debit` (lines 6872) therefore simply retrieve a pointer to the user's Account object with utility function `getAccount`, then use this pointer to invoke the appropriate Account member function on the user's Account object. We know that the calls to `getAccount` within these member functions will never return `NULL`, because `userAccountNumber` must refer to an existing Account. Note that `getAvailableBalance` and `getTotalBalance` return the values returned by the corresponding Account member functions. Also note that `credit` and `debit` simply redirect parameter `amount` to the Account member functions they invoke.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1305]

**Figure G.16. TTransaction class member-function definitions.**

(This item is displayed on pages 1305 - 1306 in the print version)

```
1 // Transaction.cpp
2 // Member-function definitions for class Transaction.
3 #include "Transaction.h" // Transaction class definition
4 #include "Screen.h" // Screen class definition
5 #include "BankDatabase.h" // BankDatabase class definition
6
7 // constructor initializes common features of all Transactions
8 Transaction::Transaction(int userAccountNumber, Screen &atmScreen,
9 BankDatabase &atmBankDatabase)
10 : accountNumber(userAccountNumber),
11 screen(atmScreen),
12 bankDatabase(atmBankDatabase)
13 {
14 // empty body
15 } // end Transaction constructor
16
17 // return account number
18 int Transaction::getAccountNumber() const
19 {
20 return accountNumber;
21 } // end function getAccountNumber
22
23 // return reference to screen
24 Screen &Transaction::getScreen() const
25 {
26 return screen;
27 } // end function getScreen
28
29 // return reference to bank database
30 BankDatabase &Transaction::getBankDatabase() const
31 {
32 return bankDatabase;
33 } // end function getBankDatabase
```

Class transaction has a constructor (declared in line 13 of Fig. G.15 and defined in lines 815 of Fig. G.16) that takes the current user's account number and references to the ATM's screen and the bank's database as arguments. Because transaction is an abstract class, this constructor will never be called directly to instantiate transaction objects. Instead, the constructors of the transaction derived classes will use base-class initializer syntax to invoke this constructor.

Class transaction has three public get functions `getAccountNumber` (declared in line 17 of Fig. G.15 and defined in lines 1821 of Fig. G.16), `getScreen` (declared in line 18 of Fig. G.15 and defined in lines 2427 of Fig. G.16) and `getBankDatabase` (declared in line 19 of Fig. G.15 and defined in lines 3033 of Fig. G.16). TRansaction derived classes inherit these member functions from transaction and use them to gain access to class transaction's private data members.

Class TRansaction also declares a pure virtual function `execute` (line 22 of Fig. G.15). It does not make sense to provide an implementation for this member function, because a generic transaction cannot be executed. Thus, we declare this member function to be a pure virtual function and force each TRansaction derived class to provide its own concrete implementation that executes that particular type of transaction.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1308]

 PREV

page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1311]

**Figure G.19. Withdrawal class definition.**

(This item is displayed on page 1308 in the print version)

```
1 // Withdrawal.h
2 // Withdrawal class definition. Represents a withdrawal transaction.
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 #include "Transaction.h" // Transaction class definition
7 class Keypad; // forward declaration of class Keypad
8 class CashDispenser; // forward declaration of class CashDispenser
9
10 class Withdrawal : public Transaction
11 {
12 public:
13 Withdrawal(int, Screen &, BankDatabase &, Keypad &, CashDispenser &);
14 virtual void execute(); // perform the transaction
15 private:
16 int amount; // amount to withdraw
17 Keypad &keypad; // reference to ATM's keypad
18 CashDispenser &cashDispenser; // reference to ATM's cash dispenser
19 int displayMenuOfAmounts() const; // display the withdrawal menu
20 }; // end class Withdrawal
21
22 #endif // WITHDRAWAL_H
```

**Figure G.20. Withdrawal class member-function definitions.**

(This item is displayed on pages 1309 - 1311 in the print version)

```

1 // Withdrawal.cpp
2 // Member-function definitions for class Withdrawal.
3 #include "Withdrawal.h" // Withdrawal class definition
4 #include "Screen.h" // Screen class definition
5 #include "BankDatabase.h" // BankDatabase class definition
6 #include "Keypad.h" // Keypad class definition
7 #include "CashDispenser.h" // CashDispenser class definition
8
9 // global constant that corresponds to menu option to cancel
10 const static int CANCELED = 6;
11
12 // Withdrawal constructor initialize class's data members
13 Withdrawal::Withdrawal(int userAccountNumber, Screen &atmScreen,
14 BankDatabase &atmBankDatabase, Keypad &atmKeypad,
15 CashDispenser &atmCashDispenser)
16 : Transaction(userAccountNumber, atmScreen, atmBankDatabase),
17 keypad(atmKeypad), cashDispenser(atmCashDispenser)
18 {
19 // empty body
20 } // end Withdrawal constructor
21
22 // perform transaction; overrides Transaction's pure virtual function
23 void Withdrawal::execute()
24 {
25 bool cashDispensed = false; // cash was not dispensed yet
26 bool transactionCanceled = false; // transaction was not canceled yet
27
28 // get references to bank database and screen
29 BankDatabase &bankDatabase = getBankDatabase();
30 Screen &screen = getScreen();
31
32 // loop until cash is dispensed or the user cancels
33 do
34 {
35 // obtain the chosen withdrawal amount from the user
36 int selection = displayMenuOfAmounts();
37
38 // check whether user chose a withdrawal amount or canceled
39 if (selection != CANCELED)
40 {
41 amount = selection; // set amount to the selected dollar amount
42
43 // get available balance of account involved
44 double availableBalance =
45 bankDatabase.getAvailableBalance(getAccountNumber());
46
47 // check whether the user has enough money in the account
48 if (amount <= availableBalance)
49 {

```

```

50 // check whether the cash dispenser has enough money
51 if (cashDispenser.isSufficientCashAvailable(amount))
52 {
53 // update the account involved to reflect withdrawal
54 bankDatabase.debit(getAccountNumber(), amount);
55
56 cashDispenser.dispenseCash(amount); // dispense cash
57 cashDispensed = true; // cash was dispensed
58
59 // instruct user to take cash
60 screen.displayMessageLine(
61 "\nPlease take your cash from the cash dispenser.");
62 } // end if
63 else // cash dispenser does not have enough cash
64 screen.displayMessageLine(
65 "\nInsufficient cash available in the ATM."
66 "\n\nPlease choose a smaller amount.");
67 } // end if
68 else // not enough money available in user's account
69 {
70 screen.displayMessageLine(
71 "\nInsufficient funds in your account."
72 "\n\nPlease choose a smaller amount.");
73 } // end else
74 } // end if
75 else // user chose cancel menu option
76 {
77 screen.displayMessageLine("\nCanceling transaction...");
78 transactionCanceled = true; // user canceled the transaction
79 } // end else
80 } while (!cashDispensed && !transactionCanceled); // end do...while
81 } // end function execute
82
83 // display a menu of withdrawal amounts and the option to cancel;
84 // return the chosen amount or 0 if the user chooses to cancel
85 int Withdrawal::displayMenuOfAmounts() const
86 {
87 int userChoice = 0; // local variable to store return value
88
89 Screen &screen = getScreen(); // get screen reference
90
91 // array of amounts to correspond to menu numbers
92 int amounts[] = { 0, 20, 40, 60, 100, 200 };
93
94 // loop while no valid choice has been made
95 while (userChoice == 0)
96 {
97 // display the menu
98 screen.displayMessageLine("\nWithdrawal options:");

```

```

99 screen.displayMessageLine("1 - $20");
100 screen.displayMessageLine("2 - $40");
101 screen.displayMessageLine("3 - $60");
102 screen.displayMessageLine("4 - $100");
103 screen.displayMessageLine("5 - $200");
104 screen.displayMessageLine("6 - Cancel transaction");
105 screen.displayMessage("\nChoose a withdrawal option (1-6): ");
106
107 int input = keypad.getInput(); // get user input through keypad
108
109 // determine how to proceed based on the input value
110 switch (input)
111 {
112 case 1: // if the user chose a withdrawal amount
113 case 2: // (i.e., chose option 1, 2, 3, 4 or 5), return the
114 case 3: // corresponding amount from amounts array
115 case 4:
116 case 5:
117 userChoice = amounts[input]; // save user's choice
118 break;
119 case CANCELED: // the user chose to cancel
120 userChoice = CANCELED; // save user's choice
121 break;
122 default: // the user did not enter a value from 1-6
123 screen.displayMessageLine(
124 "\nInvalid selection. Try again.");
125 } // end switch
126 } // end while
127
128 return userChoice; // return withdrawal amount or CANCELED
129 } // end function displayMenuOfAmounts

```

## Withdrawal Class Member-Function Definitions

Figure G.20 contains the member-function definitions for class `Withdrawal`. Line 3 `#includes` the class's definition, and lines 47 `#include` the definitions of the other classes used in `Withdrawal`'s member functions. Line 11 declares a global constant corresponding to the cancel option on the withdrawal menu. We will soon discuss how the class uses this constant.

Class `Withdrawal`'s constructor (defined in lines 1320 of Fig. G.20) has five parameters. It uses a base-class initializer in line 16 to pass parameters `userAccountNumber`, `atm`-`Screen` and `atmBankDatabase` to base class `transaction`'s constructor to set the data members that `Withdrawal` inherits from `TRansaction`. The constructor also takes references `atmKeypad` and `atmCashDispenser` as parameters and assigns them to reference data members `keypad` and `cashDispenser` using member initializers (line 17).

Class `Withdrawal` overrides `transaction`'s pure virtual function `execute` with a concrete implementation (lines 2381) that performs the steps involved in a withdrawal. Line 25 declares and initializes a local `bool` variable `cashDispensed`. This variable indicates whether cash has been dispensed (i.e., whether the transaction has completed successfully) and is initially `false`. Line 26 declares and initializes to `false` a `bool` variable `transactionCanceled` that indicates whether the transaction has been canceled by the user. Lines 2930 get references to the bank database and the ATM's screen by invoking member functions inherited from base class `transaction`.

[Page 1312]

Lines 3380 contain a `do...while` statement that executes its body until cash is dispensed (i.e., until `cashDispensed` becomes `true`) or until the user chooses to cancel (i.e., until `transactionCanceled` becomes `true`). We use this loop to continuously return the user to the start of the transaction if an error occurs (i.e., the requested withdrawal amount is greater than the user's available balance or greater than the amount of cash in the cash dispenser). Line 36 displays a menu of withdrawal amounts and obtains a user selection by calling private utility function `displayMenuOfAmounts` (defined in lines 85129). This member function displays the menu of amounts and returns either an `int` withdrawal amount or the `int` constant `CANCELED` to indicate that the user has chosen to cancel the transaction.

Member function `displayMenuOfAmounts` (lines 85129) first declares local variable `userChoice` (initially 0) to store the value that the member function will return (line 87). Line 89 gets a reference to the screen by calling member function `getScreen` inherited from base class `transaction`. Line 92 declares an integer array of withdrawal amounts that correspond to the amounts displayed in the withdrawal menu. We ignore the first element in the array (index 0) because the menu has no option 0. The `while` statement at lines 95126 repeats until `userChoice` takes on a value other than 0. We will see shortly that this occurs when the user makes a valid selection from the menu. Lines 98105 display the withdrawal menu on the screen and prompt the user to enter a choice. Line 107 obtains integer `input` through the keypad. The `switch` statement at lines 110125 determines how to proceed based on the user's input. If the user selects a number between 1 and 5, line 117 sets `userChoice` to the value of the element in `amounts` at index `input`. For example, if the user enters 3 to withdraw \$60, line 117 sets `userChoice` to the value of `amounts[ 3 ]` (i.e., 60). Line 118 terminates the `switch`. Variable `userChoice` no longer equals 0, so the `while` at lines 95126 terminates and line 128 returns `userChoice`. If the user selects the cancel menu option, lines 120121 execute, setting `userChoice` to `CANCELED` and causing the member function to return this value. If the user does not enter a valid menu selection, lines 123124 display an error message and the user is returned to the withdrawal menu.

The `if` statement at line 39 in member function `execute` determines whether the user has selected a withdrawal amount or chosen to cancel. If the user cancels, lines 7778 execute to display an appropriate message to the user and set `transactionCanceled` to `true`. This causes the loop-continuation test in line 80 to fail and control to return to the calling member function (i.e., ATM member function `performTransactions`). If the user has chosen a withdrawal amount, line 41 assigns local variable `selection` to data member `amount`. Lines 4445 retrieve the available balance of the current user's Account and store it in a local `double` variable `availableBalance`. Next, the `if` statement at line 48 determines whether the selected amount is less than or equal to the user's available balance. If it is not,

lines 7072 display an appropriate error message. Control then continues to the end of the do...while, and the loop repeats because both cashDispensed and transactionCanceled are still false. If the user's balance is high enough, the if statement at line 51 determines whether the cash dispenser has enough money to satisfy the withdrawal request by invoking the cashDispenser's isSufficientCashAvailable member function. If this member function returns false, lines 6466 display an appropriate error message and the do...while repeats. If sufficient cash is available, then the requirements for the withdrawal are satisfied, and line 54 debits amount from the user's account in the database. Lines 5657 then instruct the cash dispenser to dispense the cash to the user and set cashDispensed to TRue. Finally, lines 6061 display a message to the user that cash has been dispensed. Because cashDispensed is now TRue, control continues after the do...while. No additional statements appear below the loop, so the member function returns control to class ATM.

---

[Page 1313]

Notice that, in the function calls in lines 6466 and lines 7072, we divide the argument to Screen member function displayMessageLine into two string literals, each placed on a separate line in the program. We do so because each argument is too long to fit on a single line. C++ concatenates (i.e., combines) string literals adjacent to each other, even if they are on separate lines. For example, if you write "Happy " "Birthday" in a program, C++ will view these two adjacent string literals as the single string literal "Happy Birthday". As a result, when lines 6466 execute, displayMessageLine receives a single string as a parameter, even though the argument in the function call appears as two string literals.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1314]

## Deposit Class Member-Function Definitions

Figure G.22 presents the implementation file for class Deposit. Line 3 `#includes` the Deposit class definition, and lines 47 `#include`s the class definitions of the other classes used in Deposit's member functions. Line 9 declares a constant CANCELED that corresponds to the value a user enters to cancel a deposit. We will soon discuss how the class uses this constant.

Like class `withdrawal`, class `Deposit` contains a constructor (lines 1219) that passes three parameters to base class `transaction`'s constructor using a base-class initializer (line 15). The constructor also has parameters `atmKeypad` and `atmDepositsSlot`, which it assigns to its corresponding data members (line 16).

Member function `execute` (lines 2262) overrides pure `virtual` function `execute` in base class `transaction` with a concrete implementation that performs the steps required in a deposit transaction. Lines 2425 get references to the database and the screen. Line 27 prompts the user to enter a deposit amount by invoking private utility function `promptForDepositAmount` (defined in lines 6581) and sets data member `amount` to the value returned. Member function `promptForDepositAmount` asks the user to enter a deposit amount as an integer number of cents (because the ATM's keypad does not contain a decimal point; this is consistent with many real ATMs) and returns the double value representing the dollar amount to be deposited.

---

[Page 1315]

---

[Page 1316]

Line 67 in member function `promptForDepositAmount` gets a reference to the ATM's screen. Lines 7071 display a message on the screen asking the user to input a deposit amount as a number of cents or "0" to cancel the transaction. Line 72 receives the user's input from the keypad. The `if` statement at lines 7580 determines whether the user has entered a real deposit amount or chosen to cancel. If the user chooses to cancel, line 76 returns the constant CANCELED. Otherwise, line 79 returns the deposit amount after converting from the number of cents to a dollar amount by casting `input` to a `double`, then dividing by 100. For example, if the user enters 125 as the number of cents, line 79 returns 125.0 divided by 100, or 1.25125 cents is \$1.25.

The `if` statement at lines 3061 in member function `execute` determines whether the user has chosen to cancel the transaction instead of entering a deposit amount. If the user cancels, line 60 displays an appropriate message, and the member function returns. If the user enters a deposit amount, lines 3336

instruct the user to insert a deposit envelope with the correct amount. Recall that Screen member function `displayDollarAmount` outputs a double formatted as a dollar amount.

Line 39 sets a local `bool` variable to the value returned by `depositSlot`'s `isEnvelopeReceived` member function, indicating whether a deposit envelope has been received. Recall that we coded member function `isEnvelopeReceived` (lines 710 of Fig. G.10) to always return `true`, because we are simulating the functionality of the deposit slot and assume that the user always inserts an envelope. However, we code member function `execute` of class `Deposit` to test for the possibility that the user does not insert an envelope—good software engineering demands that programs account for all possible return values. Thus, class `Deposit` is prepared for future versions of `isEnvelopeReceived` that could return `false`. Lines 4450 execute if the deposit slot receives an envelope. Lines 4447 display an appropriate message to the user. Line 50 then credits the deposit amount to the user's account in the database. Lines 5455 will execute if the deposit slot does not receive a deposit envelope. In this case, we display a message to the user stating that the ATM has canceled the transaction. The member function then returns without modifying the user's account.

 PREVNEXT 

page footer

---

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)
[NEXT ▶](#)

[Page 1316 (continued)]

## G.13. Test Program ATMCaseStudy.cpp

`ATMCaseStudy.cpp` ([Fig. G.23](#)) is a simple C++ program that allows us to start, or "turn on," the ATM and test the implementation of our ATM system model. The program's `main` function (lines 6-11) does nothing more than instantiate a new `ATM` object named `atm` (line 8) and invoke its `run` member function (line 9) to start the ATM.

**Figure G.23. `ATMCaseStudy.cpp` starts the ATM system.**

```

1 // ATMCaseStudy.cpp
2 // Driver program for the ATM case study.
3 #include "ATM.h" // ATM class definition
4
5 // main function creates and runs the ATM
6 int main()
7 {
8 ATM atm; // create an ATM object
9 atm.run(); // tell the ATM to start
10 return 0;
11 } // end main

```

[◀ PREV](#)
[NEXT ▶](#)

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1317]

## G.14. Wrap-Up

Congratulations on completing the entire software engineering ATM case study! We hope you found this experience to be valuable and that it reinforced many of the concepts that you learned in [Chapters 113](#). We would sincerely appreciate your comments, criticisms and suggestions. You can reach us at [deitel@deitel.com](mailto:deitel@deitel.com). We will respond promptly.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1318]

## Appendix H. UML 2: Additional Diagram Types

[Section H.1. Introduction](#)

[Section H.2. Additional Diagram Types](#)

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1318 (continued)]

## H.1. Introduction

If you read the optional Software Engineering Case Study sections in [Chapters 27, 9](#) and [13](#), you should now have a comfortable grasp of the UML diagram types that we use to model our ATM system. The case study is intended for use in first- or second-semester courses, so we limit our discussion to a concise subset of the UML. The UML 2 provides a total of 13 diagram types. The end of [Section 2.8](#) summarizes the six diagram types that we use in the case study. This appendix lists and briefly defines the seven remaining diagram types.

 PREV

NEXT 

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1319]

- [Composite structure diagrams](#) model the internal structure of a complex object at runtime. New in UML 2, they allow system designers to hierarchically decompose a complex object into smaller parts. Composite structure diagrams are beyond the scope of our case study. They are more appropriate for larger industrial applications, which exhibit complex groupings of objects at execution time.
- [Interaction overview diagrams](#), new in UML 2, provide a summary of control flow in the system by combining elements of several types of behavioral diagrams (e.g., activity diagrams, sequence diagrams).
- [Timing diagrams](#), also new in UML 2, model the timing constraints imposed on stage changes and interactions between objects in a system.

To learn more about these diagrams and advanced UML topics, please visit [www.uml.org](http://www.uml.org) and the Web resources listed at the ends of [Section 1.17](#) and [Section 2.8](#).

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1320]

## Appendix I. C++ Internet and Web Resources

This appendix contains a list of C++ resources that are available on the Internet and the World Wide Web. These resources include FAQs (Frequently Asked Questions), tutorials, links to the ANSI/ISO C++ standard, information about popular C++ compilers and access to free compilers, demos, books, tutorials, software tools, articles, interviews, conferences, journals and magazines, online courses, newsgroups and career resources. For additional information about the American National Standards Institute (ANSI) and its activities related to C++, visit [www.ansi.org](http://www.ansi.org).

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1321]

[www.programmersheaven.com/zone3](http://www.programmersheaven.com/zone3)

This site provides links to articles, tutorials, development tools, an extensive collection of free C++ libraries and source code.

[www.hal9k.com/cug](http://www.hal9k.com/cug)

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

[www.devx.com](http://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.cprogramming.com](http://www.cprogramming.com)

This site contains interactive tutorials, quizzes, articles, journals, compiler downloads, book recommendations and free source code.

[www.acm.org/crossroads/xrds3-2/ovp32.html](http://www.acm.org/crossroads/xrds3-2/ovp32.html)

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

[www.comeaucomputing.com/resources](http://www.comeaucomputing.com/resources)

Comeau Computing's site links to technical discussions, FAQs (including one devoted to templates), user groups, newsgroups and an online C++ compiler.

[www.exciton.cs.rice.edu/CppResources](http://www.exciton.cs.rice.edu/CppResources)

The site provides a document that summarizes the technical aspects of C++. The site also discusses the differences between Java and C++.

[www.accu.informika.ru/resources/public/terse/cpp.htm](http://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](http://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.

[directory.google.com/Top/Computers/Programming/Languages/C++/Resources/Directories](http://directory.google.com/Top/Computers/Programming/Languages/C++/Resources/Directories)

Google's C++ resources directory ranks the most useful C++ sites.

[www.compinfo-center.com/c++.htm](http://www.compinfo-center.com/c++.htm)

This site provides links to C++ FAQs, newsgroups and magazines.

[www.apl.jhu.edu/~paulmac/c++-references.html](http://www.apl.jhu.edu/~paulmac/c++-references.html)

This site contains book reviews and recommendations for introductory, intermediate and advanced C++ programmers and links to online C++ resources, including books, magazines and tutorials.

[www.cmcrossroads.com/bradapp/links/cplusplus-links.html](http://www.cmcrossroads.com/bradapp/links/cplusplus-links.html)

This site divides links into categories, including Resources and Directories, Projects and Working Groups, Libraries, Training, Tutorials, Publications and Coding Conventions.

[www.codeproject.com](http://www.codeproject.com)

Articles, code snippets, user discussions, books and news about C++, C# and .NET programming are available at this site.

[www.austinlinks.com/CPlusPlus](http://www.austinlinks.com/CPlusPlus)

Quadrailay Corporation's site links to numerous C++ resources, including Visual C++/MFC Libraries, C++ programming information, C++ career resources and a list of tutorials and other online tools for learning C++.

[www.csci.csusb.edu/dick/c++std](http://www.csci.csusb.edu/dick/c++std)

Links to the ANSI/ISO C++ Standard and the `comp.std.c++` Usenet group are available at this site.

[www.research.att.com/~bs/homepage.html](http://www.research.att.com/~bs/homepage.html)

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

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 1322 (continued)]

## I.2. Tutorials

[www.cprogramming.com/tutorial.html](http://www.cprogramming.com/tutorial.html)

This site offers a step-by-step tutorial, with sample code, that covers file I/O, recursion, binary trees, template classes and more.

[www.programmersheaven.com/zone3/cat34](http://www.programmersheaven.com/zone3/cat34)

Free tutorials that are appropriate for many skill levels are available at this site.

[www.programmershelp.co.uk/c%2B%2Btutorials.php](http://www.programmershelp.co.uk/c%2B%2Btutorials.php)

This site contains free online courses and a comprehensive list of C++ tutorials. This site also provides FAQs, downloads and other resources.

[www.codeproject.com/script/articles/beginners.asp](http://www.codeproject.com/script/articles/beginners.asp)

This site lists tutorials and articles available for C++ beginners.

[www.eng.hawaii.edu/Tutor/Make](http://www.eng.hawaii.edu/Tutor/Make)

This site provides a tutorial that describes how to create makefiles.

[www.cpp-home.com](http://www.cpp-home.com)

Free tutorials, discussions, chat rooms, articles, compilers, forums and online quizzes related to C++ are available at this site. The C++ tutorials cover such topics as ActiveX/COM, MFC and graphics.

[www.codebeach.com](http://www.codebeach.com)

Code Beach contains source code, tutorials, books and links to major programming languages, including C++, Java, ASP, Visual Basic, XML, Python, Perl and C#.

[www.kegel.com/academy/tutorials.html](http://www.kegel.com/academy/tutorials.html)

This site provides links to tutorials on C, C++ and assembly languages.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1322 (continued)]

## I.3. FAQs

[www.faqs.org/faqs/by-newsgroup/comp/comp.lang.c++.html](http://www.faqs.org/faqs/by-newsgroup/comp/comp.lang.c++.html)

This site consists of links to FAQs and tutorials gathered from the Comp.Lang.C++ newsgroup.

[www.eskimo.com/~scs/C-faq/top.html](http://www.eskimo.com/~scs/C-faq/top.html)

This C FAQ list contains topics such as pointers, memory allocation and strings.

[www.technion.ac.il/technion/tcc/usg/Ref/C\\_Programming.html](http://www.technion.ac.il/technion/tcc/usg/Ref/C_Programming.html)

This site contains C/C++ programming references, including FAQs and tutorials.

[www.faqs.org/faqs/by-newsgroup/comp/comp.compilers.html](http://www.faqs.org/faqs/by-newsgroup/comp/comp.compilers.html)

This site contains a list of FAQs generated in the comp.compilers newsgroup.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1322 (continued)]

## I.4. Visual C++

[msdn.microsoft.com/visualc](http://msdn.microsoft.com/visualc)

Microsoft's Visual C++ page provides information about the latest release of Visual C++ .NET.

[www.freeprogrammingresources.com/visualcpp.html](http://www.freeprogrammingresources.com/visualcpp.html)

This site contains free programming resources for Visual C++ programmers, including tutorials and sample programming applications.

[www.mvps.org/vcfaq](http://www.mvps.org/vcfaq)

The Most Valuable Professional (MVP) site contains a Visual C++ FAQ.

[www.onesmartclick.com/programming/visual-cpp.html](http://www.onesmartclick.com/programming/visual-cpp.html)

This site contains Visual C++ tutorials, online books, tips, tricks, FAQs and debugging.

[◀ PREV](#)[NEXT ▶](#)

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1323]

## I.5. Newsgroups

[ai.kaist.ac.kr/~ymkim/Program/c++.html](http://ai.kaist.ac.kr/~ymkim/Program/c++.html)

This site offers tutorials, libraries, popular compilers, FAQs and newsgroups, including `comp.lang.c++`.

[www.coding-zone.co.uk/cpp/cnewsgroups.shtml](http://www.coding-zone.co.uk/cpp/cnewsgroups.shtml)

This site includes links to several C++ newsgroups including `comp.lang.c`, `comp.lang.c++` and `comp.lang.c++.moderated`, to name a few.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1323 (continued)]

## I.6. Compilers and Development Tools

[msdn.microsoft.com/visualc](http://msdn.microsoft.com/visualc)

The Microsoft Visual C++ site provides product information, overviews, supplemental materials and ordering information for the Visual C++ compiler.

[lab.msdn.microsoft.com/express/visualc/](http://lab.msdn.microsoft.com/express/visualc/)

You can download the Microsoft Visual C++ Express Beta for free from this Web site.

[msdn.microsoft.com/visualc/vctoolkit2003/](http://msdn.microsoft.com/visualc/vctoolkit2003/)

Visit this site to download the Visual C++ Toolkit 2003

[www.borland.com/bcppbuilder](http://www.borland.com/bcppbuilder)

This is a link to the Borland C++ Builder 6. A free command-line version is available for download.

[www.thefreecountry.com/developercity/cccompilers.shtml](http://www.thefreecountry.com/developercity/cccompilers.shtml)

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

[www.faqs.org/faqs/by-newsgroup/comp/comp.compilers.html](http://www.faqs.org/faqs/by-newsgroup/comp/comp.compilers.html)

This site lists FAQs generated within the comp.compilers newsgroup.

[www.compilers.net/Dir/Free/Compilers/CCpp.htm](http://www.compilers.net/Dir/Free/Compilers/CCpp.htm)

Compilers.net is designed to help users locate compilers.

[developer.intel.com/software/products/compilers/cwin/index.htm](http://developer.intel.com/software/products/compilers/cwin/index.htm)

The Intel® C++ Compiler 8.1 for Windows is available at this site.

[www.intel.com/software/products/compilers/clin/index.htm](http://www.intel.com/software/products/compilers/clin/index.htm)

The Intel® C++ Compiler 8.1 for Linux is available at this site.

[www.symbian.com/developer/development/cppdev.html](http://www.symbian.com/developer/development/cppdev.html)

Symbian provides a C++ Developer's Pack and links to various resources, including code and development tools for C++ programmers (particularly those working with the Symbian operating system).

[www.gnu.org/software/gcc/gcc.html](http://www.gnu.org/software/gcc/gcc.html)

The GNU Compiler Collection (GCC) site includes links to download GNU compilers for C++, C, Objective C and other languages.

[www.bloodshed.net/devcpp.html](http://www.bloodshed.net/devcpp.html)

Bloodshed Dev-C++ is a free integrated development environment for C++.

### **Third Party Vendors That Provide Libraries for Precise Financial Calculations**

[www.roguewave.com/products/sourcepro/analysis/](http://www.roguewave.com/products/sourcepro/analysis/)

RogueWave Software's SourcePro Analysis libraries include classes for precise monetary calculations, data analysis and essential mathematical algorithms.

[www.boic.com/numorder.htm](http://www.boic.com/numorder.htm)

Base One International Corporation's Bas/1 Number class implements highly precise mathematical calculations.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1324]

## I.7. Standard Template Library

### Tutorials

[www.cs.brown.edu/people/jak/programming/stl-tutorial/tutorial.html](http://www.cs.brown.edu/people/jak/programming/stl-tutorial/tutorial.html)

This STL tutorial is organized by examples, philosophy, components and extending STL. You will find code examples using the STL components, useful explanations and helpful diagrams.

[www.xraylith.wisc.edu/~khan/software/stl/os\\_examples/examples.html](http://www.xraylith.wisc.edu/~khan/software/stl/os_examples/examples.html)

This site is helpful for people just learning about the STL. You will find an introduction to the STL and ObjectSpace STL Tool Kit examples.

[cplus.about.com/od/stltutorial/l/blstl.htm](http://cplus.about.com/od/stltutorial/l/blstl.htm)

This tutorial discusses all of the features of the STL.

[www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html](http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html)

This Linux-focused site includes an STL tutorial and examples.

[www.cs.rpi.edu/~musser/stl-book](http://www.cs.rpi.edu/~musser/stl-book)

The RPI STL site includes information on how STL differs from other C++ libraries and on how to compile programs that use STL. A list of STL include files, example programs that use STL, STL Container Classes, and STL Iterator Categories are available. The site also provides an STL-compatible compiler list, FTP sites for STL source code and related materials.

### References

[www.sgi.com/tech/stl](http://www.sgi.com/tech/stl)

The Silicon Graphics Standard Template Library Programmer's Guide is a useful resource for STL

information. You can download STL source code from this site, and find the latest information, design documentation and links to other STL resources.

[www.byte.com/art/9510/sec12/art3.htm](http://www.byte.com/art/9510/sec12/art3.htm)

The Byte Magazine site has a copy of an article written by one of the creators of the Standard Template Library, Alexander Stepanov, that provides information on the use of the STL in generic programming.

## ANSI/ISO C++ Standard

[www.ansi.org](http://www.ansi.org)

You can purchase a copy of the C++ Standard from this site.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1326]

## Outline

[J.1](#) Introduction

[J.2](#) Editing XHTML

[J.3](#) First XHTML Example

[J.4](#) Headers

[J.5](#) Linking

[J.6](#) Images

[J.7](#) Special Characters and More Line Breaks

[J.8](#) Unordered Lists

[J.9](#) Nested and Ordered Lists

[J.10](#) Basic XHTML Tables

[J.11](#) Intermediate XHTML Tables and Formatting

[J.12](#) Basic XHTML Forms

[J.13](#) More Complex XHTML Forms

[J.14](#) Internet and World Wide Web Resources

[Summary](#)

[Terminology](#)



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1326 (continued)]

## J.1. Introduction

In this appendix, we introduce **XHTML**<sup>[1]</sup> the Extensible HyperText Markup Language for creating Web content. Unlike procedural programming languages such as C, Fortran, Cobol and Visual Basic, XHTML is a **markup language** that specifies the format of text that is displayed in a Web browser, such as Microsoft's Internet Explorer or Netscape's Communicator.

<sup>[1]</sup> XHTML has replaced the HyperText Markup Language (HTML) as the primary means of describing Web content. XHTML provides more robust, richer and more extensible features than HTML. For more on XHTML/HTML, visit [www.w3.org/markup](http://www.w3.org/markup).

One key issue when using XHTML is the separation of the **presentation of a document** (i.e., the document's appearance when rendered by a browser) from the structure of the document's information. Throughout this appendix, we will discuss this issue in depth.

In this appendix, we build several complete Web pages featuring text, hyperlinks, images, horizontal rules and line breaks. We also discuss more substantial XHTML features, including presentation of information in tables and incorporating forms for collecting information from a Web-page visitor. By the end of this appendix, you will be familiar with the most commonly used XHTML features and will be able to create more complex Web documents. In this appendix, we do not present any C++ programming.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1327]

## Good Programming Practice J.1



Assign documents file names that describe their functionality. This practice can help you identify documents faster. It also helps people who want to link to a page, by giving them an easy-to-remember name. For example, if you are writing an XHTML document that contains product information, you might want to call it `products.html`.

Machines running specialized software called a **Web server** store XHTML documents. Clients (e.g., Web browsers) request specific resources, such as XHTML documents, from the Web server. For example, typing [www.deitel.com/books/downloads.htm](http://www.deitel.com/books/downloads.htm) into a Web browser's address field requests `downloads.htm` from the Web server running at [www.deitel.com](http://www.deitel.com). This document is located in a directory named `books`.

[◀ PREV](#)

page footer

[NEXT ▶](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1328]

Lines 13 are required in XHTML documents to conform with proper XHTML **syntax**. Lines 56 are **XHTML comments**. XHTML document creators insert comments to improve markup readability and to describe the content of a document. Comments also help other people read and understand an XHTML document's markup and content. Comments do not cause the browser to perform any action when the user loads the XHTML document into the Web browser to view the document. XHTML comments always start with `<!--` and end with `-->`. Each of our XHTML examples includes comments that specify the figure number and file name and provide a brief description of the example's purpose. Subsequent examples include comments in the markup, especially to highlight new features.

## Good Programming Practice J.2



Place comments throughout your markup. Comments help other programmers understand the markup, assist in debugging and list useful information that you do not want the browser to render. Comments also help you understand your own markup when you revisit a document for modifications or updates in the future.

XHTML markup contains text that represents the content of a document and **elements** that specify a document's structure. Some important elements of an XHTML document include the **html element**, the **head element** and the **body element**. The **html element** encloses the **head section** (represented by the **head element**) and the **body section** (represented by the **body element**). The head section contains information about the XHTML document, such as the **title** of the document. The head section also can contain special document-formatting instructions called **style sheets** and client-side programs called **scripts** for creating dynamic Web pages. The body section contains the page's content that the browser displays when the user visits the Web page.

XHTML documents delimit an element with **start** and **end** tags. A start tag consists of the element name in angle brackets (e.g., `<html>`). An end tag consists of the element name preceded by a `/` in angle brackets (e.g., `</html>`). In this example, lines 8 and 16 define the start and end of the **html element**. Note that the end tag in line 16 has the same name as the start tag, but is preceded by a `/` inside the angle brackets. Many start tags define **attributes** that provide additional information about an element. Browsers can use this additional information to determine how to process the element. Each attribute has a **name** and a **value**, separated by an equal sign (`=`). Line 8 specifies a required attribute (`xmlns`) and value (<http://www.w3.org/1999/xhtml>) for the **html element** in an XHTML document.

## Common Programming Error J.1



Not enclosing attribute values in either single or double quotes is a syntax error.

## Common Programming Error J.2



Using uppercase letters in an XHTML element or attribute name is a syntax error.

An XHTML document divides the `html` element into two sections `head` and `body`. Lines 911 define the Web page's head section with a `head` element. Line 10 specifies a `title` element. This is called a **nested element**, because it is enclosed in the `head` element's start and end tags. The `head` element also is a nested element, because it is enclosed in the `html` element's start and end tags. The **title element** describes the Web page. Titles usually appear in the **title bar** at the top of the browser window and also as the text identifying a page when users add the page to their list of Favorites or Bookmarks, which enable users to return to their favorite sites. Search engines (i.e., sites that allow users to search the Web) also use the `title` for cataloging purposes.

---

[Page 1329]

## Good Programming Practice J.3



Indenting nested elements emphasizes a document's structure and promotes readability.

## Common Programming Error J.3



XHTML does not permit tags to overlap a nested element's end tag must appear in the document before the enclosing element's end tag. For example, the nested XHTML tags `<head><title>hello</head></title>` cause a syntax error, because the enclosing `head` element's ending `</head>` tag appears before the nested `title` element's ending `</title>` tag.

## Good Programming Practice J.4



Use a consistent title naming convention for all pages on a site. For example, if a site is named "Bailey's Web Site," then the title of the main page might be "Bailey's Web SiteLinks." This practice can help users better understand the Web site's structure.

Line 13 opens the document's body element. The body section of an XHTML document specifies the document's content, which may include text and tags.

Some tags, such as the **paragraph tags** (`<p>` and `</p>`) in line 14, mark up text for display in a browser. All text placed between the `<p>` and `</p>` tags form one paragraph. When the browser renders a paragraph, a blank line usually precedes and follows paragraph text.

This document ends with two closing tags (lines 1516). These tags close the `body` and `html` elements, respectively. The ending `</html>` tag in an XHTML document informs the browser that the XHTML markup is complete.

To view this example in Internet Explorer, perform the following steps:

1. Copy the [Appendix J](#) examples onto your machine (these examples are available on the CD-ROM that accompanies this book).
2. Launch Internet Explorer, and select Open... from the File Menu. This displays the Open dialog.
3. Click the Open dialog's Browse... button to display the Microsoft Internet Explorer file dialog.
4. Navigate to the directory containing the [Appendix J](#) examples, and select the file `main.html`; then click Open.
5. Click OK to have Internet Explorer (or any other browser) render the document. Other examples are opened in a similar manner.

At this point, your browser window should appear similar to the sample screen capture shown in Fig. J.1.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1331]

Header element `h1` (line 15) is considered the most significant header and is rendered in a larger font than the other five headers (lines 1620). Each successive header element (i.e., `h2`, `h3`, etc.) is rendered in a smaller font.

### Portability Tip J.1



The text size used to display each header element can vary significantly between browsers.

### Look-and-Feel Observation J.1



Placing a header at the top of every XHTML page helps viewers understand the purpose of each page.

### Look-and-Feel Observation J.2



Use larger headers to emphasize more important sections of a Web page.

[◀ PREV](#)

[page footer](#)

[NEXT ▶](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1333]

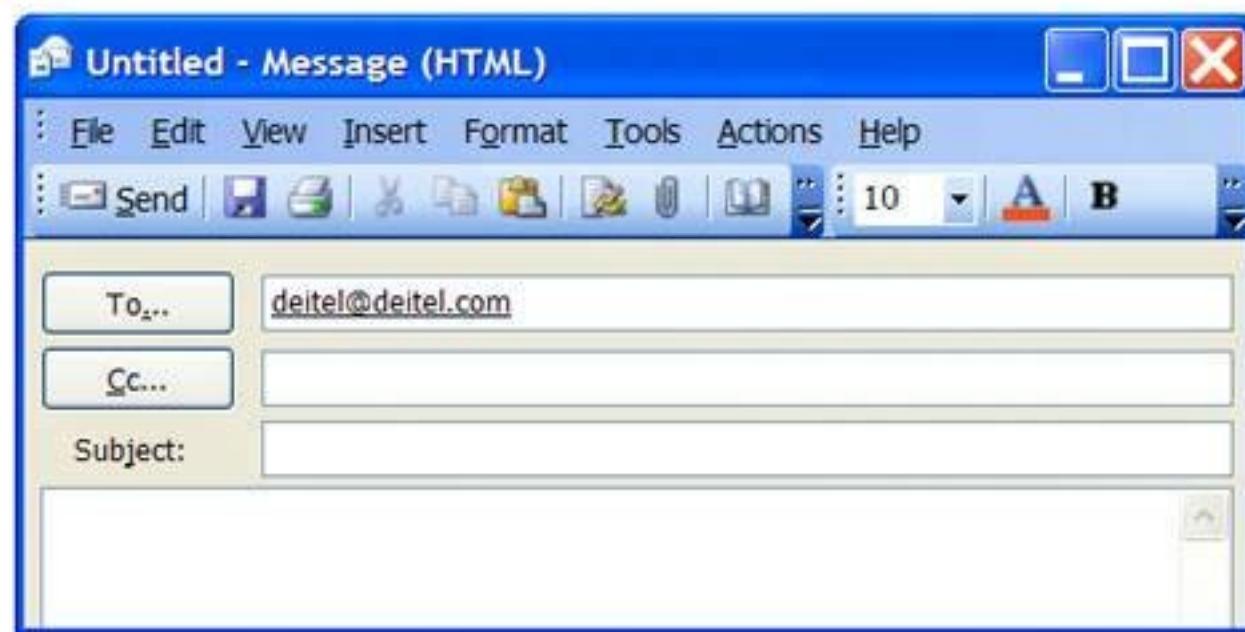
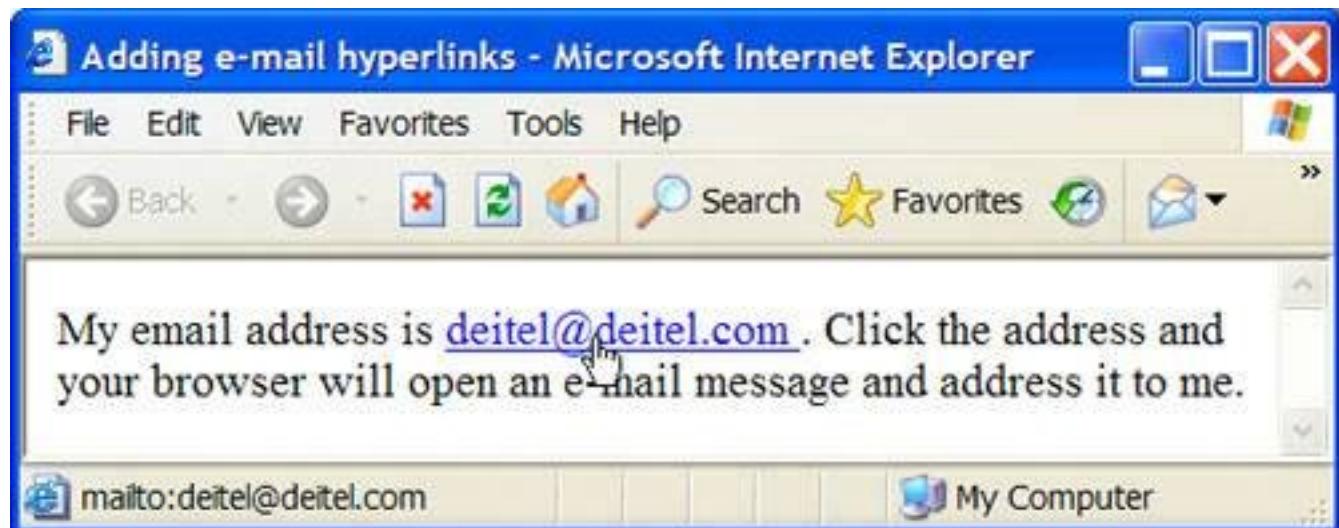
Anchors can link to e-mail addresses through a `mailto:` URL. When someone clicks this type of anchored link, most browsers launch the default e-mail program (e.g., Outlook Express) to enable the user to write an e-mail message to the linked address. [Figure J.4](#) demonstrates this type of anchor.

#### Figure J.4. Linking to an e-mail address.

(This item is displayed on pages 1333 - 1334 in the print version)

```
1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Fig. J.4: contact.html -->
6 <!-- Adding email hyperlinks. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Adding e-mail hyperlinks</title>
11 </head>
12
13 <body>
14
15 <p>My email address is
16 <a href =<.../span> "mailto:deitel@deitel.com">
17 deitel@deitel.com
18
19 . Click the address and your browser will
20 open an e-mail message and address it to me.
21 </p>
22 </body>
23 </html>
```

[\[View full size image\]](#)



Lines 1719 contain an e-mail link. The form of an e-mail anchor is `<a href = "mailto:emailaddress">...</a>`. In this case, we link to the e-mail address [deitel@deitel.com](mailto:deitel@deitel.com).

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1334]

## Performance Tip J.1



Including the `width` and `height` attributes in an `<img>` tag will help the browser load and render pages faster.

[Page 1335]

## Common Programming Error J.4



Entering new dimensions for an image that change its inherent width-to-height ratio might distort the appearance of the image. For example, if your image is 200 pixels wide and 100 pixels high, you should ensure that any new dimensions have a 2:1 width-to-height ratio.

Lines 1617 use an `img` element to insert an image in the document. The image file's location is specified with the `img` element's `src` attribute. In this case, the image is located in the same directory as this XHTML document, so only the image's file name is required. Optional attributes `width` and `height` specify the image's width and height, respectively. The document author can scale an image by increasing or decreasing the values of the image `width` and `height` attributes. If these attributes are omitted, the browser uses the image's actual width and height. Images are measured in `pixels` ("picture elements"), which represent dots of color on the screen. The image in [Fig. J.5](#) is 181 pixels wide and 238 pixels high.

Every `img` element in an XHTML document has an `alt` attribute. If a browser cannot render an image, the browser displays the `alt` attribute's value. A browser might not be able to render an image for several reasons. It might not support images as is the case with a `text-based browser` (i.e., a browser that can display only text) or the client may have disabled image viewing to reduce download time. [Figure J.5](#) shows Internet Explorer rendering the `alt` attribute's value when a document references a nonexistent image file (`fish.jpg`).

The `alt` attribute is important for creating accessible Web pages for users with disabilities, especially those with vision impairments and text-based browsers. Specialized software called a `speech synthesizer` often is used by people with disabilities. Such software applications "speak" the `alt` attribute's value so that the user knows what the browser is displaying.

Some XHTML elements (called **empty elements**) contain only attributes and do not mark up text (i.e., text is not placed between the start and end tags). Empty elements (e.g., `img`) must be terminated, either by using the **forward slash character** (`/`) inside the closing right angle bracket (`>`) of the start tag or by explicitly including the end tag. When using the forward slash character, we add a space before the forward slash to improve readability (as shown at the ends of lines 17 and 20). Rather than using the forward slash character, lines 1920 could be written with a closing `</img>` tag as follows:

```
<img src =<.../span> "cool8se.jpg" height = "238" width = "181"
 alt = "An imaginary landscape.">
```

By using images as hyperlinks, Web developers can create graphical Web pages that link to other resources. In Fig. J.6, we create six different image hyperlinks.

**Figure J.6. Using images as link anchors.**

(This item is displayed on pages 1336 - 1337 in the print version)

```
1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Fig. J.6: nav.html -->
6 <!-- Using images as link anchors. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Using images as link anchors</title>
11 </head>
12
13 <body>
14
15 <p>
16 <a href =<.../span> "links.html">
17 <img src =<.../span> "buttons/links.jpg" width = "65"
18 height = "50" alt = "Links Page" />
19

20
21 <a href =<.../span> "list.html">
22 <img src =<.../span> "buttons/list.jpg" width = "65"
23 height = "50" alt = "List Example Page" />
24

25
```

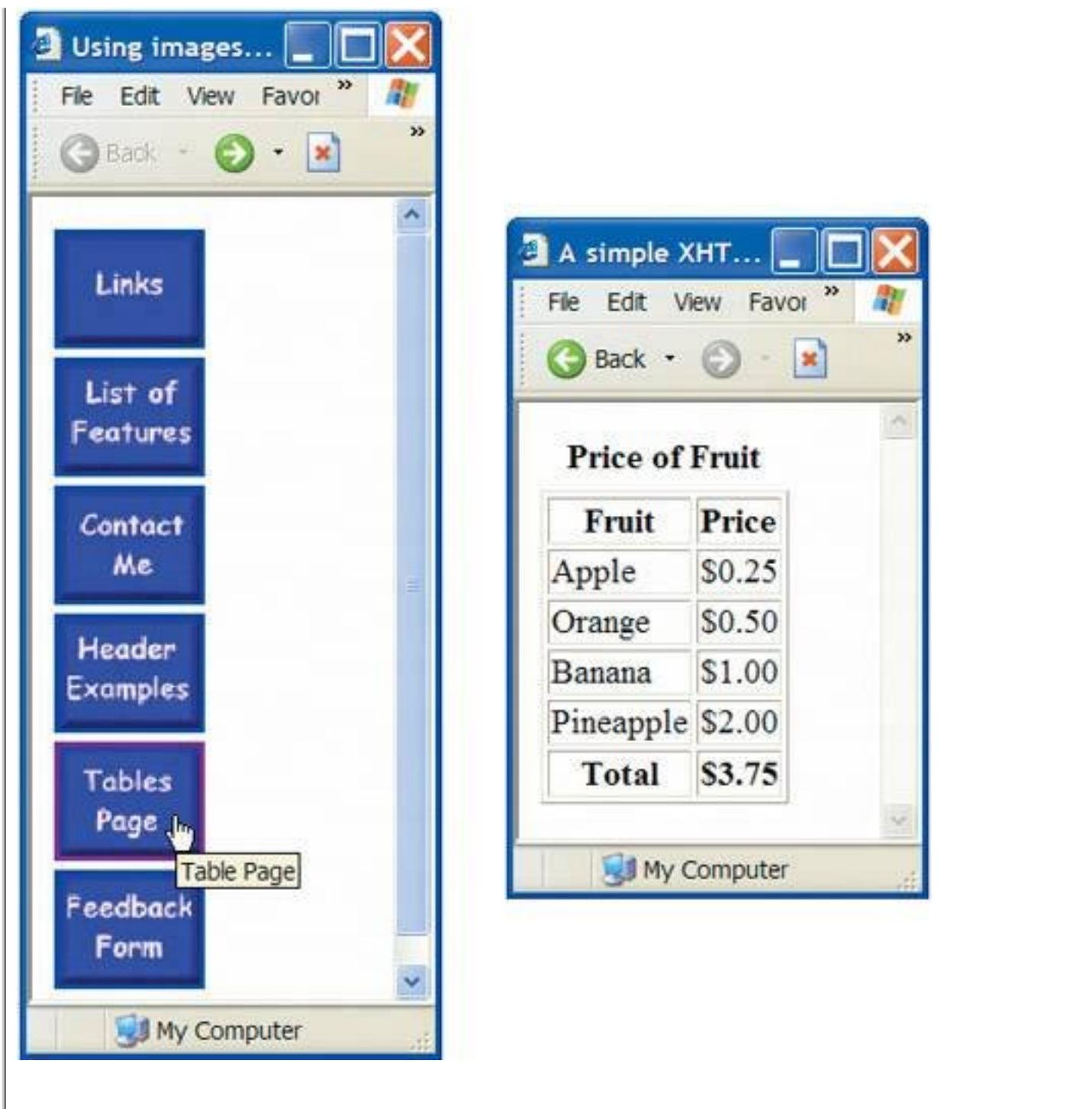
```
26 <a href =<.../span> "contact.html">
27 <img src =<.../span> "buttons/contact.jpg" width = "65"
28 height = "50" alt = "Contact Page" />
29

30
31 <a href =<.../span> "header.html">
32 <img src =<.../span> "buttons/header.jpg" width = "65"
33 height = "50" alt = "Header Page" />
34

35
36 <a href =<.../span> "table.html">
37 <img src =<.../span> "buttons/table.jpg" width = "65"
38 height = "50" alt = "Table Page" />
39

40
41 <a href =<.../span> "form.html">
42 <img src =<.../span> "buttons/form.jpg" width = "65"
43 height = "50" alt = "Feedback Form" />
44

45 </p>
46
47 </body>
48 </html>
```



Lines 1619 create an **image hyperlink** by nesting an `img` element within an anchor (`a`) element. The value of the `img` element's `src` attribute value specifies that this image (`links.jpg`) resides in a directory named `buttons`. The `buttons` directory and the XHTML document are in the same directory. Images from other Web documents also can be referenced (after obtaining permission from the document's owner) by setting the `src` attribute to the name and location of the image.

---

[Page 1337]

In line 19, we introduce the `br` element, which most browsers render as a line break. Any markup or text

following a `br` element is rendered on the next line. Like the `img` element, `br` is an example of an empty element terminated with a forward slash. We add a space before the forward slash to enhance readability.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1339]

In addition to special characters, this document introduces a **horizontal rule**, indicated by the `<hr />` tag in line 24. Most browsers render a horizontal rule as a horizontal line. The `<hr />` tag also inserts a line break above and below the horizontal line.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

[Page 1340]

## J.8. Unordered Lists

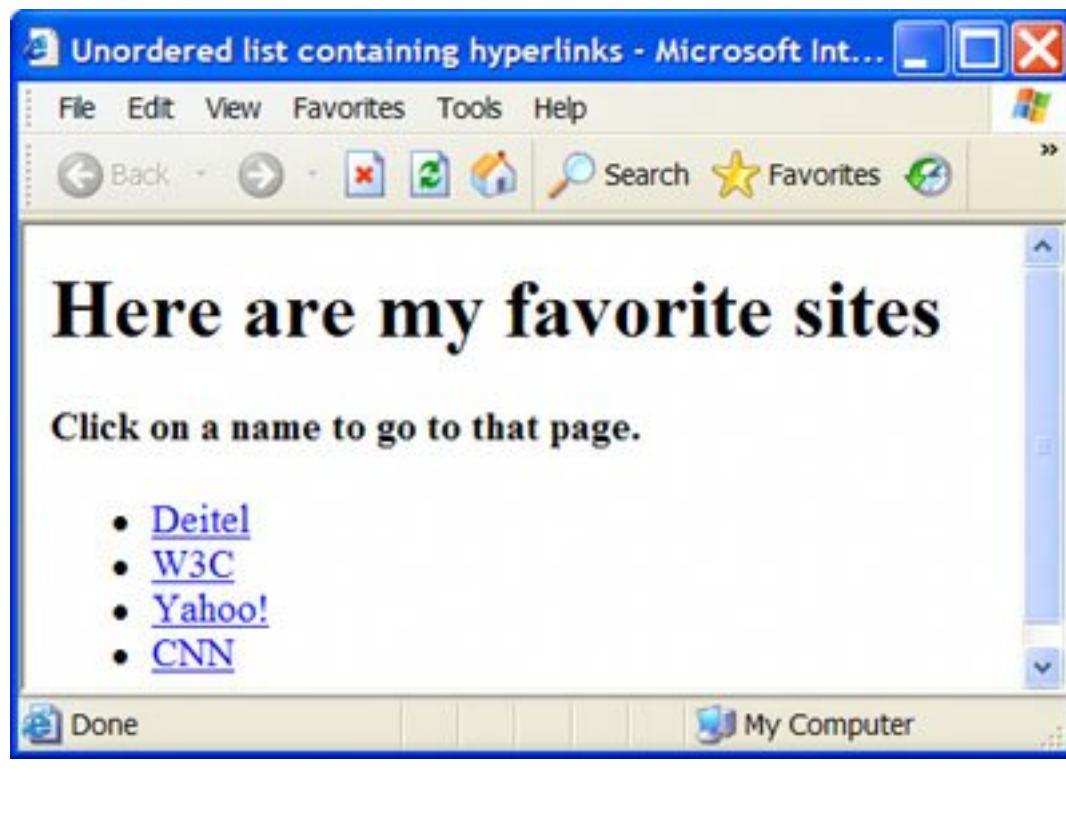
Up to this point, we have presented basic XHTML elements and attributes for linking to resources, creating headers, using special characters and incorporating images. In this section, we discuss how to organize information on a Web page using lists. Later in the appendix, we introduce another feature for organizing information, called a table. [Figure J.8](#) displays text in an unordered list (i.e., a list that does not order its items by letter or number). The [unordered list element `ul`](#) creates a list in which each item begins with a bullet (called a [disc](#)).

**Figure J.8. Unordered lists in XHTML.**

(This item is displayed on pages 1340 - 1341 in the print version)

```
1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Fig. J.8: links2.html -->
6 <!-- Unordered list containing hyperlinks. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Unordered list containing hyperlinks</title>
11 </head>
12
13 <body>
14
15 <h1>Here are my favorite sites</h1>
16
17 <p>Click on a name to go to that page.</p>
18
19 <!-- create an unordered list -->
20
21
22 <!-- add four list items -->
23 <a href =<.../span> "http://www.deitel.com">Deitel
24
```

```
25 <a href =<.../span> "http://www.w3.org">W3C
26
27 <a href =<.../span> "http://www.yahoo.com">Yahoo!
28
29 <a href =<.../span> "http://www.cnn.com">CNN
30
31
32
33 </body>
34 </html>
```



Each entry in an unordered list (element `ul` in line 20) is an `li` (list item) element (lines 23, 25, 27 and 29). Most Web browsers render these elements with a line break and a bullet symbol indented from the beginning of the new line.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1341]

The first ordered list begins in line 33. Attribute `type` specifies the `sequence type` (i.e., the set of numbers or letters used in the ordered list). In this case, setting `type` to "I" specifies upper-case roman numerals. Line 47 begins the second ordered list and sets attribute `type` to "a", specifying lowercase letters for the list items. The last ordered list (lines 7175) does not use attribute `type`. By default, the list's items are enumerated from one to three.

A Web browser indents each nested list to indicate a hierachal relationship. By default, the items in the outermost unordered list (line 18) are preceded by discs. List items nested inside the unordered list of line 18 are preceded by circles. Although not demonstrated in this example, subsequent nested list items are preceded by squares. Unordered list items can be explicitly set to discs, circles or squares by setting the `ul` element's `type` attribute to "disc", "circle" or "square", respectively.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

---

[Page 1343]

Tables are defined with the **table** element. Lines 1618 specify the start tag for a table element that has several attributes. The **border** attribute specifies the table's border width in pixels. To create a table without a border, set **border** to "0". This example assigns attribute **width** "40%", to set the table's width to 40 percent of the browser's width. A developer can also set attribute **width** to a specified number of pixels.

---

[Page 1344]

As its name implies, attribute **summary** (line 17) describes the table's contents. Speech devices use this attribute to make the table more accessible to users with visual impairments. The **caption** element (line 22) describes the table's content and helps text-based browsers interpret the table data. Text inside the **<caption>** tag is rendered above the table by most browsers. Attribute **summary** and element **caption** are two of many XHTML features that make Web pages more accessible to users with disabilities.

---

[Page 1345]

#### Error-Prevention Tip J.1



Try resizing the browser window to see how the width of the window affects the width of the table.

A table has three distinct sections **head**, **body** and **foot**. The head section (or **header cell**) is defined with a **thead** element (lines 2631), which contains header information, such as column names. Each **tr** element (lines 2730) defines an individual **table row**. The columns in the head section are defined with **th** elements. Most browsers center text formatted by **th** (table header column) elements and display it in bold. Table header elements are nested inside table row elements.

---

[Page 1346]

The body section, or **table body**, contains the table's primary data. The table body (lines 3454) is defined in a **tbody** element. **Data cells** contain individual pieces of data and are defined with **td** (**table data**) elements.

The foot section (lines 5863) is defined with a `tfoot` (**table foot**) element and represents a footer. Text commonly placed in the footer includes calculation results and footnotes. Like other sections, the foot section can contain table rows and each row can contain columns.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1348]

Table cells are sized to fit the data they contain. Document authors can create larger data cells by using attributes `rowspan` and `colspan`. The values assigned to these attributes specify the number of rows or columns occupied by a cell. The `th` element at lines 3639 uses the attribute `rowspan = "2"` to allow the cell containing the picture of the camel to use two vertically adjacent cells (thus the cell spans two rows). The `th` element at lines 4245 uses the attribute `colspan = "4"` to widen the header cell (containing Camelid comparison and Approximate as of 9/2002) to span four cells.

Line 42 introduces attribute `valign`, which aligns data vertically and may be assigned one of four values "top" aligns data with the top of the cell, "middle" vertically centers data (the default for all data and header cells), "bottom" aligns data with the bottom of the cell and "baseline" ignores the fonts used for the row data and sets the bottom of all text in the row on a common `baseline` (i.e., the horizontal line to which each character in a word is aligned).

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1351]

Forms can contain visual and non-visual components. Visual components include clickable buttons and other graphical user interface components with which users interact. Non-visual components, called **hidden inputs**, store any data that the document author specifies, such as e-mail addresses and XHTML document file names that act as links. The form begins at line 23 with the `form` element. Attribute `method` specifies how the form's data is sent to the Web server.

Using `method = "post"` appends form data to the browser request, which contains the protocol (i.e., HTTP) and the requested resource's URL. Scripts located on the Web server's computer (or on a computer accessible through the network) can access the form data sent as part of the request. For example, a script may take the form information and update an electronic mailing list. The other possible value, `method = "get"`, appends the form data directly to the end of the URL. For example, the URL `/cgi-bin/formmail` might have the form information `name = bob` appended to it.

The `action` attribute in the `<form>` tag specifies the URL of a script on the Web server; in this case, it specifies a script that e-mails form data to an address. Most Internet Service Providers (ISPs) have a script like this on their site; ask the Web-site system administrator how to set up an XHTML document to use the script correctly.

Lines 2936 define three `input` elements that specify data to provide to the script that processes the form (also called the **form handler**). These three `input` elements have `type` attribute "hidden", which allows the document author to send form data that is not entered by a user to a script.

The three hidden inputs are an e-mail address to which the data will be sent, the e-mail's subject line and a URL where the browser will be redirected after submitting the form. Two other `input` attributes are `name`, which identifies the `input` element, and `value`, which provides the value that will be sent (or posted) to the Web server.

## Good Programming Practice J.6



Place hidden `input` elements at the beginning of a form, immediately after the opening `<form>` tag. This placement allows document authors to locate hidden input elements quickly.

We introduce another type of `input` in lines 3839. The "text" `input` inserts a **text box** into the form. Users can type data in text boxes. The `label` element (lines 3740) provides users with information about the `input` element's purpose.

## Common Programming Error J.6



Forgetting to include a `label` element for each form element is a design error. Without these labels, users cannot determine the purpose of individual form elements.

The `input` element's `size` attribute specifies the number of characters visible in the text box. Optional attribute `maxlength` limits the number of characters input into the text box. In this case, the user is not permitted to type more than 30 characters into the text box.

There are two types of `input` elements in lines 5256. The "submit" `input` element is a button. When the user presses a "submit" button, the browser sends the data in the form to the Web server for processing. The `value` attribute sets the text displayed on the button (the default value is Submit). The "reset" `input` element allows a user to reset all `form` elements to their default values. The `value` attribute of the "reset" `input` element sets the text displayed on the button (the default value is Reset).

[PREV](#)[NEXT](#)

---

**page footer**

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1355]

## Common Programming Error J.7



When your `form` has several checkboxes with the same name, you must make sure that they have different values, or the scripts running on the Web server will not be able to distinguish between them.

## Common Programming Error J.8



When using a group of radio buttons in a form, forgetting to set the `name` attributes to the same name is a logic error that lets the user select all of the radio buttons at the same time.

The `select` element (lines 123136) provides a drop-down list from which the user can select an item. The `name` attribute identifies the drop-down list. The `option` element (lines 124135) adds items to the drop-down list. The `option` element's `selected` attribute specifies which item initially is displayed as the selected item in the `select` element.



[PREV](#)

[page footer](#)



The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1359]

## J.14. Internet and World Wide Web Resources

[www.w3.org/TR/xhtml11](http://www.w3.org/TR/xhtml11)

The XHTML 1.1 Recommendation contains general information, information on compatibility issues, document type definition information, definitions, terminology and much more relating to XHTML.

[www.xhtml.org](http://www.xhtml.org)

XHTML.org provides XHTML development news and links to other XHTML resources, which include books and articles.

[www.w3schools.com/xhtml/default.asp](http://www.w3schools.com/xhtml/default.asp)

The XHTML School provides XHTML quizzes and references. This page also contains links to XHTML syntax, validation and document type definitions.

[hotwired.lycos.com/webmonkey/00/50/index2a.html](http://hotwired.lycos.com/webmonkey/00/50/index2a.html)

This site provides an article about XHTML. Key sections of the article overview XHTML and discuss tags, attributes and anchors.

[wdvl.com/Authoring/Languages/XML/XHTML](http://wdvl.com/Authoring/Languages/XML/XHTML)

The Web Developers' Virtual Library provides an introduction to XHTML. This site also contains articles, examples and links to other technologies.

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1360]

- Web browsers typically underline text hyperlinks and color them blue by default.
- The `<strong>` tag usually causes a browser to render text in a bold font.
- Users can insert links with the `a` (anchor) element. The most important attribute for the `a` element is `href`, which specifies the resource (e.g., page, file, e-mail address) being linked.
- Anchors can link to an e-mail address using a `mailto:` URL. When someone clicks this type of anchored link, most browsers launch the default e-mail program (e.g., Outlook Express) to initiate e-mail messages to the linked addresses.
- The `img` element's `src` attribute specifies an image's location. Optional attributes `width` and `height` specify the image width and height, respectively. Images are measured in pixels ("picture elements"), which represent dots of color on the screen.
- The `alt` attribute makes Web pages more accessible to users with disabilities, especially those with vision impairments.
- Some XHTML elements are empty elements, contain only attributes and do not mark up text. Empty elements (e.g., `img`) must be terminated, either by using the forward slash character (/) or by explicitly writing an end tag.
- The `br` element causes most browsers to render a line break. Any markup or text following a `br` element is rendered on the next line.
- XHTML provides special characters or entity references (in the form `&code;`) for representing characters that cannot be marked up.
- Most browsers render a horizontal rule, indicated by the `<hr />` tag, as a horizontal line. The `HR` element also inserts a line break above and below the horizontal line.
- The unordered list element `ul` creates a list in which each item in the list begins with a bullet symbol (called a disc). Each entry in an unordered list is an `li` (list item) element. Most Web browsers render these elements with a line break and a bullet symbol at the beginning of the line.
- Lists may be nested to represent hierarchical data relationships.
- Attribute `type` specifies the sequence type (i.e., the set of numbers or letters used in the ordered list).
- XHTML tables mark up tabular data and are one of the most frequently used features in XHTML.
- The `table` element defines an XHTML table. Attribute `border` specifies the table's border width, in pixels. Tables without borders set this attribute to "0".
- Element `summary` summarizes the table's contents and is used by speech devices to make the table more accessible to users with visual impairments.
- Element `caption` describes the table's content. The text inside the `<caption>` tag is rendered above the table in most browsers.
- A table can be split into three distinct sections: `head` (`thead`), `body` (`tbody`) and `foot` (`tfoot`). The `head` section contains information such as table titles and column headers. The `table body` contains the primary table data. The `table foot` contains information such as footnotes.
- Element `TR`, or `table row`, defines individual table rows. Element `th` defines a header cell. Text in `th` elements usually is centered and displayed in bold by most browsers. This element can be present in any section of the table.
- Data within a row are defined with `TD`, or `table data`, elements.
- Element `colgroup` groups and formats columns. Each `col` element can format any number of columns (specified with the `span` attribute).

- The document author has the ability to merge data cells with the `rowspan` and `colspan` attributes. The values assigned to these attributes specify the number of rows or columns occupied by the cell. These attributes can be placed inside any data-cell tag.

[Page 1361]

- XHTML provides forms for collecting information from users. Forms contain visual components, such as buttons that users click. Forms may also contain non-visual components, called hidden inputs, which are used to store any data, such as e-mail addresses and XHTML document file names used for linking.
- A form begins with the `form` element. Attribute `method` specifies how the form's data is sent to the Web server.
- The "text" input inserts a text box into the form. Text boxes allow the user to input data.
- The `input` element's `size` attribute specifies the number of characters visible in the `input` element. Optional attribute `maxlength` limits the number of characters input into a text box.
- The "submit" input submits the data entered in the form to the Web server for processing. Most Web browsers create a button that submits the form data when clicked. The "reset" input allows a user to reset all `form` elements to their default values.
- The `textarea` element inserts a multiline text box, called a text area, into a form. The number of rows in the text area is specified with the `rows` attribute and the number of columns (i.e., characters) is specified with the `cols` attribute.
- The "password" input inserts a password box into a form. A password box allows users to enter sensitive information, such as credit-card numbers and passwords, by "masking" the information input with another character. Asterisks are the masking character used for password boxes. The actual value input is sent to the Web server, not the asterisks that mask the input.
- The `checkbox` input allows the user to make a selection. When the checkbox is selected, a check mark appears in the checkbox. Otherwise, the checkbox is empty. Checkboxes can be used individually and in groups. Checkboxes that are part of the same group have the same name.
- A radio button is similar in function and use to a checkbox, except that only one radio button in a group can be selected at any time. All radio buttons in a group have the same `name` attribute value and have different attribute values.
- The `select` input provides a drop-down list of items. The `name` attribute identifies the drop-down list. The `option` element adds items to the drop-down list. The `selected` attribute, like the `checked` attribute for radio buttons and checkboxes, specifies which list item is displayed initially.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1362]

<hr /> tag (horizontal rule)

href attribute

.htm (XHTML file-name extension)

.html (XHTML file-name extension)

<html> tag

hyperlink

image hyperlink

img element

input element

level of nesting

<li> (list item) tag

linked document

mailto: URL

markup language

maxlength attribute

method attribute

name attribute

nested list

nested tag

ol (ordered list) element

p (paragraph) element

password box

"radio" (attribute value)

rows attribute (textarea)

rowspan attribute (tr)

selected attribute

size attribute (input)

special character

src attribute (img)

type attribute

<strong> tag

sub element

subscript

superscript

syntax

table element

tag

Terminology

`tbody` element

`td` element

text editor

`textarea`

`textarea` element

`tfoot` (table foot) element

`<thead>...</thead>`

`title` element

`TR` (table row) element

unordered-list element (`ul`)

`valign` attribute (`th`)

`value` attribute

Web page

Web server

`width` attribute

World Wide Web (WWW)

XHTML (Extensible Hypertext Markup Language)

XHTML comment

XHTML form

XHTML markup

## XHTML tag

XML declaration

`xmlns` attribute

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)
[NEXT ▶](#)

[Page 1363]

## Appendix K. XHTML Special Characters

The table of Fig. K.1 shows many commonly used XHTML special characters called character entity references by the World Wide Web Consortium. For a complete list of character entity references, see the site

[www.w3.org/TR/REC-html40/sgml/entities.html](http://www.w3.org/TR/REC-html40/sgml/entities.html)

**Figure K.1. XHTML special characters.**

Character	XHTML encoding	Character	XHTML encoding
non-breaking space	&#160;	ê	&#234;
§	&#167;	ì	&#236;
©	&#169;	í	&#237;
®	&#174;	î	&#238;
π	&#188;	ñ	&#241;
ƒ	&#189;	ò	&#242;
Ω	&#190;	ó	&#243;
à	&#224;	ô	&#244;
á	&#225;	õ	&#245;
â	&#226;	÷	&#247;
ã	&#227;	ù	&#249;
å	&#229;	ú	&#250;
ç	&#231;	û	&#251;

è	&#232;	•	&#8226;
é	&#233;	™	&#8482;

 PREVNEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1365]

## Outline

[L.1 Introduction](#)

[L.2 Breakpoints and the Continue Command](#)

[L.3 The Locals and Watch Windows](#)

[L.4 Controlling Execution Using the Step Into, Step Over, Step Out and Continue Commands](#)

[L.5 The Autos Window](#)

[L.6 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1365 (continued)]

## L.1. Introduction

In Chapter 2, you learned that there are two types of errorscompilation errors and logic errorsand you learned how to eliminate compilation errors from your code. Logic errors (also called **bugs**) do not prevent a program from compiling successfully, but do cause the program to produce erroneous results when it runs. Most C++ compiler vendors provide software called a **debugger**, which allows you to monitor the execution of your programs to locate and remove logic errors. The debugger will be one of your most important program development tools. This appendix demonstrates key features of the Visual Studio .NET debugger. Chapter M discusses the features and capabilities of the GNU C++ debugger. We provide several free Dive Into™ Series publications to help students and instructors familiarize themselves with the debuggers provided with various other development tools. These publications are available on the CD that accompanies the text and can be downloaded from [www.deitel.com/books/downloads](http://www.deitel.com/books/downloads).

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1366]

**Figure L.1. Header file for the Account class.**

```

1 // Fig. L.1: Account.h
2 // Definition of Account class.
3
4 class Account
5 {
6 public:
7 Account(int); // constructor initializes balance
8 void credit(int); // add an amount to the account balance
9 void debit(int); // subtract an amount from the account balance
10 int getBalance(); // return the account balance
11 private:
12 int balance; // data member that stores the balance
13 } // end class Account

```

**Figure L.2. Definition for the Account class.**

(This item is displayed on pages 1366 - 1367 in the print version)

```

1 // Fig. L.2: Account.cpp
2 // Member-function definitions for class Account.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Account.h" // include definition of class Account
8
9 // Account constructor initializes data member balance
10 Account::Account(int initialBalance)
11 {
12 balance = 0; // assume that the balance begins at 0
13
14 // if initialBalance is greater than 0, set this value as the
15 // balance of the Account; otherwise, balance remains 0
16 if (initialBalance > 0)
17 balance = initialBalance;

```

```
18
19 // if initialBalance is negative, print error message
20 if (initialBalance < 0)
21 cout << "Error: Initial balance cannot be negative.\n" << endl;
22 } // end Account constructor
23
24 // credit (add) an amount to the account balance
25 void Account::credit(int amount)
26 {
27 balance = balance + amount; // add amount to balance
28 } // end function credit
29
30 // debit (subtract) an amount from the account balance
31 void Account::debit(int amount)
32 {
33 if (amount <= balance) // debit amount does not exceed balance
34 balance = balance - amount;
35
36 else // debit amount exceeds balance
37 cout << "Debit amount exceeded account balance.\n" << endl;
38 } // end function debit
39
40 // return the account balance
41 int Account::getBalance()
42 {
43 return balance; // gives the value of balance to the calling function
44 } // end function getBalance
```

---

[Page 1367]

**Figure L.3. Test class for debugging.**

```
1 // Fig. L.3: figL_03.cpp
2 // Create and manipulate Account objects.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 // include definition of class Account from Account.h
9 #include "Account.h"
10
11 // function main begins program execution
12 int main()
13 {
14 Account account1(50); // create Account object
15
16 // display initial balance of each object
17 cout << "account1 balance: $" << account1.getBalance() << endl;
18
19 int withdrawalAmount; // stores withdrawal amount read from user
20
21 cout << "\nEnter withdrawal amount for account1: " // prompt
22 cin >> withdrawalAmount; // obtain user input
23 cout << "\nAttempting to subtract " << withdrawalAmount
24 << " from account1 balance\n\n";
25 account1.debit(withdrawalAmount); // try to subtract from account1
26
27 // display balances
28 cout << "account1 balance: $" << account1.getBalance() << endl;
29 return 0; // indicate successful termination
30 } // end main
```

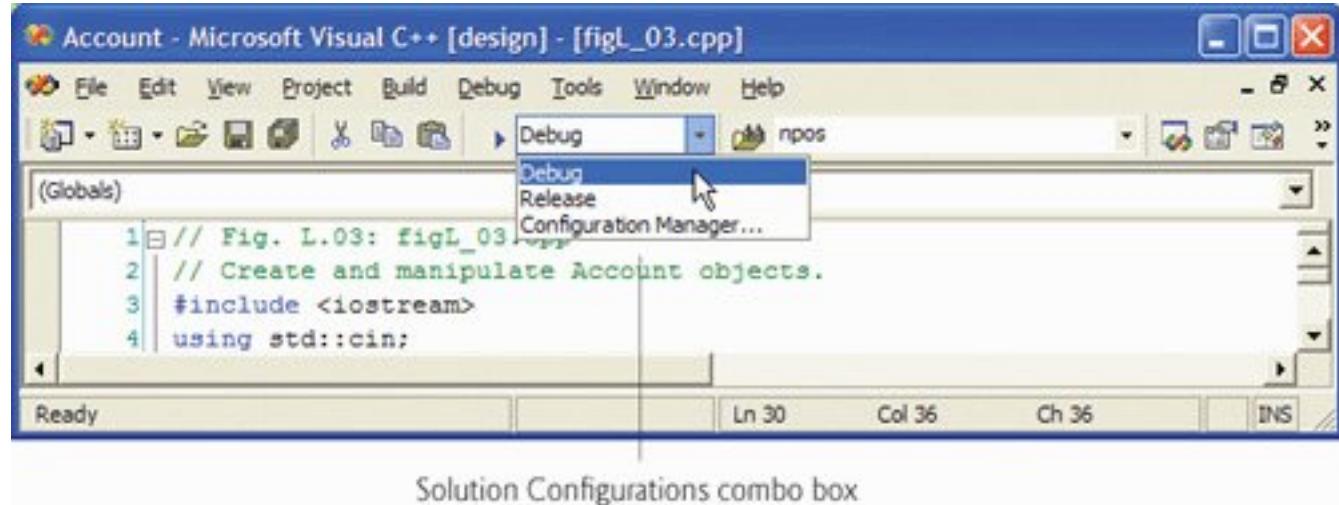
In the following steps, you will use breakpoints and various debugger commands to examine the value of the variable `withdrawalAmount` declared in [Fig. L.3](#).

- Enabling the debugger. The debugger is enabled by default. If it is not enabled, you have to change the settings of the **Solution Configurations combo box** (Fig. L.4) in the toolbar. To do this, click the combo box's down arrow to access the Solution Configurations combo box, then select Debug. The toolbar will display Debug in the Solution Configurations combo box.

[Page 1368]

**Figure L.4. Enabling the debugger.**

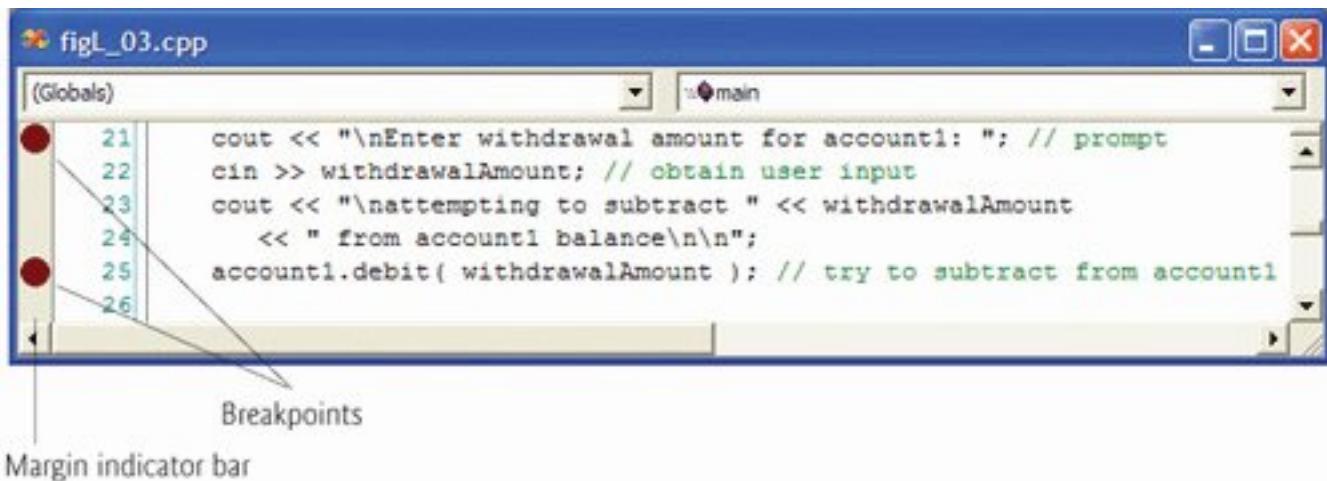
[View full size image]



- Inserting breakpoints in Visual Studio .NET. To insert a breakpoint in Visual Studio .NET, click inside the **margin indicator bar** (the gray margin at the left of the code window in Fig. L.5) next to the line of code at which you wish to break or right click that line of code and select Insert Breakpoint. You can set as many breakpoints as necessary. Set breakpoints at lines 21 and 25 of your code. A solid maroon circle appears in the margin indicator bar where you clicked, indicating that a breakpoint has been set (Fig. L.5). When the program runs, the debugger suspends execution at any line that contains a breakpoint. The program is said to be in **break mode** when the debugger pauses the program's execution. Breakpoints can be set before running a program, in break mode and while a program is running.

**Figure L.5. Setting two breakpoints.**

[View full size image]



- Beginning the debugging process. After setting breakpoints in the code editor, select Build > Build Solution to compile the program, then select Debug > Start to begin the debugging process. During debugging of a C++ program, a Command Prompt window appears (Fig. L.6), allowing program interaction (input and output). The program pauses when execution reaches the breakpoint at line 21. At this point, the title bar of the IDE will display [break] (Fig. L.7), indicating that the IDE is in break mode.

[Page 1369]

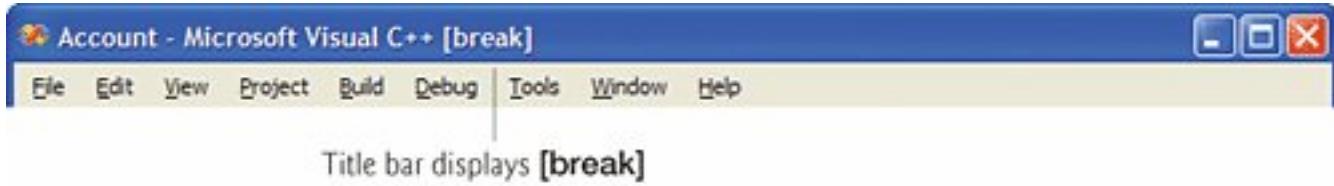
**Figure L.6. Inventory program running.**

[\[View full size image\]](#)



**Figure L.7. Title bar of the IDE displaying [break].**

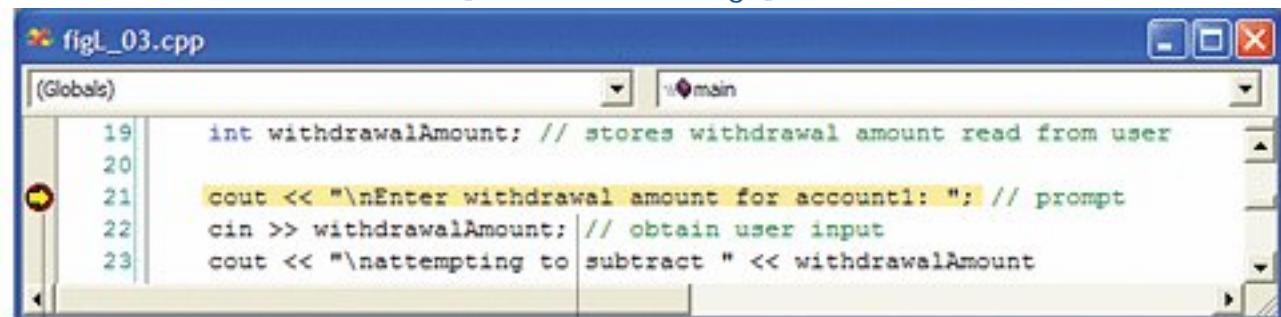
[\[View full size image\]](#)



4. Examining program execution. Program execution suspends at the first breakpoint (line 21), and the IDE becomes the active window (Fig. L.8). The **yellow arrow** to the left of line 21 indicates that this line contains the next statement to execute. [Note: We have added the yellow highlighting to these images. Your code will not contain this highlighting.]

**Figure L.8. Program execution suspended at the first breakpoint.**

[View full size image]



Yellow arrow                          Next executable statement

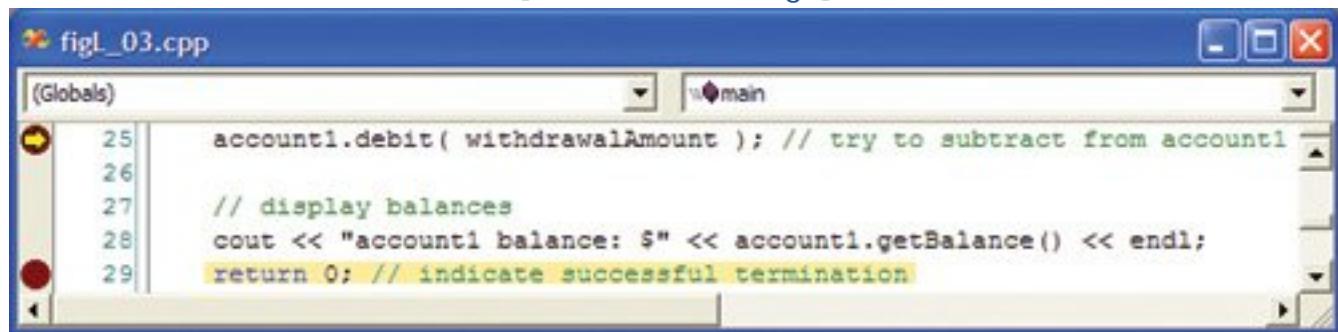
```
figL_03.cpp
(Globals) ┌──────────┐ ┌─────────┐
19 | int withdrawalAmount; // stores withdrawal amount read from user
20 |
21 | cout << "\nEnter withdrawal amount for account1: "; // prompt
22 | cin >> withdrawalAmount; // obtain user input
23 | cout << "\nAttempting to subtract " << withdrawalAmount
```

5. Using the Continue command to resume execution. To resume execution, select Debug > Continue. The **Continue command** will execute any statements between the next executable statement and the next breakpoint or the end of `main`, whichever comes first. The program continues executing and pauses for input at line 22. Input 13 as the withdrawal amount. The program executes until it stops at the next breakpoint, line 25. Notice that when you place your mouse pointer over the variable name `withdrawalAmount`, the value that the variable stores is displayed in a **Quick Info box** (Fig. L.9). In a sense, you are peeking inside the computer at the value of one of your variables. As you'll see, this can help you spot logic errors in your programs.

**Figure L.9. Setting a breakpoint at line 29.**

(This item is displayed on page 1370 in the print version)

[View full size image]



```
figL_03.cpp
(Globals) ┌──────────┐ ┌─────────┐
25 | account1.debit(withdrawalAmount); // try to subtract from account1
26 |
27 | // display balances
28 | cout << "account1 balance: $" << account1.getBalance() << endl;
29 | return 0; // indicate successful termination
```

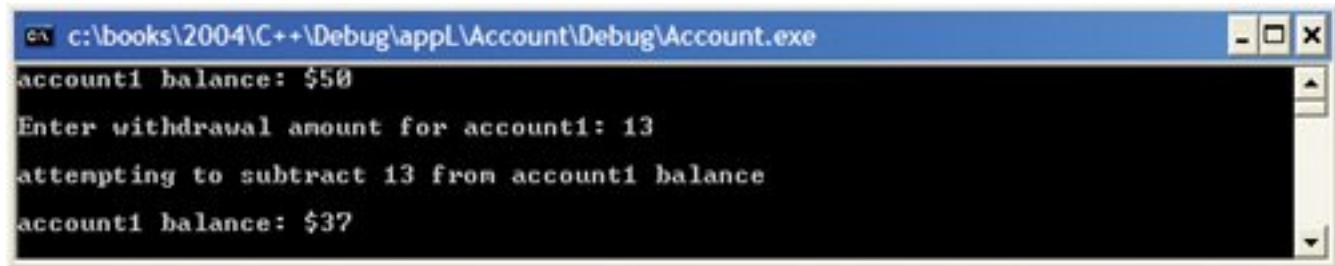
6. Setting a breakpoint at the `return` statement. Set a breakpoint at line 29 in the source code by clicking in the margin indicator bar to the left of line 29 (Fig. L.9). This will prevent the program from closing immediately after displaying its result. When there are no more breakpoints at which to suspend execution, the program will execute to completion and the Command Prompt window will close. If you do not set this breakpoint, you will not be able to view the program's output before the console window closes.

[Page 1370]

7. Continuing program execution. Use the Debug > Continue command to execute line 25. The program displays the result of its calculation (Fig. L.10).

**Figure L.10. Program output.**

[View full size image]

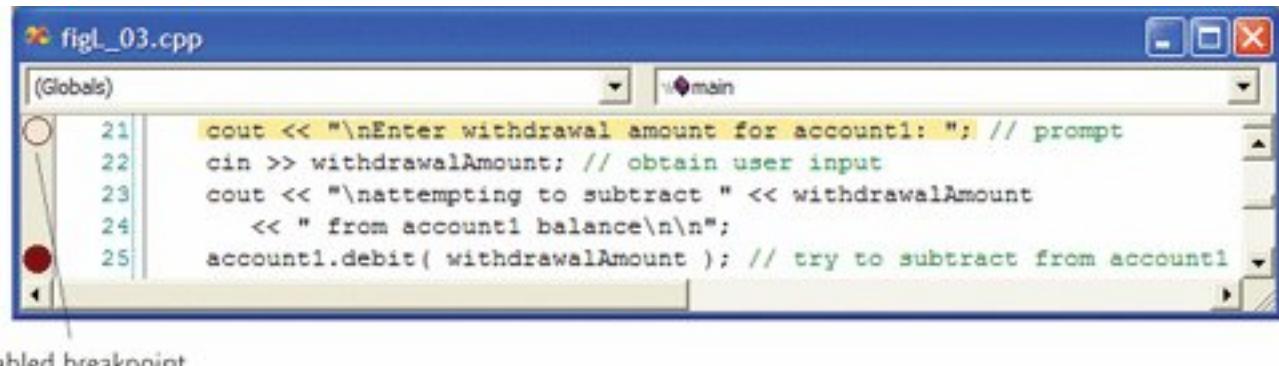


```
c:\books\2004\C++\Debug\appL\Account\Debug\Account.exe
account1 balance: $50
Enter withdrawal amount for account1: 13
attempting to subtract 13 from account1 balance
account1 balance: $37
```

8. **Disabling a breakpoint.** To disable a breakpoint, right click a line of code on which a breakpoint has been set (or the breakpoint itself) and select Disable Breakpoint. The disabled breakpoint is indicated by a hollow maroon circle (Fig. L.11). Disabling rather than removing a breakpoint allows you to re-enable the breakpoint (by clicking inside the hollow circle) in a program. This also can be done by right clicking the line marked by the hollow maroon circle (or the maroon circle itself) and selecting Enable Breakpoint.

**Figure L.11. Disabled breakpoint.**

[View full size image]



figl\_03.cpp

```
(Globals) ┌─────────┐ ┌─────────┐
 | | ┌──┐
 | | | main
 | | | ┌─────────┐
 | | | | 21 | cout << "\nEnter withdrawal amount for account1: " // prompt
 | | | | 22 | cin >> withdrawalAmount; // obtain user input
 | | | | 23 | cout << "\nattempting to subtract " << withdrawalAmount
 | | | | 24 | << " from account1 balance\n\n";
 | | | | 25 | account1.debit(withdrawalAmount); // try to subtract from account1
```

Disabled breakpoint

9. Removing a breakpoint. To remove a breakpoint that you no longer need, right click a line of code on which a breakpoint has been set and select Remove Breakpoint. You also can remove a breakpoint by clicking the maroon circle in the margin indicator bar.
10. Finishing program execution. Select Debug > Continue to execute the program to completion.

---

[Page 1371]

In this section, you learned how to enable the debugger and set breakpoints so that you can examine the results of code while a program is running. You also learned how to continue execution after a program suspends execution at a breakpoint and how to disable and remove breakpoints.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1372]

**Figure L.15. Examining variable withdrawalAmount.**

Name	Value	Type
account1	{balance=50}	Account
withdrawalAmount	13	int

5.

Evaluating arithmetic and boolean expressions. Visual Studio .NET allows you to evaluate arithmetic and boolean expressions using the Watch window. There are four different Watch windows, but we will be using only the first window. Select Debug > Windows > Watch > Watch 1. In the first row of the Name column (which should be blank initially), type `(withdrawalAmount + 3) * 5`, then press Enter. Notice that the Watch window can evaluate arithmetic expressions. In this case, it displays the value 80 (Fig. L.16). In the next row of the Name column in the Watch window, type `withdrawalAmount == 3`, then press Enter. This expression determines whether the value contained in `withdrawalAmount` is 3. Expressions containing the `==` symbol are treated as boolean expressions. The value returned is `false` (Fig. L.16), because `withdrawalAmount` does not currently contain the value 3.

**Figure L.16. Examining the values of expressions.**

[\[View full size image\]](#)

Evaluating an arithmetic expression

Evaluating a bool expression

Name	Value	Type
<code>(withdrawalAmount + 3) * 5</code>	80	int
<code>withdrawalAmount == 3</code>	false	bool

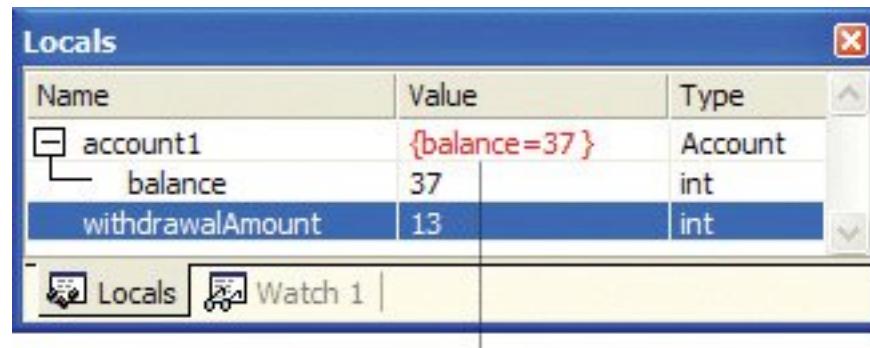
[Page 1373]

6.

Resuming execution. Select Debug > Continue to resume execution. Line 25 executes, debiting the account with the withdrawal amount, and the program is once again suspended at line 28. Select Debug > Windows > Locals. The updated `account1` value is now displayed in red to indicate that it has been modified since the last breakpoint (Fig. L.17). The value in `withdrawalAmount` is not in red because it

has not been updated since the last breakpoint. Click the plus box to the left of account1 in the Name column of the Locals window. This allows you to view each of account1's data member values individually.

**Figure L.17. Displaying the value of local variables.**



The Locals window displays the following data:

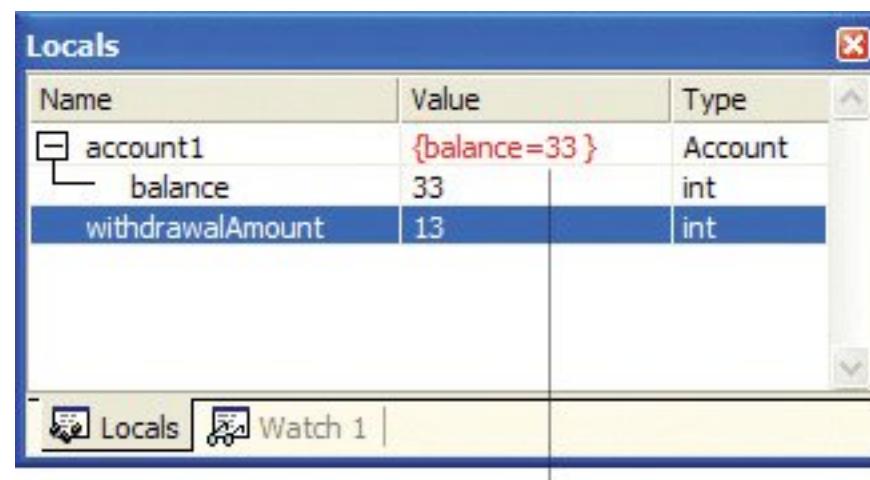
Name	Value	Type
account1	{balance=37}	Account
balance	37	int
withdrawalAmount	13	int

Value of the account1 variable displayed in red

## 7.

Modifying values. Based on the value input by the user (13), the account balance output by the program should be \$37. However, you can use the debugger to change the values of variables in the middle of the program's execution. This can be valuable for experimenting with different values and for locating logic errors in programs. You can use the Locals window to change the value of a variable. In the Locals window, click the Value field in the balance row to select the value 37. Type 33, then press Enter. The debugger changes the value of balance and displays its new value in red (Fig. L.18).

**Figure L.18. Modifying the value of a variable.**



The Locals window displays the following data after modification:

Name	Value	Type
account1	{balance=33}	Account
balance	33	int
withdrawalAmount	13	int

Value modified in the debugger

## 8.

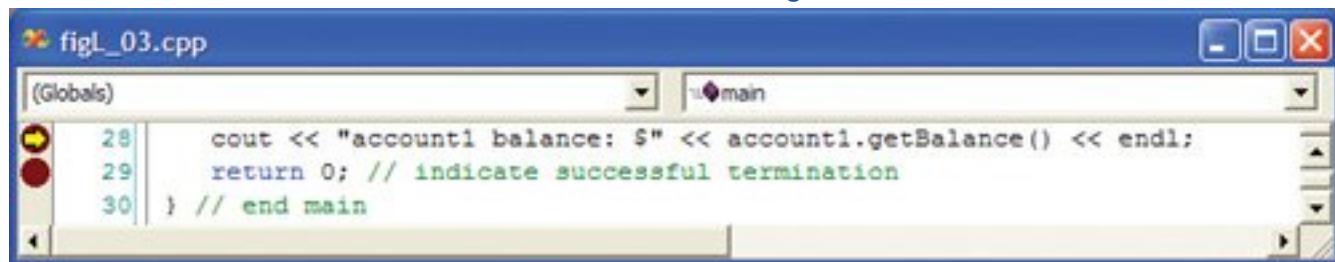
Setting a breakpoint at the `return` statement. Set a breakpoint at line 29 in the source code by clicking

in the margin indicator bar to the left of line 29 (Fig. L.19). This will prevent the program from closing immediately after displaying its result. If you do not set this breakpoint, you will not be able to view the program's output before the console window closes.

**Figure L.19. Setting a breakpoint at line 29.**

(This item is displayed on page 1374 in the print version)

[View full size image]



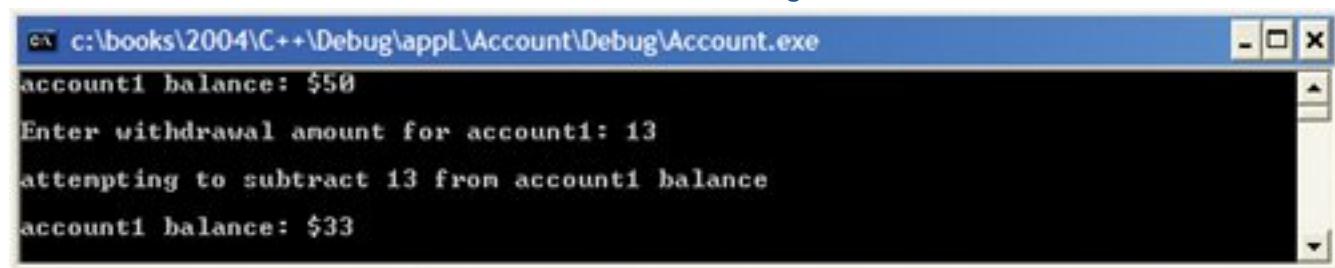
**9.**

Viewing the program result. Select Debug > Continue to continue program execution. Function `main` executes until the `return` statement on line 29 and displays the result. Notice that the result is \$33 (Fig. L.20). This shows that the previous step changed the value of `balance` from the calculated value (37) to 33.

[Page 1374]

**Figure L.20. Output displayed after modifying the account1 variable.**

[View full size image]



**10.**

Stopping the debugging session. Select Debug > Stop Debugging. This will close the Command Prompt window. Remove all remaining breakpoints.

In this section, you learned how to use the debugger's Watch and Locals windows to evaluate arithmetic and boolean expressions. You also learned how to modify the value of a variable during your program's execution.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1375]

**2.**

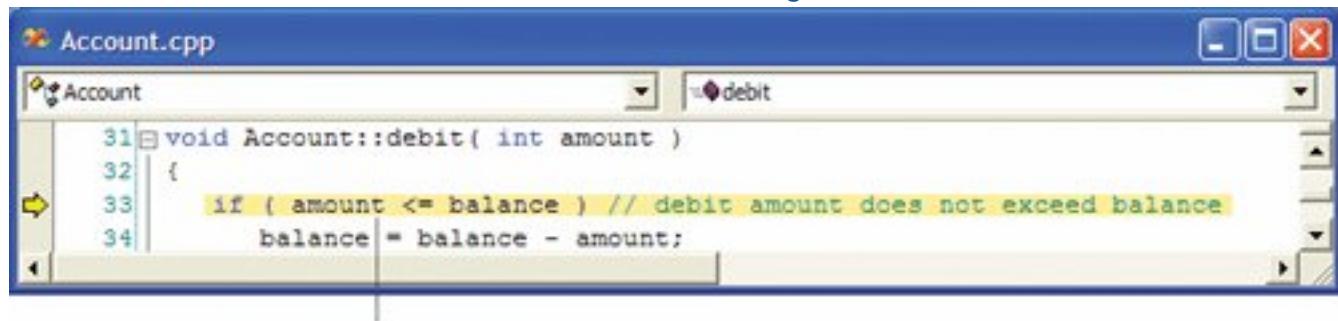
Starting the debugger. Select Debug > Start. Enter the value 13 at the Enter withdrawal amount for account1: prompt. Execution will halt when the program reaches the breakpoint at line 25.

**3.**

Using the Step Into command. The **Step Into command** executes the next statement in the program (the yellow highlighted line of Fig. L.24) and immediately halts. If the statement to be executed as a result of the Step Into command is a function call, control is transferred to the called function. The Step Into command allows you to enter a function and confirm its execution by individually executing each statement inside the function. Select Debug > Step Into to enter the debit function (Fig. L.22). If the debugger is not at line 33, select Debug > Step Into again to reach that line.

**Figure L.22. Stepping into the `debit` function.**

[View full size image]



**4.**

Using the Step Over command. Select Debug > Step Over to execute the current statement (line 33 in Fig. L.22) and transfer control to line 34 (Fig. L.23). The **Step Over command** behaves like the Step Into command when the next statement to execute does not contain a function call. You will see how the Step Over command differs from the Step Into command in Step 10.

**Figure L.23. Stepping over a statement in the `debit` function.**

[View full size image]

```

Account.cpp
Account
33 if (amount <= balance) // debit amount does not exceed balance
34 balance = balance - amount;
35

```

Control is transferred to the next statement

**Figure L.24. Using the Step Into command to execute a statement.**

[View full size image]

```

figL_03.cpp
(Globals)
24 << " from account1 balance\n\n";
25 account1.debit(withdrawalAmount); // try to subtract from account1
26

```

Next statement to execute is a function call

[Page 1376]

## 5.

Using the Step Out command. Select Debug > Step Out to execute the remaining statements in the function and return control to the next executable statement (line 28 in Fig. L.3), which contains the function call. Often, in lengthy functions, you will want to look at a few key lines of code, then continue debugging the caller's code. The **Step Out command** is useful for such situations, where you do not want to continue stepping through the entire function line by line.

## 6.

Setting a breakpoint. Set a breakpoint (Fig. L.25) at the `return` statement of `main` at line 29 of Fig. L.3. You will make use of this breakpoint in the next step.

**Figure L.25. Setting a second breakpoint in the program.**

[View full size image]

```

figL_03.cpp
(Globals)
28 cout << "account1 balance: $" << account1.getBalance() << endl;
29 return 0; // indicate successful termination
30 } // end main

```

**7.**

Using the Continue command. Select Debug > Continue to execute until the next breakpoint is reached at line 29. This feature saves time when you do not want to step line by line through many lines of code to reach the next breakpoint.

**8.**

Stopping the debugger. Select Debug > Stop Debugging to end the debugging session. This will close the Command Prompt window.

**9.**

Starting the debugger. Before we can demonstrate the next debugger feature, you must start the debugger again. Start it, as you did in Step 2, and enter as input the same value (13). The debugger pauses execution at line 25.

**10.**

Using the Step Over command. Select Debug > Step Over (Fig. L.26) Recall that this command behaves like the Step Into command when the next statement to execute does not contain a function call. If the next statement to execute contains a function call, the called function executes in its entirety (without pausing execution at any statement inside the function), and the yellow arrow advances to the next executable line (after the function call) in the current function. In this case, the debugger executes line 25, located in main (Fig. L.3). Line 25 calls the debit function. The debugger then pauses execution at line 28, the next executable line in the current function, main.

**Figure L.26. Using the debugger's Step Over command.**

[View full size image]

```
figL_03.cpp
(Globals) ┌───┐
 └───┘
 25 | account1.debit(withdrawalAmount); // try to subtract from account1
 26 |
 27 | // display balances
 28 | cout << "account1 balance: $" << account1.getBalance() << endl;
 29 | return 0; // indicate successful termination
```

The debit function is executed without stepping into it when the **Step Over** command is selected

[Page 1377]

**11.**

Stopping the debugger. Select Debug > Stop Debugging. This will close the Command Prompt window. Remove all remaining breakpoints.

In this section, you learned how to use the debugger's Step Into command to debug functions called

during your program's execution. You saw how the Step Over command can be used to step over a function call. You used the Step Out command to continue execution until the end of the current function. You also learned that the Continue command continues execution until another breakpoint is found or the program exits.

 PREV

NEXT 

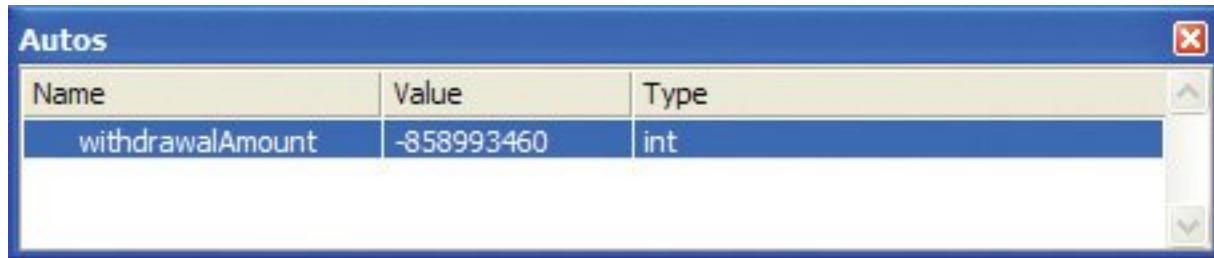
page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1378]

**Figure L.30. Autos window displaying local variable withdrawalAmount.**



**5.**

Entering data. Select Debug > Step Over to execute line 22. At the program's input prompt, enter a value for the withdrawal amount. The Autos window ([Fig. L.29](#)) will update the value of local variable withdrawalAmount with the value you entered. [Note: The first line of the Autos window contains the `istream` object (`cin`) you used to input data.]

**6.**

Stopping the debugger. Select Debug > Stop Debugging to end the debugging session. Remove all remaining breakpoints.

In this section, you learned about the Autos window, which allows you to view the variables used in the most recent command.

[◀ PREV](#)

page footer

[NEXT ▶](#)

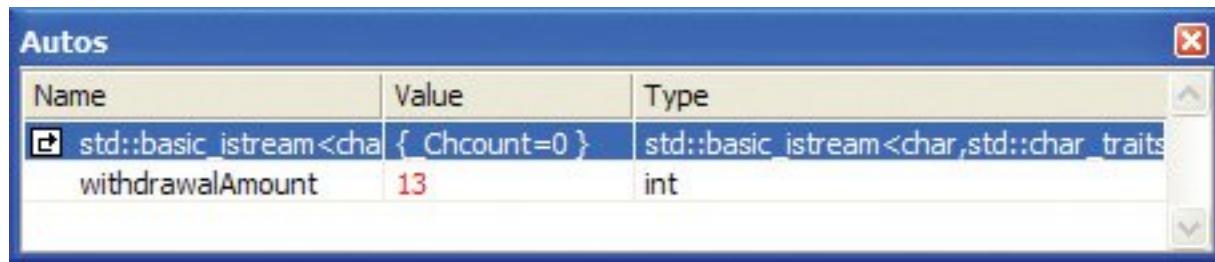
The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1379]

**Figure L.31. Autos window displaying updated local variable withdrawalAmount.**

(This item is displayed on page 1378 in the print version)



The screenshot shows the 'Autos' window from a debugger. It has three columns: 'Name', 'Value', and 'Type'. There are two rows of data. The first row is collapsed, indicated by a small triangle icon to its left. The second row is expanded, showing the variable 'withdrawalAmount' with a value of '13' and a type of 'int'. The 'Value' column also displays the type information for the collapsed row.

Name	Value	Type
std::basic_istream<char> { Chcount=0 }		std::basic_istream<char, std::char_traits<char>, std::ios_base::public_members<char> >
withdrawalAmount	13	int

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1380]

- The Autos window allows you to view the contents of the variables used in the last statement that was executed. The Autos window also lists the values in the next statement to be executed.

 PREV

NEXT 

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1380 (continued)]

## Terminology

Autos window

break mode

breakpoint

bug

Continue command

debugger

disabling a breakpoint

Locals window

margin indicator bar

Quick Info box

Solution Configurations combo box

Step Into command

Step Out command

Step Over command

Watch window

yellow arrow in break mode

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1380 (continued)]

## Self-Review Exercises

**L.1** Fill in the blanks in each of the following statements:

a.

When the debugger suspends program execution at a breakpoint, the program is said to be in \_\_\_\_\_ mode.

b.

The \_\_\_\_\_ feature in Visual Studio .NET allows you to "peek into the computer" and look at the value of a variable.

c.

You can examine the value of an expression by using the debugger's  
\_\_\_\_\_ window.

d.

The \_\_\_\_\_ command behaves like the Step Into command when the next statement to execute does not contain a function call.

**L.2** State whether each of the following is true or false. If false, explain why.

**a.**

When program execution suspends at a breakpoint, the next statement to be executed is the statement after the breakpoint.

**b.**

When a variable's value is changed, it becomes yellow in the Autos and Locals windows.

**c.**

During debugging, the Step Out command executes the remaining statements in the current function and returns program control to the place where the function was called.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1380 (continued)]

## Answers to Self-Review Exercises

**L.1** a) break. b) Quick Info box. c) Watch. d) Step Over.

**L.2** a) False. When program execution suspends at a breakpoint, the next statement to be executed is the statement at the breakpoint. b) False. A variable turns red when its value is changed. c) True.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1382]

## Outline

[M.1 Introduction](#)

[M.2 Breakpoints and the run, stop, continue and print Commands](#)

[M.3 The print and set Commands](#)

[M.4 Controlling Execution Using the step, finish and next Commands](#)

[M.5 The watch Command](#)

[M.6 Wrap-Up](#)

[Summary](#)

[Terminology](#)

[Self-Review Exercises](#)

[Answers to Self-Review Exercises](#)

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page 1382 (continued)]

## M.1. Introduction

In Chapter 2, you learned that there are two types of errorscompilation errors and logic errorsand you learned how to eliminate compilation errors from your code. Logic errors do not prevent a program from compiling successfully, but they do cause the program to produce erroneous results when it runs. GNU includes software called a **debugger** that allows you to monitor the execution of your programs so you can locate and remove logic errors.

The debugger will be one of your most important program development tools. Many IDEs provide their own debuggers similar to the one included in GNU or provide a graphical user interface to GNU's debugger. This appendix demonstrates key features of GNU's debugger. Appendix L discusses the features and capabilities of the Visual Studio .NET debugger. We provide several free Dive Into™ Series publications to help students and instructors familiarize themselves with the debuggers provided with various development tools. These publications are available on the CD that accompanies the text and can be downloaded from [www.deitel.com/books/downloads](http://www.deitel.com/books/downloads).

[◀ PREV](#)[NEXT ▶](#)

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1383]

**Figure M.1. Header file for the Account class.**

```

1 // Fig. M.1: Account.h
2 // Definition of Account class.
3
4 class Account
5 {
6 public:
7 Account(int); // constructor initializes balance
8 void credit(int); // add an amount to the account balance
9 void debit(int); // subtract an amount from the account balance
10 int getBalance(); // return the account balance
11 private:
12 int balance; // data member that stores the balance
13 } // end class Account

```

**Figure M.2. Definition for the Account class.**

(This item is displayed on pages 1383 - 1384 in the print version)

```

1 // Fig. M.2: Account.cpp
2 // Member-function definitions for class Account.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Account.h" // include definition of class Account
8
9 // Account constructor initializes data member balance
10 Account::Account(int initialBalance)
11 {
12 balance = 0; // assume that the balance begins at 0
13
14 // if initialBalance is greater than 0, set this value as the
15 // balance of the Account; otherwise, balance remains 0
16 if (initialBalance > 0)
17 balance = initialBalance;
18
19 // if initialBalance is negative, print error message
20 if (initialBalance < 0)
21 cout << "Error: Initial balance cannot be negative.\n" << endl;
22 } // end Account constructor

```

```

23
24 // credit (add) an amount to the account balance
25 void Account::credit(int amount)
26 {
27 balance = balance + amount; // add amount to balance
28 } // end function credit
29
30 // debit (subtract) an amount from the account balance
31 void Account::debit(int amount)
32 {
33 if (amount <= balance) // debit amount does not exceed balance
34 balance = balance - amount;
35
36 else // debit amount exceeds balance
37 cout << "Debit amount exceeded account balance.\n" << endl;
38 } // end function debit
39
40 // return the account balance
41 int Account::getBalance()
42 {
43 return balance; // gives the value of balance to the calling function
44 } // end function getBalance

```

[Page 1384]

**Figure M.3. Test class for debugging.**

```

1 // Fig. M.3: figM_03.cpp
2 // Create and manipulate Account objects.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 // include definition of class Account from Account.h
9 #include "Account.h"
10
11 // function main begins program execution
12 int main()
13 {
14 Account account1(50); // create Account object
15
16 // display initial balance of each object
17 cout << "account1 balance: $" << account1.getBalance() << endl;
18
19 int withdrawalAmount; // stores withdrawal amount read from user
20
21 cout << "\nEnter withdrawal amount for account1: " // prompt

```

```

22 cin >> withdrawalAmount; // obtain user input
23 cout << "\nAttempting to subtract " << withdrawalAmount
24 << " from account1 balance\n\n";
25 account1.debit(withdrawalAmount); // try to subtract from account1
26
27 // display balances
28 cout << "account1 balance: $" << account1.getBalance() << endl;
29 return 0; // indicate successful termination
30 } // end main

```

In the following steps, you will use breakpoints and various debugger commands to examine the value of the variable `withdrawalAmount` declared in line 19 of Fig. M.3.

1. Compiling the program for debugging. The GNU debugger works only with executable files that were compiled with the `-g` compiler option, which generates information that is used by the debugger to help you debug your programs. Compile the program with the `-g` command-line option by typing `g++ -g -o figM_03 figM_03.cpp Account.cpp`.

---

[Page 1385]

2. Starting the debugger. Type `gdb figM_03` (Fig. M.4). This command will start the GNU debugger and display the (gdb) prompt at which you can enter commands.

**Figure M.4. Starting the debugger to run the program.**

```

~/Debug$ gdb figM_03
GNU gdb 6.1-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db library "
/lib/libthread_db.so.1".

(gdb)

```

3. Running a program in the debugger. Run the program through the debugger by typing [run](#) (Fig. M.5). If you do not set any breakpoints before running your program in the debugger, the program will run to completion.

**Figure M.5. Running the program with no breakpoints set.**

```
(gdb) run
Starting program: /home/student/Debug/figM_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

4. Inserting breakpoints using the GNU debugger. You set a breakpoint at a specific line of code in your program. The line numbers used in these steps are from the source code in Fig. M.3. Set a breakpoint at line 17 in the source code by typing [break 17](#). The [break command](#) inserts a breakpoint at the line number specified after the command. You can set as many breakpoints as necessary. Each breakpoint is identified in terms of the order in which it was created. The first breakpoint created is known as Breakpoint 1. Set another breakpoint at line 25 by typing [break 25](#) (Fig. M.6). When the program runs, it suspends execution at any line that contains a breakpoint and the program is said to be in [break mode](#). Breakpoints can be set even after the debugging process has begun. [Note: If you do not have a numbered listing for your code, you can use the [list](#) command to output your code with line numbers. For more information about the [list](#) command type [help list](#) from the [gdb](#) prompt.]

---

[Page 1386]

**Figure M.6. Setting two breakpoints in the program.**

```
(gdb) break 17
Breakpoint 1 at 0x80487d8: file figM_03.cpp, line 17.
(gdb) break 25
Breakpoint 2 at 0x8048871: file figM_03.cpp, line 25.
(gdb)
```

- Running the program and beginning the debugging process. Type `run` to execute your program and begin the debugging process (Fig. M.7). The program pauses when execution reaches the breakpoint at line 17. At this point, the debugger notifies you that a breakpoint has been reached and displays the source code at that line (17). That line of code contains the next statement that will execute.

**Figure M.7. Running the program until it reaches the first breakpoint.**

```
(gdb) run
Starting program: /home/student/Debug/figM_03

Breakpoint 1, main () at figM_03.cpp:17
17 cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

- Using the `continue` command to resume execution. Type `continue`. The `continue command` causes the program to continue running until the next breakpoint is reached (line 25). Enter `13` at the prompt. The debugger notifies you when execution reaches the second breakpoint (Fig. M.8). Note that `figM_03`'s normal output appears between messages from the debugger.

**Figure M.8. Continuing execution until the second breakpoint is reached.**

```
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

Breakpoint 2, main () at figM_03.cpp:25
25 account1.debit(withdrawalAmount); // try to subtract from
account1
(gdb)
```

[Page 1387]

7. Examining a variable's value. Type `print withdrawalAmount` to display the current value stored in the `withdrawalAmount` variable (Fig. M.9). The `print command` allows you to peek inside the computer at the value of one of your variables. This command will help you find and eliminate logic errors in your code. Note that the value displayed is 13the value read in and assigned to `withdrawalAmount` in line 22 of Fig. M.3. Use the `print` command to output the contents of the `account1` object. When an object is output through the debugger with the `print` command, the object is output with braces surrounding the object's data members. In this case, there is a single data member`balance`which has a value of 50.

**Figure M.9. Printing the values of variables.**

```
(gdb) print withdrawalAmount
$1 = 13
(gdb) print account1
$2 = {balance = 50}
(gdb)
```

8. Using convenience variables. When the `print` command is used, the result is stored in a convenience variable such as `$1`. Convenience variables, which are temporary variables, named using a dollar sign followed by an integer, are created by the debugger as you print values during your debugging session. A convenience variable can be used in the debugging process to perform arithmetic and evaluate boolean expressions. Type `print $1`. The debugger displays the value of `$1` (Fig. M.10), which contains the value of `withdrawalAmount`. Note that printing the value of `$1` creates a new convenience variable `$3`.

**Figure M.10. Printing a convenience variable.**

```
(gdb) print $1
$3 = 13
(gdb)
```

9. Continuing program execution. Type `continue` to continue the program's execution. The debugger encounters no additional breakpoints, so it continues executing and eventually terminates (Fig. M.11).

**Figure M.11. Finishing execution of the program.**

```
(gdb) continue
Continuing.
account1 balance: $37

Program exited normally.
(gdb)
```

- 10.** Removing a breakpoint. You can display a list of all of the breakpoints in the program by typing `info break`. To remove a breakpoint, type `delete`, followed by a space and the number of the breakpoint to remove. Remove the first breakpoint by typing `delete 1`. Remove the second breakpoint as well. Now type `info break` to list the remaining breakpoints in the program. The debugger should indicate that no breakpoints are set (Fig. M.12).

**Figure M.12. Viewing and removing breakpoints.**

```
(gdb) info break
Num Type Disp Enb Address What
1 breakpoint keep y 0x080487d8 in main at figM_03.cpp:17
breakpoint already hit 1 time
2 breakpoint keep y 0x08048871 in main at figM_03.cpp:25
breakpoint already hit 1 time
(gdb) delete 1
(gdb) delete 2
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

- 11.** Executing the program without breakpoints. Type `run` to execute the program. Enter the value `13` at the prompt. Because you successfully removed the two breakpoints, the program's output is displayed without the debugger entering break mode (Fig. M.13).

**Figure M.13. Program executing with no breakpoints set.**

```
(gdb) run
Starting program: /home/student/Debug/figM_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

12. Using the `quit` command. Use the [quit command](#) to end the debugging session ([Fig. M.14](#)). This command causes the debugger to terminate.

**Figure M.14. Exiting the debugger using the quit command.**



(gdb) quit  
~/Debug\$

[Page 1389]

In this section, you learned how to enable the debugger using the `gdb` command and run a program with the `run` command. You saw how to set a breakpoint at a particular line number in the `main` function. The `break` command can also be used to set a breakpoint at a line number in another file or at a particular function. Typing `break`, then the filename, a colon and the line number will set a breakpoint at a line in another file. Typing `break`, then a function name will cause the debugger to enter the break mode whenever that function is called.

Also in this section, you saw how the `help list` command will provide more information on the `list` command. If you have any questions about the debugger or any of his commands, type `help` or `help` followed by the command name for more information.

Finally, you learned to examine variables with the `print` command and remove breakpoints with the `delete` command. You learned how to use the `continue` command to continue execution after a breakpoint is reached and the `quit` command to end the debugger.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1390]

**Figure M.17. Printing expressions with the debugger.**

```
(gdb) print withdrawalAmount - 2
$1 = 11
(gdb) print withdrawalAmount == 11
$2 = false
(gdb)
```

**5.**

Modifying values. The debugger allows you to change the values of variables during the program's execution. This can be valuable for experimenting with different values and for locating logic errors in programs. You can use the debugger's `set` command to change the value of a variable. Type `set withdrawalAmount = 42`. The debugger changes the value of `withdrawalAmount`. Type `print withdrawalAmount` to display its new value ([Fig. M.18](#)).

**Figure M.18. Setting the value of a variable while in break mode.**

```
(gdb) set withdrawalAmount = 42
(gdb) print withdrawalAmount
$3 = 42
(gdb)
```

**6.**

Viewing the program result. Type `continue` to continue program execution. Line 25 of [Fig. M.3](#) executes, passing `withdrawalAmount` to `Account` member function `debit`. Function `main` then displays the new balance. Note that the result is \$8 ([Fig. M.19](#)). This shows that the preceding step changed the value of `withdrawalAmount` from its initial value (13) to 42.

## Figure M.19. Using a modified variable in the execution of a program.

(This item is displayed on page 1391 in the print version)

```
(gdb) continue
Continuing.
account1 balance: $8

Program exited normally.
(gdb)
```

### 7.

Using the `quit` command. Use the `quit` command to end the debugging session (Fig. M.20). This command causes the debugger to terminate.

## Figure M.20. Exiting the debugger using the `quit` command.

(This item is displayed on page 1391 in the print version)

```
(gdb) quit
~/Debug$
```

In this section, you learned how to use the debugger's `print` command to evaluate arithmetic and boolean expressions. You also learned how to use the `set` command to modify the value of a variable during your program's execution.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1392]

**Figure M.22. Using the `finish` command to complete execution of a function and return to the calling function.**

```
(gdb) finish
Run till exit from #0 Account::debit (this=0xbffffd70, amount=13)
 at Account.cpp:33
main () at figM_03.cpp:28
28 cout << "account1 balance: " << account1.getBalance() << endl;
(gdb)
```

**6.**

Using the `continue` command to continue execution. Enter the `continue` command to continue execution. No additional breakpoints are reached, so the program terminates.

**7.**

Running the program again. Breakpoints persist until the end of the debugging session in which they are set even after execution of the program, all breakpoints are maintained. The breakpoint you set in Step 2 will be there in the next execution of the program. Type `run` to run the program. As in Step 3, the program runs until the breakpoint at line 25 is reached, then the debugger pauses and waits for the next command ([Fig. M.23](#)).

**Figure M.23. Restarting the program.**

```
(gdb) run
Starting program: /home/student/Debug/figM_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

Breakpoint 1, main () at figM_03.cpp:25
25 account1.debit(withdrawalAmount); // try to subtract from
account1
(gdb)
```

**8.**

Using the `next` command. Type `next`. This command behaves like the `step` command, except when the next statement to execute contains a function call. In that case, the called function executes in its entirety and the program advances to the next executable line after the function call (Fig. M.24). In Step 4, the `step` command enters the called function. In this example, the `next` command causes `Account` member function `debit` to execute, then the debugger pauses at line 28.

---

[Page 1393]

**Figure M.24. Using the `next` command to execute a function in its entirety.**

```
(gdb) next
28 cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

**9.**

Using the `quit` command. Use the `quit` command to end the debugging session (Fig. M.25). While the program is running, this command causes the program to immediately terminate rather than execute the remaining statements in `main`.

## Figure M.25. Exiting the debugger using the quit command.

```
(gdb) quit
The program is running. Exit anyway? (y or n) y
~/Debug$
```

In this section, you learned how to use the debugger's `step` and `finish` commands to debug functions called during your program's execution. You saw how the `next` command can be used to step over a function call. You also learned that the `quit` command ends a debugging session.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1394]

**Figure M.28. Stepping into the constructor.**

```
(gdb) step
Account (this=0xbffffd70, initialBalance=50) at Account.cpp:12
12 balance = 0; // assume that the balance begins at 0
(gdb) step
Hardware watchpoint 2: account1.balance

Old value = 1073833120
New value = 0
Account (this=0xbffffd70, initialBalance=50) at Account.cpp:16
16 if (initialBalance > 0)
(gdb)
```

## 5.

Finishing the constructor. Type `step` to execute line 16, then type `step` again to execute line 17. The debugger will notify you that data member `balance`'s value has changed from 0 to 50 ([Fig. M.29](#)).

**Figure M.29. Reaching a watchpoint notification.**

```
(gdb) step
17 balance = initialBalance;
(gdb) step
Hardware watchpoint 2: account1.balance

Old value = 0
New value = 50
Account (this=0xbffffd70, initialBalance=50) at Account.cpp:20
20 if (initialBalance < 0)
(gdb)
```

[Page 1395]

**6.**

Withdrawing money from the account. Type `continue` to continue execution and enter a withdrawal value at the prompt. The program executes normally. Line 25 of [Fig. M.3](#) calls `Account` member function `debit` to reduce the `Account` object's balance by a specified amount. Line 34 of [Fig. M.2](#) inside function `debit` changes the value of `balance`. The debugger notifies you of this change and enters break mode ([Fig. M.30](#)).

**Figure M.30. Entering break mode when a variable is changed.**

```
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

Hardware watchpoint 2: account1.balance

Old value = 50
New value = 37
0x08048a01 in Account::debit (this=0xbffffd70, amount=13) at Account.cpp:34
 34 balance = balance - amount;
(gdb)
```

**7.**

Continuing execution. Type `continue` the program will finish executing function `main` because the program does not attempt any additional changes to `balance`. The debugger removes the watch on `account1`'s `balance` data member because the variable goes out of scope when function `main` ends. Removing the watchpoint causes the debugger to enter break mode. Type `continue` again to finish execution of the program ([Fig. M.31](#)).

**Figure M.31. Continuing to the end of the program.**

```
(gdb) continue
Continuing.
end of function
account1 balance: $37
```

Watchpoint 2 deleted because the program has left the block in which its expression is valid.

```
0x4012fa65 in exit () from /lib/libc.so.6
(gdb) continue
Continuing.
```

Program exited normally.

```
(gdb)
```

## 8.

Restarting the debugger and resetting the watch on the variable. Type `run` to restart the debugger. Once again, set a watch on `account1` data member `balance` by typing `watch account1.balance`. This watchpoint is labeled as `watchpoint 3`. Type `continue` to continue execution (Fig. M.32).

---

[Page 1396]

**Figure M.32. Resetting the watch on a data member.**

```
(gdb) run
Starting program: /home/student/Debug/figM_03

Breakpoint 1, main () at figM_03.cpp:14
14 Account account1(50); // create Account object
(gdb) watch account1.balance
Hardware watchpoint 3: account1.balance
(gdb) continue
Continuing.
Hardware watchpoint 3: account1.balance

Old value = 1073833120
New value = 0
Account (this=0xbffffd70, initialBalance=50) at Account.cpp:16
16 if (initialBalance > 0)
(gdb)
```

**9.**

Removing the watch on the data member. Suppose you want to watch a data member for only part of a program's execution. You can remove the debugger's watch on variable `balance` by typing `delete 3` (Fig. M.33). Type `continue` the program will finish executing without reentering break mode.

**Figure M.33. Removing a watch.**

```
(gdb) delete 3
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

end of function
account1 balance: $37

Program exited normally.
(gdb)
```

In this section, you learned how to use the `watch` command to enable the debugger to notify you of changes to the value of a data member throughout the life of a program. You also learned how to use the `delete` command to remove a watch on a data member before the end of the program.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1397]

 PREV  
page footer

NEXT 

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1397 (continued)]

## Summary

- GNU includes software called a debugger, which allows you to monitor the execution of your programs to locate and remove logic errors.
- The GNU debugger works only with executable files that were compiled with the `-g` compiler option, which generates information that is used by the debugger to help you debug your programs.
- The `gdb` command will start the GNU debugger and enable you to use its features. The `run` command will run a program through the debugger.
- Breakpoints are markers that can be set at any executable line of code. When program execution reaches a breakpoint, execution pauses.
- The `break` command inserts a breakpoint at the line number specified after the command.
- When the program runs, it suspends execution at any line that contains a breakpoint and is said to be in break mode.
- The `continue` command causes the program to continue running until the next breakpoint is reached.
- The `print` command allows you to peek inside the computer at the value of one of your variables.
- When the `print` command is used, the result is stored in a convenience variable such as `$1`. Convenience variables are temporary variables that can be used in the debugging process to perform arithmetic and evaluate boolean expressions.
- You can display a list of all of the breakpoints in the program by typing `info break`.
- To remove a breakpoint, type `delete`, followed by a space and the number of the breakpoint to remove.
- Use the `quit` command to end the debugging session.
- The `set` command allows the programmer to assign new values to variables.
- The `step` command executes the next statement in the program. If the next statement to execute is a function call, control transfers to the called function. The `step` command enables you to enter a function and study the individual statements of that function.
- The `finish` command executes the remaining statements in the function and returns control to the place where the function was called.
- The `next` command behaves like the `step` command, except when the next statement to execute contains a function call. In that case, the called function executes in its entirety and the program advances to the next executable line after the function call.
- The `watch` command sets a watch on any variable or data member of an object currently in scope during execution of the debugger. Whenever the value of a watched variable changes, the debugger enters break mode and notifies you that the value has changed.



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1398]

## Terminology

`break command`

`break mode`

`breakpoint`

`continue command`

`debugger`

`delete command`

`finish command`

`-g compiler option`

`gdb command`

`info break command`

`next command`

`print command`

`quit command`

`run command`

`set command`

[step command](#)

[watch command](#)

[◀ PREV](#)

[NEXT ▶](#)

**page footer**

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1398 (continued)]

## Self-Review Exercises

**M.1** Fill in the blanks in each of the following statements:

a.

A breakpoint cannot be set at a(n)\_\_\_\_\_.

b.

You can examine the value of an expression by using the debugger's \_\_\_\_\_ command.

c.

You can modify the value of a variable by using the debugger's \_\_\_\_\_ command.

d.

During debugging, the \_\_\_\_\_ command executes the remaining statements in the current function and returns program control to the place where the function was called.

e.

The debugger's \_\_\_\_\_ command behaves like the `step` command when the next statement to execute does not contain a function call.

f.

The `watch` debugger command allows you to view all changes to a(n) \_\_\_\_\_.

**M.2** State whether each of the following is true or false. If false, explain why.

**a.**

When program execution suspends at a breakpoint, the next statement to be executed is the statement after the breakpoint.

**b.**

Watches can be removed using the debugger's `remove` command.

**c.**

The `-g` compiler option must be used when compiling programs for debugging.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page 1398 (continued)]

## Answers to Self-Review Exercises

**M.1** a) non-executable line. b) print. c) set. d) finish. e) next. f) data member.

**M.2** a) False. When program execution suspends at a breakpoint, the next statement to be executed is the statement at the breakpoint. b) False. Watches can be removed using the debugger's delete command. c) True.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page 1400]

Carroll, M. D. and M. A. Ellis. Designing and Coding Reusable C++. Reading, MA: Addison-Wesley, 1995.

Coplien, J. O. and D. C. Schmidt. Pattern Languages of Program Design. Reading, MA: Addison-Wesley, 1995.

Deitel, H. M, P. J. Deitel and D. R. Choffnes. Operating Systems, Third Edition. Upper Saddle River, NJ: Prentice Hall, 2004.

Deitel, H. M and P. J. Deitel. Java How to Program, Sixth Edition. Upper Saddle River, NJ: Prentice Hall, 2005.

Deitel, H. M. and P. J. Deitel. C How to Program, Fourth Edition. Upper Saddle River, NJ: Prentice Hall, 2004.

Duncan, R. "Inside C++: Friend and Virtual Functions, and Multiple Inheritance." PC Magazine 15 October 1991, 417420.

Ellis, M. A. and B. Stroustrup. The Annotated C++ Reference Manual. Reading, MA: Addison-Wesley, 1990.

Embley, D. W., B. D. Kurtz and S. N. Woodfield. Object-Oriented Systems Analysis: A Model-Driven Approach. Englewood Cliffs, NJ: Yourdon Press, 1992.

Entsminger, G. and B. Eckel. The Tao of Objects: A Beginner's Guide to Object-Oriented Programming. New York, NY: Wiley Publishing, 1990.

Firesmith, D. G. and B. Henderson-Sellers. "Clarifying Specialized Forms of Association in UML and OML." Journal of Object-Oriented Programming May 1998: 4750.

Flamig, B. Practical Data Structures in C++. New York, NY: John Wiley & Sons, 1993.

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

Gehani, N. and W. D. Roome. The Concurrent C Programming Language. Summit, NJ: Silicon Press, 1989.

Giancola, A. and L. Baker. "Bit Arrays with C++." The C Users Journal Vol. 10, No. 7, July 1992, 2126.

Glass, G. and B. Schuchert. The STL <Primer>. Upper Saddle River, NJ: Prentice Hall PTR, 1995.

- Gooch, T. "Obscure C++." Inside Microsoft Visual C++ Vol. 6, No. 11, November 1995, 1315.
- Hansen, T. L. The C++ Answer Book. Reading, MA: Addison-Wesley, 1990.
- Henricson, M. and E. Nyquist. Industrial Strength C++: Rules and Recommendations. Upper Saddle River, NJ: Prentice Hall, 1997.
- International Standard: Programming LanguagesC++. ISO/IEC 14882:1998. New York, NY: American National Standards Institute, 1998.
- Jacobson, I. "Is Object Technology Software's Industrial Platform?" IEEE Software Magazine Vol. 10, No. 1, January 1993, 2430.
- Jaeschke, R. Portability and the C Language. Indianapolis, IN: Sams Publishing, 1989.
- Johnson, L. J. "Model Behavior." Enterprise Development May 2000: 2028.
- Josuttis, N. The C++ Standard Library: A Tutorial and Reference. Boston, MA: Addison-Wesley, 1999.

---

[Page 1401]

- Knight, A. "Encapsulation and Information Hiding." The Smalltalk Report Vol. 1, No. 8 June 1992, 1920.
- Koenig, A. "What is C++ Anyway?" Journal of Object-Oriented Programming April/May 1991, 4852.
- Koenig, A. "Implicit Base Class Conversions." The C++ Report Vol. 6, No. 5, June 1994, 1819.
- Koenig, A. and B. Stroustrup. "Exception Handling for C++ (Revised)," Proceedings of the USENIX C++ Conference, San Francisco, CA, April 1990.
- Koenig, A. and B. Moo. Ruminations on C++: A Decade of Programming Insight and Experience. Reading, MA: Addison-Wesley, 1997.
- Kruse, R. L. and A. J. Ryba. Data Structures and Program Design in C++. Upper Saddle River, NJ: Prentice Hall, 1999.
- Langer, A. and K. Kreft. Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference. Reading, MA: Addison-Wesley, 2000.
- Lejter, M., S. Meyers and S. P. Reiss. "Support for Maintaining Object-Oriented Programs," IEEE Transactions on Software Engineering Vol. 18, No. 12, December 1992, 10451052.

- Lippman, S. B. and J. Lajoie. C++ Primer, Third Edition, Reading, MA: Addison-Wesley, 1998.
- Lorenz, M. Object-Oriented Software Development: A Practical Guide. Englewood Cliffs, NJ: Prentice Hall, 1993.
- Lorenz, M. "A Brief Look at Inheritance Metrics." The Smalltalk Report Vol. 3, No. 8 June 1994, 1, 45.
- Martin, J. Principles of Object-Oriented Analysis and Design. Englewood Cliffs, NJ: Prentice Hall, 1993.
- Martin, R. C. Designing Object-Oriented C++ Applications Using the Booch Method. Englewood Cliffs, NJ: Prentice Hall, 1995.
- Matsche, J. J. "Object-Oriented Programming in Standard C." Object Magazine Vol. 2, No. 5, January/February 1993, 7174.
- McCabe, T. J. and A. H. Watson. "Combining Comprehension and Testing in Object-Oriented Development." Object Magazine Vol. 4, No. 1, March/April 1994, 6366.
- McLaughlin, M. and A. Moore. "Real-Time Extensions to the UML." Dr. Dobb's Journal December 1998: 8293.
- Melewski, D. "UML Gains Ground." Application Development Trends October 1998: 3444.
- Melewski, D. "UML: Ready for Prime Time?" Application Development Trends November 1997: 3044.
- Melewski, D. "Wherefore and What Now, UML?" Application Development Trends December 1999: 6168.
- Meyer, B. Object-Oriented Software Construction, Second Edition. Englewood Cliffs, NJ: Prentice Hall, 1997.
- Meyer, B. Eiffel: The Language. Englewood Cliffs, NJ: Prentice Hall, 1992.
- Meyer, B. and D. Mandrioli. Advances in Object-Oriented Software Engineering. Englewood Cliffs, NJ: Prentice Hall, 1992.

---

[Page 1402]

- Meyers, S. "Mastering User-Defined Conversion Functions." The C/C++ Users Journal Vol. 13, No. 8, August 1995, 5763.
- Meyers, S. More Effective C++: 35 New Ways to Improve Your Programs and Designs. Reading, MA: Addison-Wesley, 1996.

Meyers, S. Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Second Edition. Reading, MA: Addison-Wesley, 1998.

Meyers, S. Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library. Reading, MA: Addison-Wesley, 2001.

Muller, P. Instant UML. Birmingham, UK: Wrox Press Ltd, 1997.

Murray, R. C++ Strategies and Tactics. Reading, MA: Addison-Wesley, 1993.

Musser, D. R. and A. A. Stepanov. "Algorithm-Oriented Generic Libraries." Software Practice and Experience Vol. 24, No. 7, July 1994.

Musser, D. R., G. J. Derge and A. Saini. STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition. Reading, MA: Addison-Wesley, 2001.

Nerson, J. M. "Applying Object-Oriented Analysis and Design." Communications of the ACM, Vol. 35, No. 9, September 1992, 6374.

Nierstrasz, O., S. Gibbs and D. Tsichritzis. "Component-Oriented Software Development." Communications of the ACM Vol. 35, No. 9, September 1992, 160165.

Perry, P. "UML Steps to the Plate." Application Development Trends May 1999: 3336.

Pinson, L. J. and R. S. Wiener. Applications of Object-Oriented Programming. Reading, MA: Addison-Wesley, 1990.

Pittman, M. "Lessons Learned in Managing Object-Oriented Development." IEEE Software Magazine Vol. 10, No. 1, January 1993, 4353.

Plauger, P. J. The Standard C Library. Englewood Cliffs, NJ: Prentice Hall, 1992.

Plauger, D. "Making C++ Safe for Threads." The C Users Journal Vol. 11, No. 2, February 1993, 5862.

Pohl, I. C++ Distilled: A Concise ANSI/ISO Reference and Style Guide. Reading, MA: Addison-Wesley, 1997.

Press, W. H., S. A. Teukolsky, W. T. Vetterling and B. P. Flannery. Numerical Recipes in C: The Art of Scientific Computing. Cambridge, MA: Cambridge University Press, 1992.

Prieto-Diaz, R. "Status Report: Software Reusability." IEEE Software Vol. 10, No. 3, May 1993, 6166.

Prince, T. "Tuning Up Math Functions." The C Users Journal Vol. 10, No. 12, December 1992.

Prosise, J. "Wake Up and Smell the MFC: Using the Visual C++ Classes and Applications Framework." Microsoft Systems Journal Vol. 10, No. 6, June 1995, 1734.

Rabinowitz, H. and C. Schaap. Portable C. Englewood Cliffs, NJ: Prentice Hall, 1990.

Reed, D. R. "Moving from C to C++." Object Magazine Vol. 1, No. 3, September/October 1991, 4660.

Ritchie, D. M. "The UNIX System: The Evolution of the UNIX Time-Sharing System." AT&T Bell Laboratories Technical Journal Vol. 63, No. 8, Part 2, October 1984, 15771593.

---

[Page 1403]

Ritchie, D. M., S. C. Johnson, M. E. Lesk and B. W. Kernighan. "UNIX Time-Sharing System: The C Programming Language." The Bell System Technical Journal Vol. 57, No. 6, Part 2, July/August 1978, 19912019.

Rosler, L. "The UNIX System: The Evolution of CPast and Future." AT&T Laboratories Technical Journal Vol. 63, No. 8, Part 2, October 1984, 16851699.

Robson, R. Using the STL: The C++ Standard Template Library. New York, NY: Springer Verlag, 2000.

Rubin, K. S. and A. Goldberg. "Object Behavior Analysis." Communications of the ACM Vol. 35, No. 9, September 1992, 4862.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. Object-Oriented Modeling and Design. Englewood Cliffs, NJ: Prentice Hall, 1991.

Rumbaugh, J., Jacobson, I. and G. Booch. The Unified Modeling Language Reference Manual, Second Edition. Reading, MA: Addison-Wesley, 2005.

Saks, D. "Inheritance." The C Users Journal May 1993, 8189.

Schildt, H. STL Programming from the Ground Up. Berkeley, CA: Osborne McGraw-Hill, 1999.

Schlaer, S. and S. J. Mellor. Object Lifecycles: Modeling the World in States. Englewood Cliffs, NJ: Prentice Hall, 1992.

Sedgwick, R. Bundle of Algorithms in C++, Parts 15: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms (Third Edition). Reading, MA: Addison-Wesley, 2002.

Sessions, R. Class Construction in C and C++: Object-Oriented Programming. Englewood Cliffs, NJ: Prentice Hall, 1992.

- Skelly, C. "Pointer Power in C and C++." *The C Users Journal* Vol. 11, No. 2, February 1993, 9398.
- Snyder, A. "The Essence of Objects: Concepts and Terms." *IEEE Software Magazine* Vol. 10, No. 1, January 1993, 3142.
- Stepanov, A. and M. Lee. "The Standard Template Library." 31 October 1995 <[www.cs.rpi.edu/~mussel/doc.ps](http://www.cs.rpi.edu/~mussel/doc.ps)>.
- Stroustrup, B. "The UNIX System: Data Abstraction in C." *AT&T Bell Laboratories Technical Journal* Vol. 63, No. 8, Part 2, October 1984, 17011732.
- Stroustrup, B. "What is Object-Oriented Programming?" *IEEE Software* Vol. 5, No. 3, May 1988, 1020.
- Stroustrup, B. "Parameterized Types for C++." *Proceedings of the USENIX C++ Conference* Denver, CO, October 1988.
- Stroustrup, B. "Why Consider Language Extensions?: Maintaining a Delicate Balance." *The C++ Report* September 1993, 4451.
- Stroustrup, B. "Making a `vector` Fit for a Standard." *The C++ Report* October 1994.
- Stroustrup, B. *The Design and Evolution of C++*. Reading, MA: Addison-Wesley, 1994.
- Stroustrup, B. *The C++ Programming Language*, Special Third Edition. Reading, MA: Addison-Wesley, 2000.
- Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++. Reading, MA: Addison-Wesley, 1994.
- 
- [Page 1404]
- Taylor, D. *Object-Oriented Information Systems: Planning and Implementation*. New York, NY: John Wiley & Sons, 1992.
- Tondo, C. L. and S. E. Gimpel. *The C Answer Book*. Englewood Cliffs, NJ: Prentice Hall, 1989.
- Urlocker, Z. "Polymorphism Unbounded." *Windows Tech Journal* Vol. 1, No. 1, January 1992, 1116.
- Van Camp, K. E. "Dynamic Inheritance Using Filter Classes." *The C/C++ Users Journal* Vol. 13, No. 6, June 1995, 6978.
- Vilot, M. J. "An Introduction to the Standard Template Library." *The C++ Report* Vol. 6, No. 8, October 1994.

- Voss, G. Object-Oriented Programming: An Introduction. Berkeley, CA: Osborne McGraw-Hill, 1991.
- Voss, G. "Objects and Messages." Windows Tech Journal February 1993, 1516.
- Wang, B. L. and J. Wang. "Is a Deep Class Hierarchy Considered Harmful?" Object Magazine Vol. 4, No. 7, November/December 1994, 3536.
- Weisfeld, M. "An Alternative to Large Switch Statements." The C Users Journal Vol. 12, No. 4, April 1994, 6776.
- Weiskamp, K. and B. Flamig. The Complete C++ Primer, Second Edition. Orlando, FL: Academic Press, 1993.
- Wiebel, M. and S. Halladay. "Using OOP Techniques Instead of switch in C++." The C Users Journal Vol. 10, No. 10, October 1993, 105112.
- Wilde, N. and R. Huitt. "Maintenance Support for Object-Oriented Programs." IEEE Transactions on Software Engineering Vol. 18, No. 12, December 1992, 10381044.
- Wilde, N., P. Matthews and R. Huitt. "Maintaining Object-Oriented Software." IEEE Software Magazine Vol. 10, No. 1, January 1993, 7580.
- Wilson, G. V. and P. Lu. Parallel Programming Using C++. Cambridge, MA: MIT Press, 1996.
- Wilt, N. "Templates in C++." The C Users Journal May 1993, 3351.
- Wirfs-Brock, R., B. Wilkerson and L. Wiener. Designing Object-Oriented Software. Englewood Cliffs, NJ: Prentice Hall PTR, 1990.
- Wyatt, B. B., K. Kavi and S. Hufnagel. "Parallelism in Object-Oriented Languages: A Survey." IEEE Software Vol. 9, No. 7, November 1992, 5666.
- Yamazaki, S., K. Kajihara, M. Ito and R. Yasuhara. "Object-Oriented Design of Telecommunication Software." IEEE Software Magazine Vol. 10, No. 1, January 1993, 8187.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page EULA-1]

## End User License Agreements

Prentice Hall License Agreement and Limited Warranty

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[Page EULA-2]

IN NO EVENT, SHALL THE COMPANY OR ITS EMPLOYEES, AGENTS, SUPPLIERS, OR CONTRACTORS BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR IN CONNECTION WITH THE LICENSE GRANTED UNDER THIS AGREEMENT, OR FOR LOSS OF USE, LOSS OF DATA, LOSS OF INCOME OR PROFIT, OR OTHER LOSSES, SUSTAINED AS A RESULT OF INJURY TO ANY PERSON, OR LOSS OF OR DAMAGE TO PROPERTY, OR CLAIMS OF THIRD PARTIES, EVEN IF THE COMPANY OR AN AUTHORIZED REPRESENTATIVE OF THE COMPANY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL LIABILITY OF THE COMPANY FOR DAMAGES WITH RESPECT TO THE SOFTWARE EXCEED THE AMOUNTS ACTUALLY PAID BY YOU, IF ANY, FOR THE SOFTWARE.

SOME JURISDICTIONS DO NOT ALLOW THE LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT ALWAYS APPLY. THE WARRANTIES IN THIS AGREEMENT GIVE YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY IN ACCORDANCE WITH LOCAL LAW.

#### ACKNOWLEDGMENT

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT, AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU ALSO AGREE THAT THIS AGREEMENT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN YOU AND THE COMPANY AND SUPERSEDES ALL PROPOSALS OR PRIOR AGREEMENTS, ORAL, OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN YOU AND THE COMPANY OR ANY REPRESENTATIVE OF THE COMPANY RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Should you have any questions concerning this Agreement or if you wish to contact the Company for any reason, please contact in writing at the address below.

Robin Short  
Prentice Hall PTR  
One Lake Street  
Upper Saddle River, New Jersey 07458

 PREV

page footer

NEXT 

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page EULA-3]

## License Agreement and Limited Warranty

The software is distributed on an "AS IS" basis, without warranty. Neither the authors, the software developers, nor Prentice Hall make any representation, or warranty, either express or implied, with respect to the software programs, their quality, accuracy, or fitness for a specific purpose. Therefore, neither the authors, the software developers, nor Prentice Hall shall have any liability to you or any other person or entity with respect to any liability, loss, or damage caused or alleged to have been caused directly or indirectly by the programs contained on the media. This includes, but is not limited to, interruption of service, loss of data, loss of classroom time, loss of consulting or anticipatory profits, or consequential damages from the use of these programs. If the media itself is defective, you may return it for a replacement. Use of this software is subject to the Binary Code License terms and conditions at the back of this book. Read the licenses carefully. By opening this package, you are agreeing to be bound by the terms and conditions of these licenses. If you do not agree, do not open the package.

Please refer to end-user license agreements on the CD-ROM for further details.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page EULA-3 (continued)]

## Using the CD-ROM

The interface to the contents of this CD is designed to start automatically through the AUTORUN.EXE file. If a startup screen does not pop up automatically when you insert the CD into your computer, double click on the welcome.htm file to launch the Student CD or refer to the file readme.txt on the CD.

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

[Page EULA-3 (continued)]

## Contents of the CD-ROM

- Software downloads: links to free C++ compilers and development tools
- Examples
- Web Resources

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

[Page EULA-3 (continued)]

## Software and Hardware System Requirements

- 450 MHz (minimum) Pentium II or faster processor
- Microsoft® Windows® Server 2003, XP Professional, XP Home Edition, XP Media Center Edition, XP Tablet PC Edition, 2000 Professional (SP3 or later required for installation), 2000 Server (SP3 or later required for installation), or
- One of the following Linux distributions: Fedora Core 3 (Formerly Red Hat Linux), Mandrakelinux 10.1 Official, Red Hat 9.0, SUSE LINUX Professional 9.2, or Turbolinux 10 Desktop
- Other than the minimum amount of RAM required by the items listed above, there are no additional RAM requirements for this CD-ROM.
- CD-ROM drive
- Internet connection
- C++ environment
- Web browser, Adobe® Acrobat® Reader® and a Zip decompression utility

[◀ PREV](#)[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

## Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]  
[[Z](#)]

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)[NEXT ▶](#)

## Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)]  
[[Y](#)] [[Z](#)]

!, logical NOT operator 2nd 3rd  
    truth table  
!=, inequality operator 2nd 3rd  
##, preprocessor operator  
#, preprocessor operator  
`#define NDEBUG`  
`#define PI 3.14159`  
`#define preprocessor directive 2nd 3rd 4th 5th`  
`#elif`  
`#endif preprocessor directive 2nd`  
`#error preprocessor directive`  
`#if preprocessor directive`  
`#ifdef preprocessor directive`  
`#ifndef preprocessor directive 2nd`  
`#include "filename"`  
`#include preprocessor directive 2nd`  
`#pragma preprocessor directive`  
`#undef preprocessor directive 2nd`  
\$ UNIX command-line prompt  
%, modulus operator  
%=, modulus assignment operator  
& and \* operators as inverses  
& to declare reference  
    in a parameter list  
&&, logical AND operator 2nd 3rd  
    truth table  
&, address operator 2nd 3rd  
&, bitwise AND  
&=, bitwise AND assignment operator 2nd 3rd

## SYMBOL

& special character (XHTML)  
&copy; special character (XHTML)  
&frac14; special character (XHTML)  
&lt; special character (XHTML)  
'\0', null character  
'\n', newline character  
\*, multiplication operator  
\*, pointer dereference or indirection operator 2nd  
\*+, multiplication assignment operator  
+  
++, postfix increment operator  
++, prefix increment operator  
+, addition operator 2nd  
+=, addition assignment operator 2nd  
    overloaded  
, comma operator  
--, postfix decrement operator  
--, prefix decrement operator  
-->, XHTML comments end tag  
-=, subtraction assignment operator  
->\*, pointer to member operator  
->, member selection via pointer

0x, prefix for a hexadecimal number  
2-D array  
::, binary scope resolution operator 2nd  
::, unary scope resolution operator 2nd 3rd  
<!--, XHTML comments start tag  
<, less-than operator  
</p> (XHTML paragraph end tag)  
</tr> (XHTML table row end tag)  
<<, stream insertion operator 2nd  
<<=, left-shift assignment operator  
<=, less-than-or-equal-to operator  
<algorithm> header file 2nd  
<assert.h> header file  
<bitset> header file 2nd  
<cassert> header file 2nd  
<cctype> header file 2nd 3rd  
<cfloat> header file  
<climits> header file  
<cmath> header file  
<csdlib> header file 2nd 3rd 4th 5th 6th 7th 8th  
<csetjmp> header file  
<csignal> header file  
<cstdio> header file  
<cstring> header file 2nd 3rd  
<ctime> header file 2nd 3rd  
<ctrl>-c  
<ctrl>-d 2nd  
<ctrl>-z 2nd 3rd  
<ctype.h> header file  
<deque> header file 2nd 3rd  
<exception> header file 2nd 3rd 4th  
<float.h> header file  
<fstream> header file 2nd  
<functional> header file 2nd 3rd  
<iomanip.h> header file 2nd  
<iomanip> header file 2nd 3rd 4th  
<iostream.h> header file  
<iostream> header file 2nd 3rd 4th 5th 6th 7th 8th  
<iterator> header file 2nd 3rd  
<limits.h> header file

## SYMBOL

<limits> header file  
<list> header file 2nd 3rd  
<locale> header file  
<map> header file 2nd 3rd 4th  
<math.h> header file  
<memory> header file 2nd  
<new.h> header file  
<new> header file  
<numeric> header file 2nd  
<p> (XHTML paragraph start tag)  
<queue> header file 2nd 3rd 4th  
<set> header file 2nd 3rd 4th  
<sstream> header file 2nd  
<stack> header file 2nd 3rd  
<stdexcept> header file 2nd 3rd  
<stdio.h> header file  
<stdlib.h> header file  
<string.h> header file  
<string> header file 2nd 3rd 4th 5th  
<time.h> header file  
<tr> (XHTML table start tag)  
<typeinfo> header file 2nd  
<utility> header file  
<vector> header file 2nd 3rd  
=, assignment operator 2nd 3rd  
==, equality operator 2nd 3rd  
>, greater-than operator  
>=, greater-than-or-equal-to operator  
>>, right shift operator  
>>, stream extraction operator  
>>=, right shift with sign extension assignment operator  
?:, ternary conditional operator 2nd  
\", double-quote-character escape sequence  
\', single-quote-character escape sequence  
\\", backslash-character escape sequence  
\a, alert escape sequence  
\n, newline escape sequence  
\r, carriage-return escape sequence  
\t, tab escape sequence  
^, bitwise exclusive OR operator 2nd

`^=`, bitwise exclusive OR assignment operator 2nd 3rd  
`__cplusplus` predefined symbolic constant  
`__DATE__` predefined symbolic constant  
`__FILE__` symbolic constant  
`__LINE__` predefined symbolic constant  
`__STDC__` predefined symbolic constant  
`__TIME__` predefined symbolic constant  
`|`, bitwise inclusive OR operator 2nd  
`|=`, bitwise inclusive OR assignment operator 2nd 3rd  
`||`, logical OR operator 2nd 3rd 4th  
    truth table  
`~`, bitwise complement operator 2nd

 PREVNEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[J\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Y\]](#)

[\[Z\]](#)

a, XHTML anchor element 2nd

abbreviating assignment expressions

abort a program

abort function 2nd 3rd 4th 5th

absolute value

abstract base class 2nd 3rd 4th

abstract class 2nd 3rd 4th

abstract data type (ADT) 2nd

abstract operation in the UML

access function

access global variable

access non-static class data members and member functions

access private member of a class

access privileges 2nd

access specifier 2nd 3rd 4th

    private

    private: label

    protected

    public 2nd

    public: label

access structure member

access the caller's data

access union members

access violation 2nd 3rd

accessing an object's members through each type of object handle

accessor function

Account class

    ATM case study 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th

    exercise

inheritance hierarchy exercise  
accounts-receivable file  
accounts-receivable system 2nd  
accumulate STL algorithm 2nd 3rd 4th 5th  
accumulated outputs  
accumulator 2nd  
action 2nd 3rd 4th 5th 6th 7th  
action attribute (XHTML form) 2nd 3rd  
action expression (UML) 2nd 3rd 4th 5th 6th 7th 8th  
action of an object  
action oriented  
action state (UML) 2nd 3rd 4th  
    symbol  
action/decision model of programming 2nd  
activation in a UML sequence diagram  
activation record  
active window  
activity (UML) 2nd 3rd  
activity diagram (UML) 2nd 3rd 4th 5th 6th 7th 8th 9th  
    do...while statement  
    for statement  
    if statement  
    sequence statement  
    switch statement  
activity of a portion of a software system  
actor in use case (UML)  
Ada Lovelace  
Ada programming language  
adapter (STL) 2nd  
add a new account to a file  
add an integer to a pointer  
addition 2nd 3rd 4th  
addition assignment operator ( $+=$ )  
addition program that displays the sum of two numbers  
address  
    of a bit field  
    of a structure  
address operator ( $\&$ ) 2nd 3rd 4th 5th 6th  
addressable storage unit  
adjacent\_difference STL algorithm  
adjacent\_find STL algorithm 2nd  
"administrative" section of the computer

ADT (abstract data type) 2nd

aggregate data type 2nd

aggregation 2nd

aiming a derived-class pointer at a base-class object

airline reservation system 2nd

alert escape sequence (' \a ')

alert escape sequence (' a ')

algebraic expression

algorithm 2nd 3rd 4th 5th 6th 7th 8th 9th 10th

action to execute

binary search

bubble sort

bucket sort

insertion sort 2nd

linear search

merge sort

order in which actions should execute

procedure

quicksort

recursive binary search

recursive linear search

selection sort 2nd

algorithms (STL)

accumulate 2nd

binary\_search 2nd

copy\_backward 2nd

copy\_backward, merge, unique and reverse

count 2nd

count\_if 2nd

remove, remove\_if, remove\_copy and remove\_copy\_if

equal

equal, mismatch and lexicographical\_compare

equal\_range 2nd

fill

fill, fill\_n, generate and generate\_n

fill\_n

find 2nd

find\_if 2nd

for\_each 2nd

generate

generate\_n

implace\_merge

includes  
inplace\_merge  
inplace\_merge, unique\_copy and reverse\_copy  
iter\_swap 2nd  
lexicographical\_compare 2nd  
lower\_bound  
lower\_bound, upper\_bound and equal\_range  
make\_heap  
max  
max\_element 2nd  
merge 2nd  
min  
min and max  
min\_element 2nd  
mismatch 2nd  
of the STL  
pop\_heap  
push\_heap  
random\_shuffle 2nd  
remove  
remove\_copy  
remove\_copy\_if 2nd  
remove\_if 2nd  
replace  
replace, replace\_if, replace\_copy and replace\_copy\_if  
replace\_copy 2nd  
replace\_copy\_if 2nd  
replace\_if 2nd  
reverse 2nd  
reverse\_copy 2nd  
separated from containers  
set\_difference 2nd  
set\_intersection 2nd  
set\_symmetric\_difference 2nd  
set\_union 2nd  
sort 2nd  
sort\_heap  
swap  
swap, iter\_swap and swap\_ranges  
swap\_ranges 2nd  
transform 2nd

unique 2nd  
unique\_copy 2nd

upper\_bound

alias 2nd 3rd 4th

for the name of an object

alignment

allocate dynamic memory 2nd 3rd 4th

allocator

alphabetizing strings

alt attribute of element img (XHTML)

alter the flow of control

ALU

ambiguity problem 2nd

American National Standards Institute (ANSI) 2nd 3rd

American Standard Code for Information Interchange (ASCII) 2nd

analysis

analysis stage of the software life cycle

Analytical Engine

and operator keyword

and\_eq operator keyword

angle brackets (<and>)

for header file names

in templates 2nd

anonymous union

ANSI (American National Standards Institute)

ANSI C

ANSI/ISO 9899: 1990

ANSI/ISO C++ Standard 2nd 3rd 4th 5th

any function of class bitset

Apache HTTP Server 2nd

Apache Software Foundation

append data to a file

append function of class string

append output symbol (>>)

Apple Computer, Inc.

arbitrary range of subscripts

argc

argument

coercion

passed to member-object constructor

to a function

to a macro

arguments in correct order

argv [ ]

arithmetic and logic unit (ALU)

arithmetic assignment operators

arithmetic calculations

arithmetic mean

arithmetic operator

arithmetic overflow 2nd

arithmetic overflow error

arithmetic underflow error

"arity" of an operator

array 2nd 3rd 4th 5th 6th

    assignment

    bounds

    bounds checking

    comparison

    initializer list

    input/output

    name 2nd

    name as a constant pointer to beginning of array 2nd 3rd

    notation for accessing elements

    of pointers to functions 2nd

    of strings

    passed by reference

    size

    subscript operator ( [ ] )

    subscripting 2nd

    use with functions

Array class

    definition with overloaded operators 2nd

    member-function and friend function definitions

array-sort function

arrays that know their size

arrow 2nd

arrow member selection operator (->) 2nd

arrowhead in a UML sequence diagram

artifact in the UML

ASCII (American Standard Code for Information Interchange) 2nd 3rd 4th

    character set

    decimal equivalent of a character

ASCII (American Standard Code for Information Interchange) character set

asctime function 2nd

assembler  
assembly language  
assert  
assign member function of class string  
assign member function of list  
assign one iterator to another  
assign the value of  
assigning a structure to a structure of the same type  
assigning a union to another union of the same type  
assigning addresses of base-class and derived-class objects to base-class and derived-class pointers  
assigning character strings to String objects  
assigning class objects  
assignment operator  
  %= modulus assignment operator  
  \*= multiplication assignment operator  
  += addition assignment operator  
  -= subtraction assignment operator  
  /= division assignment operator  
  = 2nd 3rd 4th 5th 6th  
  overloaded operator function  
assignment statement 2nd  
association (in the UML) 2nd 3rd 4th 5th  
  name  
association in a map  
associative array  
associative container 2nd 3rd 4th 5th  
associative container functions  
  count  
  equal\_range  
  find  
  insert 2nd  
  lower\_bound  
  upper\_bound  
associativity 2nd  
  chart  
  left to right 2nd 3rd  
  not changed by overloading  
  of operators 2nd  
asterisk (\*) for multiplication  
asynchronous call  
asynchronous event  
at member function

of class `string` 2nd 3rd  
of class `vector` 2nd 3rd

AT&T

`atexit` function 2nd

ATM (automated teller machine) case study 2nd

ATM class (ATM case study) 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th

ATM system 2nd 3rd 4th 5th 6th 7th 8th

`atof` function 2nd

`atoi` function 2nd 3rd 4th 5th 6th

`atol` function 2nd

attempting to call a multiply inherited function polymorphically

attempting to modify a constant pointer to constant data

attempting to modify a constant pointer to nonconstant data

attempting to modify data through a non-constant pointer to constant data

attribute 2nd 3rd

compartment in a class diagram

declaration in the UML 2nd

in the UML 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th

name in the UML

of an element (XHTML)

attributes of a variable

`auto` storage-class specifier 2nd

`auto_ptr` class 2nd

automated teller machine (ATM) 2nd 3rd

user interface

automatic array

automatic array initialization

automatic local array

automatic local object

automatic local variable 2nd 3rd

automatic object 2nd

automatic storage class 2nd 3rd 4th

automatic variable

automatically destroyed

automobile

Autos window

displaying state of objects

Autos window displaying local variable `withdrawalAmount`

Autos window displaying the state of `account1` object

Autos window displaying the state of `account1` object after initialization

Autos window displaying updated local variable `withdrawalAmount`

average 2nd 3rd 4th

## avoid repeating code

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

B programming language

Babbage, Charles

back member function

    of queue

    of sequence containers 2nd

back\_inserter function template 2nd

backslash (\) 2nd

backslash escape sequence (\\\)

backward pointer

backward traversal

bad member function (streams)

bad\_alloc exception 2nd 3rd 4th 5th 6th

bad\_cast exception

bad\_exception exception

bad\_typeid exception

badbit of stream 2nd 3rd

balanced tree

BalanceInquiry class (ATM case study) 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th

16th

Bank account program

BankDatabase class (ATM case study) 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th

banking system

bar chart 2nd 3rd

    printing program

bar of asterisks 2nd

base 2

base case in recursion 2nd 3rd

base class 2nd 3rd 4th

    catch handler

constructor

exception

initializer syntax

member accessibility in derived class

member function redefined in a derived class

pointer (or reference type)

pointer to a derived-class object

private member

subobject

base e

base specified for a stream

base-10 number system 2nd

base-16 number system

base-8 number system

baseline (XHTML)

BasePlusCommissionEmployee class header file

BasePlusCommissionEmployee class implementation file 2nd

BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission

BasePlusCommissionEmployee class test program

BasePlusCommissionEmployee class that inherits from class Commission-Employee, which does not provide protected data

BasePlusCommissionEmployee class that inherits protected data from CommissionEmployee

BASIC (Beginner's All-Purpose Symbolic Instruction Code) 2nd 3rd 4th

basic searching and sorting algorithms of the Standard Library

basic\_fstream template 2nd

basic\_ifstream template 2nd

basic\_ios class template 2nd

basic\_iostream class template 2nd 3rd 4th

basic\_istream class template 2nd 3rd

basic\_istringstream class template

basic\_ofstream class template 2nd

basic\_ostream class template 2nd 3rd

basic\_ostringstream class template

basic\_string class template

batch processing

BCPL programming language

begin iterator

begin member function of class string

begin member function of containers

begin member function of first-class containers

Beginner's All-Purpose Symbolic Instruction Code (BASIC)

beginning of a file

beginning of a stream

behavior 2nd 3rd

behavior of the system 2nd 3rd 4th

behaviors in the UML

bell

Bell Laboratories

bibliography

bidirectional iterator 2nd 3rd 4th 5th 6th 7th 8th 9th 10th

operations

bidirectional navigability in the UML

Big O notation 2nd 3rd 4th 5th 6th

binary (base 2) number system

binary arithmetic operator

binary comparison function

binary digit

binary function 2nd

binary function object 2nd

binary integer

binary number

binary number system

binary operator 2nd 3rd

binary predicate function 2nd 3rd 4th 5th 6th 7th 8th 9th

binary scope resolution operator (:) 2nd 3rd 4th 5th

binary search

algorithm 2nd

binary search tree 2nd 3rd 4th

binary tree 2nd 3rd 4th

delete

level-order traversal 2nd

search

sort 2nd

with duplicates

binary\_function class

binary\_search STL algorithm 2nd 3rd

bit 2nd

bit field 2nd 3rd

manipulation

member of structure

to save space

bit manipulation 2nd

bitand operator keyword

bitor operator keyword

"bits-and-bytes" level

bitset

  flip

  reset

bitset class 2nd 3rd 4th

bitwise AND

  assignment operator ( $\&=$ ) 2nd

  operator ( $\&$ ) 2nd 3rd 4th

bitwise AND, bitwise inclusive-OR, bitwise exclusive-OR and bitwise complement operators

bitwise assignment operators 2nd

  keywords

bitwise complement operator ( $\sim$ ) 2nd 3rd 4th 5th 6th

bitwise exclusive OR assignment operator ( $\hat{=}$ )

bitwise exclusive OR operator ( $\wedge$ ) 2nd 3rd 4th

bitwise inclusive OR assignment operator ( $\mid=$ )

bitwise inclusive OR operator ( $\mid$ ) 2nd 3rd 4th 5th

bitwise left-shift operator ( $<<$ ) 2nd

bitwise logical OR operator ( $\mid\mid$ )

bitwise operators 2nd 3rd

  keywords

bitwise right-shift operator ( $>>$ )

bitwise shift operator

blank

  blank character

  blank line 2nd

block 2nd 3rd 4th 5th 6th 7th 8th

  block is active

  block is exited

  block of data

  block of memory 2nd 3rd

  block scope 2nd

    variable

body

  of a class definition

  of a function 2nd

  of a loop 2nd 3rd 4th

body element (XHTML) 2nd

body section

Böhm, C. 2nd

"bombing"

Booch, Grady  
bool data type  
bool value true  
boolalpha stream manipulator 2nd  
Boolean attribute in the UML  
border attribute of element table  
Borland C++ 2nd 3rd  
Borland C++Builder  
bottom of a stack  
bottom tier  
boundary of a storage unit  
bounds checking  
box  
br element (XHTML)  
braces ({} ) 2nd 3rd 4th 5th 6th 7th  
    in a do...while statement  
bracket ([ ])  
branch negative  
branch zero instruction 2nd  
break debugger command  
break mode 2nd  
break statement 2nd 3rd 4th  
    exiting a for statement  
breakpoint 2nd  
    disabling  
    inserting 2nd 3rd 4th  
    maroon circle, solid  
    yellow arrow in break mode  
brittle software  
browser 2nd  
browser request  
brute force processing  
"brute force" computing  
bubble sort 2nd  
    improving performance  
bucket sort 2nd  
buffer fills  
buffer is filled  
buffer is flushed  
buffered output  
buffered standard error stream  
buffering

bug  
building a compiler  
"building blocks"  
building your own compiler 2nd  
building-block appearance  
building-block approach  
built-in type 2nd 3rd  
business logic  
    tier  
business rule  
business software  
business-critical computing  
button attribute value (type)  
byte 2nd

 PREVNEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

C legacy code 2nd 3rd 4th 5th 6th

C programming language 2nd

C# programming language

C++ 2nd 3rd 4th

    career resources

    compiler

    development environment

    environment

    executable

    language

    preprocessor 2nd

    programming environment

    programming language

    resources on the Web 2nd 3rd 4th

C++ Standard Library 2nd

    <string> header file

    class template vector

    header file location

    header files

    string class

C-like pointer-based array

C-style char \* strings 2nd

C-style dynamic memory allocation

C-style pointer-based array

c\_str member function of class string 2nd

cache

calculate a salesperson's earnings

calculate the value of  $\pi$

calculations 2nd 3rd

call a function

call stack  
calling environment  
calling function (caller) 2nd 3rd  
calling functions by reference  
calloc  
camel case  
capacity member function of a string  
capacity member function of vector 2nd  
capacity of a string  
caption element (XHTML)  
card dealing algorithm  
card games  
card shuffling and dealing simulation 2nd 3rd 4th 5th 6th  
career resource  
carriage return ('\r') 2nd  
carriage-return escape sequence ('\r')  
carry bit  
CART cookie  
cascaded assignments  
cascading += operators  
cascading member function calls 2nd 3rd  
cascading stream insertion operations  
case label 2nd  
case sensitive  
CashDispenser class (ATM case study) 2nd 3rd 4th 5th 6th  
casino  
cast 2nd  
    downcast 2nd  
cast away const-ness  
cast expression  
cast operator 2nd 3rd 4th 5th 6th  
cast operator function  
cast variable visible in debugger  
cataloging  
catch  
    a base class object  
    all exceptions  
    clause (or handler) 2nd  
    handler  
    keyword  
catch related errors  
catch(...) 2nd

ceil function

Celsius and Fahrenheit Temperatures exercise

central processing unit (CPU)

cerr (standard error unbuffered) 2nd 3rd 4th

CGI (Common Gateway Interface)

CGI script 2nd 3rd 4th 5th

cgi-bin directory 2nd 3rd

chaining stream insertion operations

char \*\*

char data type 2nd 3rd 4th 5th

character 2nd

character array 2nd 3rd 4th 5th

as a string

character code

character constant

character entity reference

character manipulation

character presentation

character sequences

character set 2nd 3rd 4th

character string 2nd 3rd

character-handling functions 2nd

character-handling functions isdigit, isalpha, isalnum and isxdigit

character-handling functions islower, isupper, tolower and toupper

character-handling functions isspace, iscntrl, ispunct, isprint and isgraph

character-string manipulation

character's numerical representation

characters represented as numeric codes

checkbox (XHTML)

checkbox attribute value (type)

checked access

checked attribute

checkerboard pattern 2nd

checkout line in a supermarket

child

child node

Chinese

cin (standard input stream) 2nd 3rd 4th 5th 6th

function clear

function eof 2nd

function get 2nd

function getline

function tie  
circular include  
circular, doubly linked list  
circular, singly linked list  
clarity 2nd 3rd 4th  
class 2nd 3rd 4th 5th 6th 7th  
    attribute  
    client-code programmer  
    constructor  
    data member 2nd  
    default constructor 2nd  
    define a constructor  
    define a member function  
    defining  
    definition  
    development  
    implementation programmer  
    instance of  
    interface  
    interface described by function prototypes  
    member function 2nd  
    member function implementations in a separate source-code file  
    name  
    naming convention  
    object of  
    public services  
    services  
class average on a quiz  
class average problem 2nd  
class diagram  
    for the ATM system model 2nd  
    in the UML 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th  
Class DivideByZeroException definition  
class hierarchy 2nd 3rd 4th  
class Integer definition  
class keyword 2nd  
class library 2nd 3rd 4th 5th  
class members default to private access  
class scope 2nd 3rd  
class template 2nd 3rd 4th 5th  
    definition  
    explicit specialization

c

scope  
specialization 2nd  
Stack 2nd

class template auto\_ptr

class variable

class-implementation programmer

class-scope variable is hidden

class's object code

class's source code

classes

Array

auto\_ptr 2nd

binary\_function

bitset and the Sieve of Eratosthenes

Complex

deque

exception

HugeInt

invalid\_argument

list

Node

Polynomial

RationalNumber

string 2nd

vector 2nd

classic stream libraries

clear function of ios\_base

clear member function of containers

clear member function of first-class containers

client 2nd 3rd

client code

client computer 2nd

client of a class 2nd 3rd

client of a queue

client of an object 2nd

client tier

client-code programmer 2nd

client/server computing

clients.txt

clog (standard error buffered) 2nd 3rd

close a stream

close function of `ofstream`  
cn (top-level domain)  
COBOL (COmmon Business Oriented Language) 2nd  
coefficient  
coin tossing 2nd  
`col` element (XHTML)  
`colgroup` element (XHTML)  
collaboration diagram in the UML 2nd  
collaboration in the UML 2nd 3rd  
collection classes  
colon (:) 2nd 3rd 4th  
`cols` attribute of element `textarea` (XHTML)  
`colspan` attribute of element `th` (XHTML)  
column  
column headings  
column subscript  
com (top-level domain)  
combining Class Time and Class Date exercise  
combining control statements in two ways  
comma operator (,) 2nd 3rd  
comma-separated list 2nd 3rd 4th

- of base classes
- of parameters

command line  
command-and-control software system  
command-line argument 2nd  
command-line prompt  
comment 2nd 3rd  
commercial data processing  
commission worker  
`CommissionEmployee` class header file  
`CommissionEmployee` class implementation file  
`CommissionEmployee` class represents an employee paid a percentage of gross sales  
`CommissionEmployee` class test program  
`CommissionEmployee` class uses member functions to manipulate its `private` data  
`CommissionEmployee` class with `protected` data  
Common Gateway Interface (CGI) 2nd  
Common Programming Error  
communication diagram in the UML 2nd 3rd  
commutative  
commutative operation  
comparator function object 2nd

less 2nd

compare member function of class string

comparing

blocks of memory

iterators

strings 2nd 3rd 4th

unions

compilation error

compilation phase

compilation process for a Simple program

compilation unit

compile

compile-time error

compiler 2nd 3rd 4th 5th

compiler dependent

compiler error

compiler generates SML instruction

compiler optimization

compiler option

compiling 2nd 3rd

multiple-source-file program 2nd

compl operator keyword

complement operator (~)

Complex class 2nd

exercise

member-function definitions

complex conditions

complex numbers 2nd

component

component diagram in the UML

component in the UML 2nd

component-oriented software development

components

composite structure diagram in the UML

composition 2nd 3rd 4th 5th 6th 7th

compound interest 2nd 3rd 4th

calculation with for 2nd

compound statement 2nd

computation

computer

Computer Assisted Instruction exercise

Computer name: field

computer network 2nd  
computer networking  
computer program  
computer programmer  
computer simulator  
Computers in Education exercise  
computing the sum of the elements of an array  
concatenate  
concatenate stream insertion operations  
concatenate strings 2nd  
concatenate two linked list objects  
concept  
concrete class 2nd  
condition 2nd 3rd 4th 5th  
conditional compilation 2nd  
conditional execution of preprocessor directives  
conditional expression 2nd 3rd 4th  
conditional operator (? :)  
conditional preprocessor directives  
conditionally compiled output statement  
confusing equality (==) and assignment (=) operators 2nd  
conserving memory  
consistent state 2nd 3rd  
const 2nd 3rd 4th 5th  
const char \*  
const keyword  
const member function  
const member function on a const object  
const member function on a non-const object  
const object 2nd 3rd  
    must be initialized  
const objects and const member functions  
const pointer 2nd  
const qualifier 2nd 3rd  
const qualifier before type specifier in parameter declaration  
const type qualifier applied to an array parameter  
const variables must be initialized  
const version of operator[ ]  
const with function parameters  
"const-ness"  
const\_cast  
    cast away const-ness

const\_cast operator 2nd 3rd  
const\_iterator 2nd 3rd 4th 5th 6th 7th 8th 9th  
const\_reference  
const\_reverse\_iterator 2nd 3rd 4th 5th  
constant  
    floating-point  
constant integral expression 2nd  
constant pointer 2nd  
    to an integer constant  
    to constant data 2nd  
    to nonconstant data 2nd  
constant reference  
constant reference parameter  
constant runtime  
constant variable 2nd  
constructed inside out  
constructor  
    called recursively  
    cannot be virtual  
    cannot specify a return type  
    conversion 2nd 3rd  
    copy  
    default  
    default arguments  
    defining  
    explicit  
    function prototype  
    in a UML class diagram  
    in a union  
    naming  
    parameter list  
    single argument 2nd 3rd 4th  
constructors and destructors called automatically  
container 2nd 3rd 4th 5th 6th  
container adapter 2nd 3rd  
container adapter functions  
    pop  
    push  
container adapters  
    priority\_queue  
    queue  
    stack

container class 2nd 3rd 4th 5th 6th 7th

containers

- begin function
- clear function
- empty function
- end function
- erase function 2nd
- max\_size function
- rbegin function
- rend function
- size function
- swap function

Content-Type header 2nd 3rd

CONTENT\_LENGTH environment variable

continue command (GNU C++ debugger)

Continue command (Visual C++ .NET debugger)

continue statement 2nd 3rd

- terminating a single iteration of a for statement

control character

control statement 2nd 3rd 4th

- do...while 2nd 3rd 4th 5th

- for 2nd 3rd 4th 5th

- if 2nd 3rd

- if...else

- nested if...else

- nesting 2nd 3rd

- repetition statement

- selection statement

- sequence statement

- stacking 2nd 3rd

- switch 2nd 3rd

- while 2nd 3rd 4th 5th

control structure

control variable

- name

controlling expression (in a switch)

controlling the printing of trailing zeros and decimal points for doubles

converge on the base case (recursion)

conversational computing

conversion between a fundamental type and a class

conversion constructor 2nd 3rd 4th

conversion operator  
conversions among fundamental types  
    by cast  
convert a binary number to decimal  
convert a hexadecimal number to decimal  
convert among user-defined types and built-in types  
convert an octal number to decimal  
convert between types  
convert lowercase letters  
Converting a string to uppercase  
converting from a higher data type to a lower data type  
converting strings to C-style strings and character arrays  
cookie 2nd 3rd 4th  
copy a string using array notation  
copy a string using pointer notation  
copy constructor 2nd 3rd 4th 5th 6th  
    pass-by-value parameter passing  
copy member function of class string  
copy of the argument  
copy STL algorithm 2nd  
"copy-and-paste" approach  
copy\_backward STL algorithm 2nd 3rd  
copying strings 2nd  
Copying the Book Examples from the CD-ROM (box)  
core memory  
correct number of arguments  
correct order of arguments  
correction  
correctly initializing and using a constant variable  
cos function  
cosine  
count function of associative container  
count STL algorithm 2nd 3rd  
count\_if STL algorithm 2nd 3rd  
counter 2nd 3rd 4th  
counter variable  
counter-controlled repetition 2nd 3rd 4th 5th 6th 7th 8th 9th  
    with the for statement  
counting loop  
counting up by one  
cout (<>) (standard output stream) 2nd 3rd 4th 5th 6th 7th  
cout member function put

cout member function write  
CPU 2nd  
craps simulation 2nd 3rd 4th  
"crashing"  
create new data types 2nd 3rd  
CreateAndDestroy class  
    definition  
        member-function definitions  
creating a random access file  
Creating a random-access file with  
    blank records sequentially  
Creating a sequential file  
creating an association  
Creating and traversing a binary tree  
Credit inquiry program  
credit limit on a charge account  
credit processing program  
crossword puzzle generator  
cryptogram  
Ctrl key  
current position in a stream  
cursor

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

dangerous pointer manipulation  
dangling pointer 2nd  
dangling reference  
dangling-else problem 2nd  
data  
data abstraction 2nd 3rd  
data cells  
data hiding 2nd  
data hierarchy 2nd  
data member 2nd 3rd 4th 5th 6th  
    private  
data member function of class `string`  
data member function of `classstring`  
data persistence  
data representation  
data structure 2nd 3rd  
data structures  
data tier  
data type  
data types  
    `bool`  
    `char` 2nd  
    `double` 2nd 3rd  
    `float` 2nd  
    in the UML  
    `int`  
    `long`  
    `long double`  
    `long int` 2nd

short  
short int  
unsigned  
unsigned char  
unsigned int 2nd 3rd  
unsigned long  
unsigned long int  
unsigned short  
unsigned short int

database

database management system (DBMS)

Date class 2nd 3rd 4th

definition

definition with overloaded increment operators

exercise

member function definitions

member-function and friend-function definitions

test program

date source file is compiled

DBMS (database management system)

deallocate memory 2nd 3rd 4th

debug 2nd

Debug configuration

debugge (GNU C++)

help command

debugger 2nd

debugger (GNU C++)

-g compiler option

break command

break mode

breakpoint

continue command

defined

delete command

finish command

gdb command

info break command

inserting a breakpoint

list command

logic error

next command

print command 2nd

quit command  
run command  
set command 2nd  
step command  
suspending program execution  
watch command

## debugger (Visual C++ .NET)

Autos window displaying state of objects  
break mode  
breakpoint  
Continue command  
Debug configuration  
defined  
Inserting a breakpoint  
Locals window  
logic error  
margin indicator bar  
Solution Configurations combo box  
Step Into command  
Step Out command  
Step Over command  
suspending program execution  
Watch window (Visual C++ .NET) 2nd

debugging 2nd  
debugging aid  
debugging tool

dec stream manipulator 2nd 3rd

decimal (base 10) number system 2nd  
decimal (base-10) number system 2nd

decimal digit

decimal number 2nd 3rd

decimal point 2nd 3rd 4th 5th

decision 2nd 3rd

decision in the UML

decision symbol

deck of cards 2nd

declaration 2nd

    of a function

declarations

    using

declaring a static member function const

decrement

    a control variable

a pointer  
decrement operator -- 2nd  
    overloading  
decrypter  
deeply nested statements  
default access mode for class is `private`  
default argument 2nd 3rd  
default arguments with constructors  
default case 2nd 3rd 4th  
default constructor 2nd 3rd 4th 5th 6th 7th 8th 9th  
    provided by the compiler  
    provided by the programmer  
default delimiter  
default memberwise assignment 2nd 3rd  
default memberwise copy 2nd  
default precision  
default to decimal  
default to `public` access  
defensive programming  
define a constructor  
defining a class  
defining a member function of a class  
definite repetition  
definition  
Deitel Buzz Online newsletter  
`deitel@deitel.com`  
del element  
delay loop  
delegation  
`delete []` (dynamic array deallocation)  
delete a record from a file  
delete debugger command  
delete HTTP request type  
delete operator 2nd 3rd 4th 5th 6th  
deleting an item from a binary tree  
deleting dynamically allocated memory  
delimiter 2nd 3rd  
    with default value '\n'  
demonstrating a `mutable` data member  
demonstrating class template `Stack` 2nd  
demonstrating function `substr`

demonstrating functions `erase` and `replace`  
demonstrating input from an `istringstream` object  
demonstrating multiple inheritance  
demonstrating operator `const_cast`  
demonstrating string assignment and concatenation  
demonstrating the `.*` and `->*` operators  
demonstrating the operator keywords  
demonstrating the `string` `find` member functions  
demonstrating the `string` `insert` functions  
demonstrating the use of namespaces  
DeMorgan's laws  
Department of Defense (DOD)  
deployment diagram in the UML  
Deposit class (ATM case study) 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th  
DepositSlot class (ATM case study) 2nd 3rd 4th 5th 6th  
deque class 2nd

- push\_front function
- sequence container

dequeue  
dequeue operation  
dereference

- a `const` iterator
- a null pointer
- a pointer 2nd 3rd
- an iterator 2nd
  - an iterator positioned outside its container

dereferencing operator `(*)`  
derive one class from another  
derived class 2nd 3rd 4th 5th

- catch derived-class exceptions
- destructor
- indirect

descriptive words and phrases (OOD/UML case study) 2nd  
design process 2nd 3rd 4th 5th  
design specification  
destructive write  
destructor 2nd 3rd

- called in reverse order of constructors 2nd
- in a derived class
- overloading
  - receives no parameters and returns no value

dethread a node from a list

developer.intel.com/software/products/compilers/cwin/index.htm  
diagnostics that aid program debugging  
dialog  
diamond 2nd  
diamond inheritance  
diamond symbol (UML) 2nd  
dice game  
Die-rolling program using an array instead of switch  
difference\_type  
digit 2nd 3rd  
direct access  
direct base class  
directly reference a value  
disabling a breakpoint  
disc (XHTML)  
disk 2nd 3rd  
disk drive  
disk file  
disk I/O completion  
disk space 2nd 3rd  
displacement  
display screen 2nd  
distributed client/server applications  
distributed computing  
divide by zero 2nd  
divide-and-conquer approach 2nd  
DivideByZeroException  
divides function object  
division 2nd 3rd  
division by zero is undefined  
DNS (domain name system)  
DNS lookup  
do...while repetition statement 2nd 3rd 4th 5th  
document a program  
dollar amount  
domain  
domain name  
domain name system (DNS) server  
dot operator (.) 2nd 3rd 4th 5th  
dotted line  
double data type 2nd 3rd 4th  
double quote 2nd

double-array subscripting  
double-ended queue  
double-precision floating-point number  
double-selection statement 2nd 3rd  
double-word boundary  
"doubly initializing" member objects  
doubly linked list 2nd 3rd  
downcasting 2nd  
driver program  
dummy value  
duplicate elimination 2nd 3rd  
duplicate keys 2nd  
duplicate node values  
dynamic array  
dynamic binding 2nd 3rd 4th 5th 6th  
dynamic cast  
dynamic content  
dynamic data structure 2nd  
dynamic memory  
dynamic memory allocation 2nd 3rd  
dynamic memory management  
dynamic vs. static Web content  
dynamic Web content 2nd  
dynamic\_cast operator 2nd  
dynamically allocated memory 2nd 3rd 4th 5th 6th 7th  
    allocate and deallocate storage  
    array of characters  
    array of integers 2nd  
    for an array  
dynamically determine function to execute

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

e-mail (electronic mail)

e-mail anchor

EBCDIC (Extended Binary Coded Decimal Interchange Code)

edit

edit a file

edit phase

editor

edu (top-level domain)

efficiency of

- binary search
- bubble sort
- insertion sort
- linear search
- merge sort
- selection sort

Eight Queens

- brute force approaches

element of a string

element of an array

elided UML diagram

ellipsis (...) in a function prototype

emacs

emacs text editor

embedded parentheses

Employee class 2nd

- definition showing composition

- definition with a static data member to track the number of Employee objects in memory

- Exercise

- header file

- hierarchy driver program

implementation file  
member-function definitions  
member-function definitions, including constructor with a member-initializer list  
empty element (XHTML)  
empty exception specification  
empty member function  
  of a sequence container  
  of containers  
  of priority\_queue  
  of queue  
  of stack  
  of string 2nd  
empty parameter list  
empty parentheses 2nd 3rd  
empty quotation marks  
empty space  
empty statement  
empty string 2nd 3rd  
encapsulation 2nd 3rd 4th 5th  
encrypted integer  
encrypter  
encryption 2nd 3rd  
end iterator  
end line  
end member function of class string  
end member function of containers  
end member function of first-class containers  
end of a sequence  
end of a stream  
end of a string  
"end of data entry"  
end tag (XHTML)  
end-of-file 2nd 3rd 4th 5th 6th  
  key combination 2nd  
endl stream manipulator 2nd  
English-like abbreviations  
Enhancing Class Date exercise  
Enhancing Class Rectangle exercise  
Enhancing Class Time exercise 2nd  
enqueue function  
enqueue operation  
Enter key 2nd 3rd

entity reference (XHTML)

entry point

enum keyword

enumeration 2nd

enumeration constant 2nd

environment

environment variable

CONTENT\_LENGTH

HTTP\_COOKIE

QUERY\_STRING

EOF 2nd 3rd

eof stream member function 2nd 3rd

eofbit of stream

equal STL algorithm 2nd

equal to

equal\_range function of associative container

equal\_range STL algorithm 2nd

equal\_to function object

equality and relational operators

equality operator (==) 2nd 3rd 4th 5th

equality operators (== and !=) 2nd

equation of straight line

erase member function

  of class string 2nd

  of containers 2nd

Erroneous attempt to initialize a constant of a built-in data type by assignment

error

  off-by-one 2nd 3rd 4th

error bits

error checking

error detected in a constructor

error message

error state of a stream 2nd 3rd

Error-Prevention Tip

error-processing code

escape character

escape early from a loop

escape sequence 2nd

  \" (double-quote character)

  \' (single-quote character)

  \\ (backslash character)

  \`a (alert)

\n (newline)  
\r (carriage return)  
\t (tab) 2nd

et (top-level domain)

evaluating a postfix expression

evaluating expressions 2nd

event

examination-results problem

Examples

Accessing an object's members through each type of object handle

Addition program that displays the sum of two numbers

Aiming a derived-class pointer at a base-class object

Algorithms `equal`, `mismatch` and `lexicographical_compare`

Algorithms `min` and `max`

Algorithms `swap`, `iter_swap` and `swap_ranges`

Array class definition with overloaded operators 2nd

Array class member-function and `friend`-function definitions

Array class test program

Array of pointers to functions

Attempting to call a multiply inherited function polymorphically

Attempting to modify a constant pointer to constant data

Attempting to modify a constant pointer to nonconstant data

Attempting to modify data through a nonconstant pointer to constant data

`auto_ptr` object manages dynamically allocated memory

Bank account program

Bar chart printing program

`BasePlusCommissionEmployee` class header file

`BasePlusCommissionEmployee` class implementation file

`BasePlusCommissionEmployee` class represents an employee who receives a base salary in

addition to a commission

`BasePlusCommissionEmployee` class test program

`BasePlusCommissionEmployee` class that inherits from class `CommissionEmployee`, which does not provide protected data

`BasePlusCommissionEmployee` class that inherits protected data from `CommissionEmployee`

Basic searching and sorting algorithms of the Standard Library

Binary function object

Bitwise AND, bitwise inclusive-OR, bitwise exclusive-OR and bitwise complement operators

Bitwise shift operators

`break` statement exiting a `for` statement

C++ Standard Library class `vector`

Cascading member function calls 2nd

CGI script that allows users to buy a book

Character arrays processed as strings  
Character-handling functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`  
Character-handling functions `islower`, `isupper`, `tolower` and `toupper`  
Character-handling functions `isspace`, `iscntrl`, `ispunct`, `isprint` and `isgraph`  
characters in length  
Class `bitset` and the Sieve of Eratosthenes  
Class `DivideByZeroException` definition  
Class `Integer` definition  
CommissionEmployee class header file  
CommissionEmployee class implementation file  
CommissionEmployee class represents an employee paid a percentage of gross sales  
CommissionEmployee class test program  
CommissionEmployee class uses member functions to manipulate its private data  
CommissionEmployee class with protected data  
Comparing strings  
Complex class definition  
Complex class member-function definitions  
Complex numbers  
Complex XHTML table  
Compound interest calculations with `for`  
Computing the sum of the elements of an array  
`const` objects and `const` member functions  
`const` type qualifier applied to an array parameter  
`const` variables must be initialized  
Constructor with default arguments  
`contact.html` 2nd  
continue statement terminating a single iteration of a `for` statement  
Controlling the printing of trailing zeros and decimal points for doubles  
Converting a string to uppercase  
Converting strings to C-style strings and character arrays  
Copying the Examples folder  
Correctly initializing and using a constant variable  
Counter-controlled repetition with the `for` statement  
Craps simulation  
`CreateAndDestroy` class definition  
`CreateAndDestroy` class member-function definitions  
Creating a random-access file with 100 blank records sequentially  
Creating a sequential file  
Creating a server-side file to store user data  
Creating and manipulating a `GradeBook` object in which the course name is limited to  
Creating and traversing a binary tree

Credit inquiry program  
Date class definition  
Date class definition with overloaded increment operators  
Date class member function definitions  
Date class member-function and friend-function definitions  
Date class test program  
Default arguments to a function  
Default memberwise assignment  
Defining and testing class GradeBook with a data member and set and get functions  
Defining class GradeBook with a member function that takes a parameter  
Defining class GradeBook with a member function, creating a GradeBook object and calling its member function  
Demonstrating a mutable data member  
Demonstrating class template Stack 2nd  
Demonstrating copy\_backward, merge, unique and reverse  
Demonstrating function substr  
Demonstrating functions erase and replace  
Demonstrating inplace\_merge, unique\_copy and reverse\_copy  
Demonstrating input from an istringstream object  
Demonstrating lower\_bound, upper\_bound and equal\_range  
Demonstrating multiple inheritance  
Demonstrating operator const\_cast  
Demonstrating set\_new\_handler  
Demonstrating Standard Library functions fill, fill\_n, generate and generate\_n  
Demonstrating Standard Library functions remove, remove\_if, remove\_copy and remove\_copy\_if  
Demonstrating Standard Library functions replace, replace\_if, replace\_copy and replace\_copy\_if  
Demonstrating string assignment and concatenation  
Demonstrating the .\* and ->\* operators  
Demonstrating the operator keywords  
Demonstrating the string find member functions  
Demonstrating the string insert functions  
Demonstrating the use of namespaces  
Die-rolling program using an array instead of switch  
do...while repetition statement  
Employee class definition showing composition  
Employee class definition with a static data member to track the number of Employee objects in memory  
Employee class header file  
Employee class hierarchy driver program

Employee class implementation file

Employee class member function definitions, including constructor with a member-initializer list

Employee class member-function definitions

Equality and relational operators

Erroneous attempt to initialize a constant of a built-in data type by assignment

Exception-handling example that throws exceptions on attempts to divide by zero

First CGI script

flags member function of `ios_base`

Floating-point values displayed in default, scientific and fixed format

Form including radio buttons and drop-down lists

Friends can access private members of class

Functions that take no arguments

Generating values to be placed into elements of an array

get, put and `eof` member functions

GradeBook class definition containing function prototypes that specify the interface of the class

GradeBook class demonstration after separating its interface from its implementation

HourlyEmployee class header file

HourlyEmployee class implementation file

HTML/XHTML special characters

Huge integers

Implementation class definition

Implementing a proxy class

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

Inheritance examples

Inheritance hierarchy for university `CommunityMembers`

Initializing a reference

Initializing an array's elements to zeros and printing the array

Initializing multidimensional arrays

Initializing the elements of an array with a declaration

inline function to calculate the volume of a cube

Input and output stream iterators

Input of a string using `cin` with stream extraction contrasted with input using `cin.get`

Inputting character data using `cin` member function `getline`

Inserting special characters into XHTML

Instantiating multiple objects of the GradeBook class and using the GradeBook constructor to specify the course name when each GradeBook object is created

Interactive portal handler

Interactive portal to create a password-protected Web page

Interface class definition

Interface class member-function definitions

Iterative factorial solution

Left justification and right justification with stream manipulators `left` and `right`

Linear search of an array  
Linking to an e-mail address  
Linking to other Web pages  
List class-template definition  
ListNode class-template definition  
Logout program for the shopping cart example  
Manipulating a linked list  
Mathematical algorithms of the Standard Library  
Member function definition of class Integer  
Member initializer used to initialize a constant of a built-in data type  
Member-function definitions for class GradeBook with a set function that validates the length of data member courseName  
Member-object initializers  
Memory-handling function `memchr`  
Memory-handling function `memcmp`  
Memory-handling function `memcpy`  
Memory-handling function `memmove`  
Memory-handling function `memset`  
Multipurpose sorting program using function pointers  
Name mangling to enable type-safe linkage  
Nested and ordered lists in XHTML  
Nested control statements: Examination-results problem  
new throwing `bad_alloc` on failure  
Non-friend/non-member functions cannot access `private` members  
Overloaded function definitions  
Overloaded stream insertion and stream extraction operators  
Pass-by-reference with a pointer argument used to cube a variable's value  
Pass-by-value used to cube a variable's value  
Passing arguments by value and by reference  
Passing arrays and individual array elements to functions  
Placing images in XHTML files  
Pointer operators & and \*

Poll analysis program  
Precision of floating-point values  
Preincrementing and postincrementing  
Printing a line of text with multiple statements  
Printing a string one character at a time using a nonconstant pointer to constant data  
Printing an integer with internal spacing and plus sign  
Printing an unsigned integer in bits  
Printing multiple lines of text with a single statement  
Printing `string` characteristics  
Printing the address stored in a `char * variable`

Printing the value of a union in both member data types  
private base-class data cannot be accessed from derived class  
Program that outputs a login page  
protected base-class data can be accessed from derived class  
Queue class-template definition  
Queue-processing program  
Randomizing the die-rolling program  
Reading a random-access file sequentially  
Reading and printing a sequential file  
Reading cookies from the client's computer  
Reading input from `QUERY_STRING`  
Referencing array elements with the array name and with pointers  
Rethrowing an exception  
Retrieving environment variables via function `getenv`  
Returning a reference to a private data member  
Rolling a six-sided die 6000 times  
SalariedEmployee class header file  
SalariedEmployee class implementation file  
SalesPerson class definition  
SalesPerson class member-function definitions  
Scoping example  
Selection sort with call-by-reference  
Set of recursive calls to method `Fibonacci`  
set operations of the Standard Library  
`set_new_handler` specifying the function to call when `new` fails  
Shifted, scaled integers produced by `1+rand() % 6`  
Signals defined in header `<csignal>`  
Simple form with hidden fields and a text box  
`sizeof` operator used to determine standard data type sizes  
`sizeof` operator when applied to an array name returns the number of bytes in the array  
Stack class-template definition  
Stack class-template definition with a composed `List` object  
Stack test program  
Stack unwinding  
Standard Library class `string`  
Standard Library `deque` class template  
Standard Library `list` class template  
Standard Library `map` class template  
Standard Library `multimap` class template  
Standard Library `multiset` class template  
Standard Library `priority_queue` adapter class

Standard Library queue adapter class templates  
Standard Library set class template  
Standard Library stack adapter class  
Standard Library vector class template  
`static` array initialization and automatic array initialization  
`static` data member tracking the number of objects of a class 2nd  
`strcat` and `strncat`  
`strcmp` and `strncmp`  
`strcpy` and `strncpy`  
Stream manipulator `showbase`  
Stream manipulators `boolalpha` and `noboolalpha`  
Stream manipulators `hex`, `oct`, `dec` and `setbase`  
String class definition with operator overloading  
String class member-function and `friend`-function definitions  
String class test program  
String copying using array notation and pointer notation  
String-conversion function `atof`  
String-conversion function `atoi`  
String-conversion function `atol`  
String-conversion function `strtod`  
String-conversion function `strtol`  
String-conversion function `strtoul`  
String-search function `strchr`  
String-search function `strcspn`  
String-search function `strpbrk`  
String-search function `strrchr`  
String-search function `strspn`  
String-search function `strstr`  
`strlen`  
`strtok`  
Summing integers with the `for` statement  
Testing error states  
Text-printing program  
`this` pointer used implicitly and explicitly to access members of an object  
Time class containing a constructor with default arguments  
Time class definition  
Time class definition modified to enable cascaded member-function calls  
Time class member function definitions, including `const` member functions  
Time class member-function definitions  
Time class member-function definitions including a constructor that takes arguments  
Time class with `const` member functions

Tree class-template definition  
TreeNode class-template definition  
Two-dimensional array manipulations  
Unary scope resolution operator  
Unformatted I/O using the `read`, `gcount` and `write` member functions  
Uninitialized local reference causes a syntax error  
Unordered lists in XHTML  
User-defined `maximum` function  
User-defined, nonparameterized stream manipulators  
Using a dynamically allocated `ostringstream` object  
Using a function template  
Using an anonymous `union`  
Using an iterator to output a `string`  
Using command-line arguments  
Using function `swap` to swap two `strings`  
Using functions `exit` and `atexit`  
Using GET with an XHTML form  
Using `goto`  
Using images as link anchors  
Using member function `fill` and stream manipulator `setfill` to change the padding character for fields larger than the values being printed  
Using member functions `get`, `put` and `eof`  
Using POST with an XHTML form  
Using signal handling  
Using Standard Library functions to perform a heapsort  
Using stream manipulator `uppercase`  
Using template functions  
Using variable-length argument lists  
Using `virtual` base classes  
Utility function demonstration  
`vector` class template element-manipulation functions  
width member function of class `ios_base`  
Writing a cookie  
XHTML document containing a form to post data to the server  
XHTML document to read user's contact information  
XHTML table  
exception  
exception class 2nd  
    what virtual function  
exception classes derived from common base class  
exception handler  
exception handling 2nd

exception not listed in exception specification

exception object

exception parameter

exception specification

Exception-handling example that throws exceptions on attempts to divide by zero

exceptional condition

Exceptions

[bad\\_alloc](#)

[bad\\_cast](#)

[bad\\_exception](#)

[bad\\_typeid](#)

[length\\_error](#)

[logic\\_error](#)

[out\\_of\\_bounds](#)

[out\\_of\\_range](#) 2nd 3rd 4th

[overflow\\_error](#)

[runtime\\_error](#) 2nd 3rd 4th

[underflow\\_error](#)

executable image

executable statement 2nd

execute a program 2nd

execution-time error

execution-time overhead

Exercises

[Account class](#)

[Account inheritance hierarchy](#)

[Bubble Sort](#) 2nd

[Bucket Sort](#)

[Celsius and Fahrenheit Temperatures](#)

[Combining Class Time and Class Date](#)

[Complex Class](#)

[Computer Assisted Instruction](#)

[Computers in Education](#)

[Date class](#)

[Employee class](#)

[Enhanced Bubble Sort](#)

[Enhancing Class Date](#)

[Enhancing Class Rectangle](#)

[Enhancing Class Time](#) 2nd

[Fibonacci Series](#)

[Guess the Number Game](#)

[HugeInt Class](#)

HugeInteger Class  
Hypotenuse  
Invoice class  
Modifying Class GradeBook  
Package inheritance hierarchy  
Package Inheritance Hierarchy  
Payroll System Modification  
Perfect Numbers  
Polymorphic banking program using Account hierarchy  
Polymorphic Screen Manager Using Shape Hierarchy  
Prime Numbers  
Quicksort  
Rational Class  
RationalNumber Class  
Rectangle Class  
Recursive Binary Search  
Recursive Linear Search  
Returning Error Indicators from Class Time's set Functions  
Reverse Digits  
Shape hierarchy  
TicTacToe Class  
exhaust memory  
exit a deeply nested structure  
exit a function  
exit a loop  
exit function 2nd 3rd 4th 5th 6th  
exit point of a control statement  
EXIT\_FAILURE  
EXIT\_SUCCESS  
exp function  
expand a macro  
explicit constructor 2nd  
explicit conversion  
explicit specialization of a class template  
explicit use of the this pointer  
exponent  
exponential "explosion" of calls  
exponential complexity  
exponential function  
exponentiation 2nd  
expression 2nd 3rd 4th 5th  
extend the base programming language

Extended Binary Coded Decimal Interchange Code (EBCDIC)

extensibility

extensibility of C++

extensibility of STL

Extensible HyperText Markup Language (XHTML)

eXtensible HyperText Markup Language (XHTML) 2nd 3rd

extensible language 2nd 3rd 4th 5th

`extern "C"`

`extern` storage-class specifier 2nd 3rd

external declaration

external linkage

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

F floating-point suffix

f floating-point suffix

fabs function

face values of cards

factorial 2nd 3rd 4th 5th

fail member function

failbit of a stream 2nd 3rd 4th

false 2nd 3rd 4th 5th

FAQs 2nd

fatal error 2nd 3rd

fatal logic error 2nd

fatal runtime error

fault-tolerant programs

Fibonacci series 2nd

Fibonacci Series exercise

field

field width 2nd 3rd 4th

fields larger than values being printed

FIFO (first-in, first-out) 2nd 3rd 4th 5th

file 2nd 3rd

    as a collection of bytes

file attribute value (XHTML)

file of n bytes

file open mode 2nd

    ios::app

    ios::ate

    ios::binary 2nd 3rd

    ios::in 2nd

    ios::out

ios::trunc  
file processing 2nd 3rd  
file scope 2nd 3rd 4th 5th  
file server  
file system directory  
file-position pointer 2nd 3rd  
file-processing classes  
filename 2nd  
filename extension  
.h  
fill character 2nd 3rd 4th 5th  
fill member function of ostream 2nd  
fill STL algorithm 2nd  
fill\_n STL algorithm 2nd  
final state 2nd  
final state in the UML  
final value of a control variable 2nd  
find function of associative container  
find member function of class string 2nd  
find STL algorithm 2nd 3rd  
find\_each STL algorithm  
find\_end STL algorithm  
find\_first\_not\_of member function of class string  
find\_first\_of member function of class string 2nd  
find\_first\_of STL algorithm  
find\_if STL algorithm 2nd 3rd  
find\_last\_of member function of class string  
finding strings and characters in a string  
finish debugger command  
first data member of pair  
first pass of Simple compiler 2nd 3rd 4th  
first refinement 2nd 3rd  
first-class container 2nd 3rd 4th 5th 6th  
begin member function  
clear function  
end member function  
erase function  
first-in, first-out (FIFO) 2nd 3rd 4th 5th  
fixed notation 2nd 3rd  
fixed stream manipulator 2nd 3rd  
fixed word size

fixed-point format  
fixed-point value  
flag  
flag value  
flags member function of `ios_base`  
flight simulator  
`flip` of `bitset`  
`float` 2nd  
`float` data type 2nd  
floating point 2nd 3rd  
floating-point arithmetic  
floating-point constant  
floating-point constant not suffixed  
floating-point division  
floating-point exception  
floating-point literal  
    `double` by default  
floating-point number 2nd  
    `double` data type  
    `double` precision  
    `float` data type  
    single precision  
floating-point number in scientific format  
floating-point size limits  
floating-point values displayed in default, scientific and fixed format  
`floor`  
`floor` function  
flow of control 2nd  
flow of control in the `if...else` statement  
flow of control of a virtual function call  
flush a stream  
flush buffer  
flush output buffer  
flushing stream  
`fmod` function  
`fmtflags` data type  
`for` map  
`for` repetition statement 2nd 3rd 4th 5th  
`for` repetition statement examples  
`for_each` STL algorithm 2nd  
force a decimal point  
forcing a plus sign

form  
form (XHTML) 2nd  
form element  
form feed ('\f') 2nd  
form handler  
form XHTML element (<form>...</form>)  
formal parameter  
formal type parameter  
format error  
format of floating-point numbers in scientific format  
format state 2nd  
format-state stream manipulators  
formatted data file processing  
formatted I/O  
formatted input/output  
formatted text  
formatting  
formulating algorithms 2nd  
FORTRAN (FORmula TRANslator)  
Fortran (FORmula TRANslator) programming language  
forward class declaration  
forward declaration  
forward iterator 2nd 3rd 4th 5th 6th 7th  
forward iterator operations  
forward pointer  
forward reference  
forward slash (/)  
FQDN (fully qualified domain name)  
fractional parts  
fractions  
fragile software  
free  
free memory  
free store  
friend 2nd 3rd  
friend function 2nd 3rd 4th  
friend functions to enhance performance  
friend of a derived class  
friend of class template  
friends are not member functions  
Friends can access private members of class  
friendship granted, not taken

friendship not symmetric  
friendship not transitive  
front member function of queue  
front member function of sequence containers 2nd  
front of a queue  
front\_inserter function template  
fstream 2nd 3rd 4th 5th  
Full computer name: field  
fully qualified domain name (FQDN)  
function 2nd 3rd 4th 5th  
    argument  
    declaration (prototype)  
    definition 2nd 3rd  
    empty parameter list 2nd  
    empty parentheses 2nd 3rd  
    header 2nd  
    local variable  
    multiple parameters  
    name 2nd 3rd 4th  
    no arguments  
    overloading 2nd 3rd  
    parameter 2nd  
    parameter as a local variable  
    parameter list  
    prototype 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th  
    return a result  
    scope  
    signature 2nd  
        that calls itself  
function body  
function call 2nd 3rd  
    operator ( ) 2nd  
    overhead 2nd  
    stack 2nd  
function object 2nd 3rd  
    binary  
    less<T>  
    predefined in the STL  
function objects  
    divides  
    equal\_to  
    greater

greater\_equal  
less  
less<int>  
less<T>  
less\_equal  
logical\_end  
logical\_not  
logical\_or  
minus  
modulus  
multiplies  
negate  
not\_equal\_to  
plus  
function overhead  
function pointer 2nd 3rd 4th  
function prototype  
    mandatory  
    parameter names optional  
    scope  
    semicolon at end  
function template 2nd 3rd 4th  
    definition  
    max  
    min  
    specialization 2nd  
functional structure of a program  
functions for manipulating data in the standard library containers  
fundamental type

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

gallery.yahoo.com  
game of "guess the number"  
game of chance  
game of craps 2nd  
game playing 2nd  
"garbage" value  
gcd  
gcount function of `istream`  
gdb command  
general class average problem  
general utilities library `<cstdlib>` 2nd 3rd 4th  
generalities  
generalization in the UML  
generate STL algorithm 2nd  
generate\_n STL algorithm 2nd  
generating mazes of any size  
generating mazes randomly  
generating values to be placed into elements of an array  
generator function  
generic algorithms  
generic class  
generic programming 2nd 3rd 4th 5th 6th  
get a value  
get and set functions  
get function  
GET HTTP method  
get HTTP request type  
get member function of `istream`  
get pointer

get request type  
get, put and eof member functions  
getenv function 2nd 3rd 4th  
getline function of cin 2nd  
getline function of the string header file 2nd  
getline member function of class string  
gets the value of  
global function 2nd  
global function to overload an operator  
global identifier  
global namespace  
global object constructors  
global scope 2nd  
global variable 2nd 3rd 4th 5th 6th 7th  
global variable name  
global, friend function  
global, non-friend function  
golden mean  
golden ratio  
good function of ios\_base  
Good Programming Practices 2nd  
goodbit of stream  
gosub  
goto elimination  
goto statement 2nd 3rd  
goto-less programming  
grade-point average  
graph  
graph information  
graphical representation of a binary tree  
Graphics Interchange Format (GIF)  
graphics package  
greater function object  
greater-than operator  
greater-than-or-equal-to operator  
greater\_equal function object  
greatest common divisor (GCD) 2nd  
gross pay  
group of related fields  
guard condition in the UML 2nd 3rd 4th 5th 6th 7th 8th  
Guess the Number Game exercise  
guillemets (« and ») in the UML

 PREVNEXT 

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

h1 header element

h2 header element 2nd

h3 header element 2nd

h4 header element

h5 header element

h6 header element

half word 2nd

handle on an object

hangman

hard disk 2nd

hardcopy printer

hardware 2nd 3rd

hardware platform

has-a relationship (composition) 2nd 3rd

head

head element (XHTML)

head HTTP request type

head of a queue 2nd

head section

header (XHTML)

header cell (XHTML table)

header element

header elements

    h1

    h2 2nd

    h3 2nd

    h4

    h5

    h6

header elements (XHTML)

header file 2nd 3rd 4th 5th 6th 7th 8th 9th

<algorithm>  
<deque>  
<exception> 2nd 3rd 4th  
<fstream> 2nd  
<functional> 2nd 3rd  
<iomanip.h>  
<iomanip> 2nd 3rd 4th  
<iostream> 2nd 3rd 4th 5th 6th 7th 8th  
<iterator> 2nd 3rd  
<limits>  
<list> 2nd 3rd  
<locale>  
<map> 2nd  
<memory> 2nd  
<new>  
<numeric> 2nd  
<queue> 2nd 3rd 4th  
<set> 2nd 3rd 4th  
<sstream> 2nd  
<stack> 2nd 3rd  
<stdexcept> 2nd 3rd  
<string> 2nd 3rd 4th 5th  
<typeinfo> 2nd  
<vector>  
location  
name enclosed in angle brackets (<>)  
name enclosed in quotes (" ")

header file <csignal>

header file <stdexcept>

heap 2nd 3rd 4th

heapsort 2nd

heapsort sorting algorithm

height attribute of element img

help debugger command

help-site.com

helper function

heuristic

Hewlett-Packard

hex stream manipulator 2nd 3rd

hexadecimal 2nd

base 16 number system 2nd 3rd 4th 5th 6th

integer

notation

number system

hidden attribute value (`type`)

hidden inputs

hide an internal data representation

hide implementation

hide implementation details 2nd

hide names in outer scopes

hide private data from clients

hiding

hierarchical boss function/worker function relationship

hierarchical boss method/worker method relationship

hierarchy of exception classes

hierarchy of shapes

high-level I/O

high-level language

highest level of precedence

"highest" type

horizontal rule

horizontal rule (XHTML)

horizontal tab ('\t') 2nd 3rd

host

host environment

host object

[hotwired.lycos.com/webmonkey/00/50/index2a.html](http://hotwired.lycos.com/webmonkey/00/50/index2a.html)

HourlyEmployee class header file

HourlyEmployee class implementation file

`hr` element (XHTML)

`href` attribute of element `a`

htdocs directory

HTML (HyperText Markup Language) 2nd

HTML document

`html` element (XHTML)

HTML list

HTML table

HTML/XHTML special characters

HTTP (Hypertext Transfer Protocol) 2nd 3rd

authentication information

HTTP (version 1.1)

[HTTP header](#) 2nd

[HTTP method](#)

[HTTP request types](#)

[delete](#)

[head](#)

[options](#)

[put](#)

[trace](#)

[HTTP transaction](#)

[HTTP\\_COOKIE environment variable](#)

[HTTPd Web server \(Apache\)](#)

[httpd.apache.org/docs-2.0/](http://httpd.apache.org/docs-2.0/)

[Huge integers exercise](#)

[HugeInt class exercise](#)

[HugeInteger Class exercise](#)

[hyperlink](#) 2nd 3rd

[hypertext](#)

[HyperText Markup Language \(HTML\)](#) 2nd

[Hypertext Transfer Protocol \(HTTP\)](#) 2nd

[hypotenuse](#) 2nd 3rd

[exercise](#)

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

IBM Corporation 2nd

IBM Personal Computer

Identification tab in the Network dialog

identifier 2nd 3rd

    for a variable name

if single-selection statement 2nd 3rd 4th 5th 6th

    activity diagram

if...else double-selection statement 2nd 3rd 4th

ifstream 2nd 3rd 4th 5th 6th

    constructor function

ignore function of istream 2nd

illegal instruction

image attribute value (type)

image hyperlink

image/gif MIME type 2nd

images in Web pages

img element

img element (XHTML)

Implementation class definition

implementation file

implementation inheritance

implementation of a member function changes

implementation phase

implementation process 2nd

Implementing a proxy class

implicit compiler-defined conversion between fundamental types

implicit conversion 2nd 3rd 4th 5th

    via conversion constructors

implicit first argument

implicit handle  
implicit pointer  
implicit, user-defined conversions  
implicitly *virtual*  
imprecision of floating-point numbers  
improper implicit conversion  
improve performance of bubble sort  
in-memory formatting  
in-memory I/O  
includes STL algorithm  
including a header file multiple times  
increment  
    a control variable 2nd 3rd  
    a pointer  
    an iterator  
increment operator (++) 2nd  
increment operators  
increment the instruction counter  
indefinite postponement  
indefinite repetition  
indentation 2nd 3rd 4th 5th  
independent software vendor (ISV) 2nd 3rd 4th  
index  
indexed access  
indexed list  
indirect base class 2nd  
indirect derived class  
indirection  
indirection operator (\*) 2nd  
indirectly reference a value  
inequality  
    operator (!=)  
    operator keywords  
infinite loop 2nd 3rd 4th 5th  
infinite recursion  
infix arithmetic expression  
infix notation  
infix-to-postfix conversion algorithm 2nd 3rd  
infix-to-postfix converter  
info break debugger command  
information hiding 2nd 3rd  
information tier

inherit implementation

inherit interface 2nd

inherit members of an existing class

inheritance 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th

examples

hierarchy 2nd

Inheritance

    hierarchy for university CommunityMembers

inheritance

    implementation vs. interface inheritance

    multiple 2nd 3rd 4th 5th 6th 7th 8th

    relationships of I/O classes 2nd

    virtual base class

inheriting interface versus inheriting implementation

initial state in the UML 2nd 3rd 4th

initial value of a control variable 2nd

initial value of an attribute

initialization phase

initialize a constant of a built-in data type

initialize a pointer

initialize pointer to 0 (null)

initialize to a consistent state

initialize with an assignment statement

initializer

initializer list 2nd 3rd

initializing a pointer declared const

initializing a reference

initializing an array's elements to zeros and printing the array

initializing multidimensional arrays

initializing the elements of an array with a declaration

inline function 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th

    calculate the volume of a cube

inline keyword 2nd

inner block

inner\_product STL algorithm

innermost pair of parentheses

inorder traversal 2nd

inOrderTraversal

inplace\_merge STL algorithm

input a line of text

Input and output stream iterators

input data

input device  
input element (XHTML)  
input from string in memory  
input iterator 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th  
input line of text into an array  
input of a string using `cin` with stream extraction contrasted with input using `cin.get`  
input sequence  
input Simple command 2nd  
input stream 2nd  
input stream iterator  
input stream object (`cin`) 2nd  
input unit  
input XHTML element  
input/output (I/O) 2nd  
input/output library functions  
input/output of objects  
input/output operations  
input/output stream header file `<iostream>`  
inputting character data using `cin` member function `getline`  
inputting from strings in memory  
insert function of associative container 2nd  
insert member function of class `string`  
insert member function of sequence container  
inserter function template  
inserting a breakpoint  
insertion 2nd  
insertion at back of vector  
insertion sort algorithm 2nd  
instance of a class  
instant access processing  
instant-access application  
instantiated  
instruction  
instruction counter  
instruction execution cycle  
int 2nd 3rd  
int &  
int operands promoted to double  
integer 2nd 3rd  
integer arithmetic  
integer division 2nd  
integer promotion

integerPower  
integers prefixed with 0 (octal)  
integers prefixed with 0x or 0X (hexadecimal)  
IntegerSet class  
integral size limits  
integrity of an internal data structure  
interaction diagram in the UML  
interaction overview diagram in the UML  
interactions among objects 2nd  
interactive attention signal  
interactive computing  
interactive signal  
interchangeability of arrays and pointers  
interest on deposit  
interest rate 2nd  
interface 2nd 3rd  
    inheritance  
Interface class definition  
Interface class member-function definitions  
interface inheritance  
interface of a class  
internal character string  
internal linkage  
internal representation of a string  
internal spacing  
    internal stream manipulator 2nd 3rd  
International Organization for Standardization (ISO)  
International Standards Organization (ISO) 2nd  
Internet 2nd  
Internet Explorer 2nd  
Internet Explorer (IE) 2nd  
Internet Protocol (IP) address  
Internet Service Provider (ISP)  
Internet STL resources  
interpreter  
interrupt  
interrupt handler  
Intranet  
intToFloat  
invalid access to storage  
invalid\_argument class  
invalid\_argument exception

I

Invoice class exercise

invoke a method

invoking a non-const member function on a const object

ios::app file open mode

ios::ate file open mode

ios::beg seek direction

ios::binary file open mode 2nd 3rd

ios::cur seek direction

ios::end seek direction

ios::in file open mode 2nd 3rd

ios::out file open mode 2nd

ios::trunc file open mode

ios\_base base class

ios\_base class

precision function

width member function

IP (Internet Protocol)

address

is-a relationship (inheritance) 2nd 3rd 4th 5th

isalnum function 2nd

isalpha function 2nd

iscntrl function 2nd

isdigit function 2nd 3rd

isgraph function 2nd

islower function 2nd 3rd

ISO

ISP (Internet Service Provider)

isprint function 2nd

ispunct function 2nd

isspace function 2nd

istream 2nd 3rd 4th 5th 6th 7th

member function ignore

istream class

peek function

seekg function

tellg function

istream\_iterator

istringstream 2nd

isupper function 2nd

ISV

isxdigit function 2nd

iter\_swap STL algorithm 2nd 3rd

iterating

iteration 2nd 3rd

  of a loop

Iterative factorial solution

iterative model

iterative solution 2nd

iterator

iterator 2nd 3rd 4th

iterator

iterator

iterator 2nd 3rd 4th 5th 6th

iterator

  ++ operator

  class 2nd

  object

  operations

  pointing to first element past the end of container

  pointing to the first element of the container

iterator typedef

iterator-category hierarchy

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]

Jacobson, Ivar

Jacopini, G. 2nd

Japanese

Java 2nd

job

Joint Photographic Experts Group (JPEG)

justified field

[◀ PREV](#)

[NEXT ▶](#)

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

key  
key/value pair 2nd 3rd  
keyboard 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th  
keyboard input  
Keypad class (ATM case study) 2nd 3rd 4th 5th 6th 7th 8th  
keyword 2nd 3rd  
    and  
    and\_eq  
    auto  
    bitand  
    bitor  
    catch  
    class 2nd  
    compl  
    const in parameter list of function  
    enum  
    explicit  
    extern  
    inline 2nd  
    mutable  
    namespace 2nd  
    not  
    not\_eq  
    or  
    or\_eq  
    private  
    public 2nd  
    return  
    static

table of keywords

template

throw

try

typedef 2nd 3rd 4th 5th 6th 7th 8th

typename 2nd

void

xor

xor\_eq

KIS ("keep it simple")

Knight's Tour

brute force approaches

closed tour test

Koenig, Andrew

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

L floating-point suffix

l floating-point suffix

L integer suffix

l integer suffix

label

label element (XHTML)

label specified in a goto statement

labels in a switch structure

Lady Ada Lovelace

large object

largest element of a collection

last-in, first-out (LIFO)

    data structure 2nd 3rd

    order 2nd 3rd

late binding

leading 0

leading 0x and leading 0X

leaf node 2nd

Lee, Meng

left brace {} 2nd

left child

left justification 2nd 3rd 4th

left justification and right justification with stream manipulators left and right

left node

left side of an assignment 2nd 3rd 4th

left stream manipulator 2nd 3rd

left subtree 2nd 3rd 4th 5th

left value

left-shift assignment operator (<<)

left-shift operator (`<<`) 2nd 3rd 4th 5th 6th  
left-to-right associativity 2nd  
left-to-right evaluation 2nd  
left-to-right pass of an expression  
legacy C code  
legacy code 2nd 3rd 4th  
length member function of class `string` 2nd  
length of a string 2nd  
length of a substring  
`length_error` exception 2nd 3rd  
less function object  
less-than operator ( 2nd  
less-than-or-equal-to operator (   
`less<double>`  
`less<int>` 2nd  
less\_equal function object  
letter  
level of indentation  
level-order traversal of a binary tree 2nd  
lexicographical comparison  
lexicographical permutator  
`lexicographical_compare` STL algorithm 2nd  
`li` element (XHTML)  
licensing classes  
lifeline of an object in a UML sequence diagram  
LIFO (last-in, first-out) order 2nd 3rd 4th 5th 6th  
limerick  
line  
line break  
line number 2nd 3rd 4th  
line of communication with a file 2nd  
line of text  
linear data structure 2nd  
linear runtime  
linear search algorithm 2nd  
linear search of an array  
link 2nd 3rd 4th 5th  
link to a class's object code  
linkage 2nd  
linkage specifications 2nd  
linked list 2nd 3rd 4th 5th 6th 7th  
linked list class template

linker 2nd  
linker error  
linking 2nd  
links.html  
links2.html  
Linux 2nd  
    shell prompt  
list  
List class template 2nd 3rd 4th  
list class template 2nd 3rd  
    assign  
    merge  
    pop\_back  
    pop\_front  
    push\_front  
    remove  
    sort  
    splice  
    swap  
    unique  
List class-template definition  
list debugger command  
list processing  
list searching performance  
list sequence container  
List<STACKTYPE>  
ListNode class-template definition  
literal  
live-code approach  
load  
loading  
local area network (LAN)  
local automatic object  
local variable 2nd 3rd 4th 5th  
local Web server  
localhost 2nd  
Locals window  
localtime function 2nd  
Location header  
location in memory  
log function  
log10 function

log<sub>2</sub> n levels in a binary search tree with n elements  
logarithm  
logarithmic runtime  
logic error 2nd 3rd  
logic\_error exception  
logical AND  
logical AND (&&) operator 2nd 3rd  
logical decision  
logical negation 2nd  
logical NOT (!) operator 2nd 3rd 4th  
logical operators  
    keywords  
logical OR (| |) operator 2nd 3rd 4th 5th  
logical unit  
logical\_and function object  
logical\_not function object  
logical\_or function object  
Logo language  
long data type 2nd 3rd  
long double data type 2nd  
long int data type 2nd 3rd  
loop 2nd 3rd 4th 5th  
loop counter  
loop iterations  
loop nested within a loop  
loop-continuation condition 2nd 3rd 4th 5th 6th 7th 8th  
loop-continuation condition fails  
loopback address  
looping statement 2nd  
Lord Byron  
loss of data  
Lovelace, Ada  
low-level I/O capabilities  
lower\_bound function of associative container  
lower\_bound STL algorithm  
lowercase letter 2nd 3rd 4th 5th 6th 7th  
"lowest type"  
lvalue  
lvalue ("left value") 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th  
lvalues as rvalues



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

m-by-n array

Mac OS X

machine dependent 2nd 3rd

machine language 2nd 3rd

machine-language code 2nd

machine-language programming

Macintosh

macro 2nd 3rd 4th 5th

argument

definition

expansion

identifier

macros defined in header <cstdarg>

magic numbers

magnetic disk

magnitude

magnitude right justified

mail order house

mailto: URL

main 2nd 3rd

mainframe

maintenance of software

make utility

"make your point"

make\_heap STL algorithm

makefile

malloc

manages dynamically allocated memory

mandatory function prototypes

mangled function name

manipulating a linked list  
manipulating individual characters  
manipulator 2nd  
"manufacturing" section of the computer  
many-to-one relationship  
mapped values  
margin indicator bar  
markup  
markup language 2nd 3rd  
maroon breakpoint circle, solid  
mask  
"masked off"  
matching catch block  
math library functions 2nd 3rd 4th

ceil  
cos  
exp  
fabs  
floor  
fmod  
log  
log10  
pow  
sin  
sqrt  
tan

mathematical algorithms  
mathematical algorithms of the Standard Library  
mathematical calculation  
mathematical classes  
mathematical computations  
max STL algorithm  
max\_element STL algorithm 2nd  
max\_size member function of a string  
max\_size member function of containers  
maxheap  
maximum function  
maximum length of a string  
maximum size of a string  
maxlength attribute of element input  
mean

meaningful names

member function 2nd 3rd 4th

    implementation in a separate source-code file

member function automatically inlined

member function call 2nd

member function calls for `const` objects

member function calls often concise

member function defined in a class definition

Member function definition of class `Integer`

member functions that take no arguments

member initializer 2nd 3rd 4th 5th 6th

member initializer for a `const` data member

member initializer list 2nd 3rd 4th 5th

member initializer used to initialize a constant of a built-in data type

member object's default constructor

member selection operator `(.)` 2nd 3rd 4th

member-function

    parameter

member-function argument

member-object initializer

member-object initializers

memberwise assignment 2nd

memberwise copy

`memchr` function 2nd 3rd

`memcmp` function 2nd

`memcpy` function 2nd

`memmove` function 2nd

memory 2nd 3rd 4th

memory access violation 2nd

memory address 2nd

memory consumption

memory functions of the string-handling library

memory leak 2nd 3rd 4th 5th 6th

    prevent

memory location 2nd 3rd

memory not allocated

memory unit

memory-access violation

memory-handling function `memchr`

memory-handling function `memcmp`

memory-handling function `memcpy`

memory-handling function `memmove`

memory-handling function `memset`

`memset` function 2nd

menu driven system

merge in the UML

`merge` member function of `list`

merge sort algorithm 2nd

`merge` STL algorithm 2nd

merge symbol

merge two arrays

merge two ordered list objects

message 2nd 3rd

message (send to an object)

message in the UML 2nd 3rd

message passing in the UML

method

method (function)

`method = "get"` (XHTML)

`method = "post"` (XHTML)

`method` attribute (XHTML) 2nd

metric conversion program

MFC

Microsoft MFC (Microsoft Foundation Classes)

Microsoft Visual C++ 2nd 3rd 4th

    home page 2nd

Microsoft Windows

Microsoft Windows-based systems

Microsoft's .NET Framework Class Library

middle tier

mileage obtained by automobiles

MIME (Multipurpose Internet Mail Extensions) type

`image/gif`

`text/html`

`text/plain`

min STL algorithm

`min_element` STL algorithm 2nd

minus function object

minus sign (-) indicating private visibility in the UML 2nd

mismatch STL algorithm 2nd 3rd

mission critical situation

mission-critical computing

mixed-type expression

model

model of a software system 2nd 3rd  
modifiable  
modifiable lvalue 2nd 3rd 4th  
modifications to the Simple compiler  
modify a constant pointer  
modify address stored in pointer variable  
Modifying Class GradeBook exercise  
modularizing a program with functions  
modulus 2nd  
    operator (%) 2nd 3rd 4th 5th  
modulus function object  
monetary calculations  
monetary formats  
most derived class  
mouse  
multidimensional array 2nd 3rd  
multimap associative container  
multiple 2nd  
multiple inheritance 2nd 3rd 4th 5th 6th 7th 8th  
    demonstration  
multiple parameters to a function  
multiple-selection statement 2nd  
multiple-source file program  
    compilation and linking process  
multiple-source-file program 2nd  
multiple-statement body  
multiplication 2nd  
multiplicative operators (\*, /, %)  
multiplicity 2nd  
multiplies function object  
multiprocessor  
multiprogramming  
Multipurpose Internet Mail Extension (MIME)  
Multipurpose sorting program using function pointers  
multitasking  
multithreading  
multitier application  
Musser, David  
mutable  
    data member 2nd  
    demonstration  
    keyword  
    storage-class specifier

mutating-sequence algorithms

mutator

My Network Places

mystery recursive exercise

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

n-tier application

name attribute of element `input`

name decoration

name function of class `type_info`

name handle 2nd

name mangling

    to enable type-safe linkage

name of a control variable

name of a source file

name of a user-defined class

name of a variable 2nd

name of an array 2nd

name of an attribute

name/value pair

named constant

namespace

    alias

    global

    nested

    qualifier

    unnamed

namespace keyword 2nd

namespace member

namespace scope

naming conflict 2nd

natural language of a computer

natural logarithm

navigability arrow in the UML

NCSA (the National Center for Supercomputing Applications)

NDEBUG (preprocessor)

near container 2nd  
negate function object  
nested building blocks 2nd  
nested control statement 2nd 3rd 4th 5th 6th 7th  
    for 2nd 3rd  
    if...else 2nd 3rd  
    nesting rule  
nested element  
nested function call  
nested list  
nested message in the UML  
nested namespace  
nested namespace  
nested parentheses  
Netscape Communicator  
Network and Dialup Connections explorer  
network connection  
Network dialog  
Network Identification  
network message arrival  
network node  
new block of memory  
new failure handler 2nd  
new operator 2nd 3rd  
    calls the constructor  
    fails 2nd  
    new\_handler function  
    returning 0 on failure 2nd  
    throwing bad\_alloc on failure  
new stream manipulators  
new\_handler function  
newline ('\n') escape sequence 2nd 3rd 4th 5th 6th  
newline character 2nd  
nickname  
noboolalpha stream manipulator  
node  
non-const member function  
non-const member function called on a const object  
non-const member function on a non-const object  
Non-friend/non-member functions cannot access private members  
non-static member function 2nd 3rd  
nonconstant pointer to constant data 2nd 3rd

nonconstant pointer to nonconstant data  
noncontiguous memory layout of a deque  
nondestructive read  
nonfatal  
    error  
    logic error  
    runtime error  
nonlinear data structures  
nonlinear, two-dimensional data structure  
nonmodifiable function code  
nonmodifying sequence algorithm  
nonparameterized stream manipulator  
nonrecoverable failures  
nontype template parameter  
nonvirtual destructor  
nonzero treated as `true`  
`noshowbase` stream manipulator 2nd  
`noshowpoint` stream manipulator  
`noshowpos` stream manipulator 2nd 3rd  
`noskipws` stream manipulator  
not operator keyword  
`not_eq` operator keyword  
`not_equal_to` function object  
note  
Notepad text editor  
`nothrow` object  
`nothrow_t` type  
noun  
noun phrase in requirements document 2nd  
nouns in a system specification  
nouns in problem statement  
`nouppercase` stream manipulator 2nd  
`nth_element`  
NULL  
null character ('`\0`') 2nd 3rd 4th 5th 6th 7th 8th 9th 10th  
null pointer (0) 2nd 3rd 4th 5th 6th 7th 8th  
null statement ;)  
null string  
null-terminated string 2nd 3rd 4th  
number of arguments  
number of elements in an array  
numeric algorithm

numerical algorithms  
numerical data type limits

 PREV

NEXT 

[page footer](#)

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

O(1) time  
O(log n) time  
O(n) time  
O( $n^2$ ) time  
O(nlog n) time  
object (or instance) 2nd 3rd 4th 5th 6th 7th  
object code 2nd 3rd 4th  
object diagram in the UML  
object file  
object goes out of scope  
object handle 2nd  
object leaves scope  
Object Management Group (OMG)  
object module  
object of a derived class 2nd  
object of a derived class is instantiated  
object orientation  
object-oriented analysis and design (OOAD) 2nd  
object-oriented design (OOD) 2nd 3rd 4th 5th 6th 7th 8th 9th 10th  
object-oriented language 2nd  
object-oriented programming (OOP) 2nd 3rd 4th 5th 6th 7th 8th 9th 10th  
object's vtable pointer  
objects contain only data  
ObjectSpace STL Tool Kit examples 2nd  
ocating  
oct stream manipulator 2nd 3rd  
octal (base-8) number system 2nd 3rd 4th  
octal number 2nd 3rd 4th 5th  
odd integer  
odd number

o

off-by-one error 2nd 3rd 4th

offset

offset from the beginning of a file

offset to a pointer

ofstream 2nd 3rd 4th 5th 6th 7th 8th 9th

    open function

ofstream constructor

"old-style" header files

om (top-level domain)

OMG (Object Management Group)

one-pass algorithm

one-to-many mapping

one-to-many relationship 2nd

one-to-one mapping 2nd

one-to-one relationship

one's complement 2nd

    operator (~)

ones position

OOAD (object-oriented analysis and design) 2nd

OOD (object-oriented design) 2nd 3rd 4th 5th 6th 7th

OOP (object-oriented programming) 2nd 3rd 4th 5th 6th 7th 8th 9th

open a file for input

open a file for output

open a nonexistent file

open file

open function of ofstream

open source software

operand 2nd 3rd 4th 5th 6th

operating system 2nd 3rd 4th 5th

operation (UML)

operation code 2nd

operation compartment in a class diagram

operation in the UML 2nd 3rd 4th 5th 6th 7th 8th 9th

operation parameter in the UML 2nd 3rd 4th

operations that can be performed on data

operator

    ! (logical NOT operator) 2nd

    != (inequality operator)

    % (modulus operator)

    %= modulus assignment

    && (logical AND operator)

    \* (multiplication operator)

- o
  - \* (pointer dereference or indirection) 2nd
  - \*= multiplication assignment
  - + (addition operator) 2nd
  - += addition assignment
  - = subtraction assignment
  - / (division operator)
  - /= division assignment
  - < (less-than operator)
  - << (stream insertion operator) 2nd
  - <= (less-than-or-equal-to operator)
  - = (assignment operator) 2nd 3rd
  - == (equality operator) 2nd
  - > (greater-than operator)
  - >= (greater-than-or-equal-to operator)
  - >> (stream extraction operator)
  - addition assignment (+=)
  - address (&)
  - arithmetic
  - arrow memberselection (->)
  - assignment
  - binary scope resolution (::)
  - conditional (?:)
  - const\_cast
  - decrement (--) 2nd
  - delete 2nd 3rd 4th 5th 6th
  - dot (.)
  - dynamic\_cast 2nd
  - increment (++) 2nd
  - member selection (.)
  - multiplicative (\*, /, %)
  - new 2nd 3rd
  - postfix decrement
  - postfix increment 2nd
  - prefix decrement
  - prefix increment 2nd 3rd
  - sizeof 2nd 3rd 4th 5th 6th 7th 8th 9th 10th
  - ternary
  - unary minus (-)
  - unary plus (+)
  - unary scope resolution (::)
  - || (logical OR operator) 2nd

operator +=  
operator associativity  
operator functions  
operator keyword  
operator keywords 2nd  
demonstration  
operator overloading 2nd 3rd 4th 5th  
decrement operators  
increment operators  
operator precedence 2nd 3rd 4th  
operator precedence and associativity chart  
operator void\* member function 2nd 3rd  
operator! member function 2nd 3rd  
operator!= 2nd  
operator()  
operator() overloaded operator  
operator+  
operator++ 2nd  
operator++(int)  
operator< 2nd  
operator<< 2nd 3rd 4th  
operator<=  
operator= 2nd  
operator== 2nd 3rd 4th  
operator>  
operator>= 2nd  
operator>> 2nd 3rd  
operator[ ]  
const version  
operators .\* and ->\*  
operators that can be overloaded  
optical disk  
optimization  
optimizations on constants  
optimized code  
optimizing compiler 2nd 3rd  
optimizing the simple compiler  
option element (XHTML)  
options HTTP request type  
or operator keyword  
or\_eq operator keyword  
order 1 (Big O)

o

order in which actions should execute 2nd  
order in which constructors and destructors are called  
order in which destructors are called  
order in which operators are applied to their operands  
order log n (Big O)  
order n (Big O)  
order n-squared (Big O)  
order of evaluation  
    of operators 2nd  
ordered list 2nd  
org (top-level domain)  
original format settings  
ostream 2nd 3rd 4th 5th  
ostream class  
    seekp function  
    tellp function  
ostream\_iterator  
ostringstream 2nd  
other character sets  
out of scope  
out-of-range array subscript  
out-of-range element 2nd  
out\_of\_bounds exception  
out\_of\_range exception 2nd 3rd 4th  
outer block  
outer for structure  
output a floating-point value  
output buffering  
output data  
output data items of built-in type  
output device  
output format of floating-point numbers  
output iterator 2nd 3rd 4th 5th 6th 7th 8th  
output of char \* variables  
output of characters  
output of floating-point values  
output of integers  
output of standard data types  
output of uppercase letters  
output sequence  
output stream  
output to string in memory

o

output unit  
outputting to strings in memory  
oval  
overflow 2nd  
overflow error  
overflow\_error exception  
overhead of a function call  
overhead of an extra function call  
overhead of virtual function  
overloaded \* operator  
overloaded -> operator  
overloading  
+  
+=  
+= concatenation operator  
+= operator  
<< operator 2nd  
<<and>>  
[ ] operator  
a member function  
addition assignment operator (+=)  
addition operator (+)  
an assignment operator  
an operator as a non-member, non-friend function  
assignment (=) operator 2nd 3rd 4th  
binary operator <  
binary operators 2nd  
cast operator function  
concatenation operator  
equality operator (==) 2nd 3rd  
function 2nd 3rd 4th  
function call operator ( ) 2nd  
function definitions  
increment operator  
inequality operator 2nd  
less than operator  
negation operator  
operator[ ] member function  
operators  
output operator  
parentheses ( )  
postfix increment operator 2nd 3rd

prefix and postfix decrement operators  
prefix and postfix increment operators  
prefix increment operator 2nd  
stream insertion and stream extraction operators 2nd 3rd 4th 5th 6th 7th  
stream insertion operator  
string concatenation operator  
subscript operator 2nd 3rd 4th  
unary operator !  
unary operators  
overloading resolution  
overloading template functions  
overloading the stream insertion operator  
override a function

 PREV

NEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

p (paragraph) element (XHTML)

package

package diagram in the UML

Package inheritance hierarchy

Package inheritance hierarchy exercise

packet

pad with specified characters

padding

padding characters 2nd 3rd 4th

padding in a structure

page layout software

Paint Shop Pro

pair 2nd

pair of braces {} 2nd

pair of iterators

palindrome

palindrome function

paragraph tag

parallel activities

parallelogram

parameter 2nd 3rd

parameter in the UML 2nd 3rd 4th

parameter list 2nd

parameterized stream manipulator 2nd 3rd 4th 5th 6th

parameterized type 2nd

parent node 2nd

parentheses operator (()) 2nd

parentheses to force order of evaluation

partial\_sort STL algorithm

partial\_sort\_copy STL algorithm  
partial\_sum STL algorithm  
partition step in quicksort 2nd  
partition STL algorithm  
partitioning element  
partitioning step  
Pascal, Blaise  
pass a structure variable to a function  
pass size of an array  
pass size of an array as an argument  
pass-by-reference 2nd 3rd 4th 5th 6th 7th  
    with a pointer parameter used to cube a variable's value  
    with pointer parameters 2nd  
    with reference parameters 2nd  
pass-by-value 2nd 3rd 4th 5th 6th 7th 8th  
    used to cube a variable's value  
passing a filename to a program  
passing an array element  
passing an entire array  
passing an object by value  
passing arguments by value and by reference  
passing arrays and individual array elements to functions  
passing arrays to functions  
passing large objects  
passing options to a program 2nd  
password attribute value (type)  
password box  
password-protected database  
"past the end" iterator  
path attribute in cookies  
path to a resource  
pattern of 1s and 0s  
payroll file  
payroll system  
Payroll System Modification exercise  
peek function of istream  
percent sign (%) (modulus operator)  
perfect number  
    exercise  
perfect numbers  
perform a task  
perform an action

performance  
performance of binary tree sorting and searching  
Performance Tip  
permutation  
personal computer 2nd  
Peter Minuit problem 2nd  
phases of a program  
PhoneNumber class  
PhotoShop Elements  
pi  
PI 2nd  
picture.html  
pieceworker  
pig Latin  
pipe  
pipe ()  
piping  
pixel  
platform  
Plauger, P.J.  
playing cards  
plus function object  
plus sign (+) displayed in output  
plus sign (+) indicating public visibility in the UML  
plus sign, + (UML)  
Point class represents an x-y coordinate pair  
point-of-sale system  
pointer 2nd  
pointer (STL)  
pointer arithmetic 2nd 3rd 4th 5th  
    machine dependent  
    on a character array  
pointer assignment  
pointer comparison  
pointer dereference (\*) operator 2nd  
pointer exercises  
pointer expression 2nd  
pointer handle  
pointer link  
pointer manipulation 2nd  
pointer notation  
pointer operators & and \*

pointer subtraction  
pointer to a function 2nd 3rd  
pointer to an object 2nd  
pointer to void (`void *`)  
pointer values as hexadecimal integers  
pointer variable  
pointer-based strings  
pointer-to-member operator  
    `->*`  
    `.*`  
pointer/offset notation  
pointer/subscript notation  
pointers and array subscripting 2nd  
pointers and arrays  
pointers declared `const`  
pointers to dynamically allocated storage 2nd  
poker playing program  
poll analysis program  
Polymorphic banking program exercise using `Account` hierarchy  
polymorphic exception processing  
polymorphic programming 2nd 3rd  
polymorphic screen manager  
Polymorphic screen manager exercise using `Shape` hierarchy  
polymorphically invoking functions in a derived class  
polymorphism 2nd 3rd 4th 5th 6th 7th 8th  
    and references  
    as an alternative to `switch` logic  
polynomial  
Polynomial class  
pop function of container adapters  
pop member function of `priority_queue`  
pop member function of `queue`  
pop member function of `stack`  
pop off a stack  
pop\_back member function of `list`  
pop\_front 2nd 3rd  
pop\_heap STL algorithm  
portability  
Portability Tip 2nd  
portable  
portable code  
portable language

portal.cgi  
portal.cpp  
position number  
positional notation  
positional value 2nd  
positional values in the decimal number system  
post HTTP request type 2nd  
post request type  
post.cpp  
postdecrement 2nd  
postfix decrement operator  
postfix evaluation algorithm 2nd  
postfix evaluator 2nd  
postfix expression 2nd  
postfix expression evaluation algorithm  
postfix increment operator 2nd  
postfix notation  
postincrement 2nd  
postincrement an iterator  
postorder traversal 2nd 3rd  
postOrderTraversal  
pow function 2nd 3rd 4th  
power  
precedence 2nd 3rd 4th 5th 6th 7th  
chart  
not changed by overloading  
of the conditional operator  
precision 2nd 3rd  
format of a floating-point number  
precision function of `ios_base`  
precision of a floating-point value  
precision of floating-point numbers  
Precision of floating-point values  
precision setting  
precompiled object file  
predecrement 2nd  
predefined function objects  
predefined namespace  
predefined symbolic constants  
predicate function 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th  
prefix decrement operator  
prefix increment operator 2nd 3rd

preincrement 2nd  
preorder traversal  
prepackaged data structures  
"prepackaged" functions  
preprocess  
preprocessor 2nd 3rd  
preprocessor directives 2nd 3rd

#define 2nd 3rd 4th  
#endif 2nd  
#if  
#ifdef  
#ifndef 2nd  
#include 2nd  
#pragma

preprocessor wrapper  
presentation of a document  
prevent class objects from being copied  
prevent memory leak  
prevent one class object from being assigned to another  
preventing header files from being included more than once  
primary memory 2nd  
prime  
prime factorization  
prime numbers

    exercise  
primitive type  
    promotion  
principal 2nd  
principle of least privilege 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th  
print a line of text  
print a linked list backwards  
print a list backwards  
print debugger command  
print spooling  
printArray function template  
printer 2nd  
printing a binary tree in a two-dimensional tree format  
printing a line of text with multiple statements  
Printing a string one character at a time using a nonconstant pointer to constant data  
printing a tree  
printing an integer with internal spacing and plus sign  
printing an unsigned integer in bits

printing character other than a space, digit or letter

printing character other than space

printing character, including space

printing dates

printing multiple lines of text with a single statement

printing string characteristics

printing the address stored in a `char * variable`

printing the value of a union in both member data types

`priority_queue` adapter class

empty function

pop function

push function

size function

top function

private

access specifier 2nd

base class

base class data cannot be accessed from derived class

data member

inheritance 2nd 3rd 4th

inheritance as an alternative to composition

members of a base class

private libraries

private static data member

probability

procedural programming language

procedure 2nd

processing phase

processing unit

product of odd integers

program

program control

program development environment

program development tool 2nd

program execution stack

program in the general 2nd

program in the specific

program termination 2nd 3rd

programmer

programmer-defined function

programming environment

projects

promotion

promotion hierarchy for built-in data types

promotion rules

prompt 2nd 3rd

prompting message

proprietary classes

protected

protected access specifier

protected base class

protected base-class data can be accessed from derived class

protected inheritance 2nd 3rd

protection mechanism

protocol

proxy class 2nd 3rd

pseudocode 2nd 3rd 4th 5th 6th 7th

first refinement

second refinement

top

top-down, stepwise refinement

two levels of refinement

pseudorandom numbers

public

access specifier 2nd

base class

inheritance 2nd

interface

keyword 2nd 3rd 4th

member of a derived class

services of a class

public static class member

public static member function

punctuation mark

pure procedure

pure specifier

pure virtual function 2nd

purpose of the program

push 2nd 3rd

push function of container adapters

push member function of priority\_queue

push member function of queue

push member function of stack

push memory location on the stack

push onto a stack  
push\_back member function of vector  
push\_front member function of deque  
push\_front member function of list  
push\_heap  
put file-position pointer 2nd  
put HTTP request type  
put member function 2nd  
put pointer  
putback function of istream  
Pythagorean triples

 PREV

NEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y] [Z]

Quadrabay Corporation's Web site

quadratic runtime

qualified name

qualityPoints

query string 2nd

QUERY\_STRING environment variable 2nd

querystring.cpp

queue 2nd 3rd 4th 5th 6th 7th

queue adapter class 2nd

  back function

  empty function

  front function

  pop function

  push function

  size function

queue class

Queue class-template definition

queue grows infinitely

queue in a computer network

queue object

Queue-processing program

Quick Info box

quicksort algorithm 2nd

quit debugger command

quotation marks

[◀ PREV](#)

[NEXT ▶](#)

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

radians  
radio attribute value (XHTML) 2nd  
radio button (XHTML)  
radius of a circle  
raise function  
raise to a power 2nd  
rand function 2nd  
RAND\_MAX symbolic constant  
random integers in range 1 to 6  
random intervals  
random number  
random-access file 2nd 3rd 4th 5th 6th 7th  
random-access iterator 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th  
random-access iterator operators  
random\_shuffle STL algorithm 2nd 3rd  
randomizing  
randomizing the die-rolling program  
range 2nd  
range checking 2nd 3rd  
rapid application development (RAD)  
Rational Class exercise  
Rational Software Corporation 2nd  
Rational Unified Process™  
RationalNumber Class exercise  
raw array  
raw data  
raw data processing  
rbegin member function of class string  
rbegin member function of containers

rbegin member function of vector  
RDBMS (relational database management system)  
rdstate function of ios\_base  
read a line of text  
read characters with getline  
read data sequentially from a file  
read function of istream 2nd 3rd 4th 5th  
read-only variable  
readability 2nd 3rd  
readcookie.cpp  
Reading a random-access file sequentially  
Reading and printing a sequential file  
real number  
realloc  
reassign a reference  
"receiving" section of the computer  
record 2nd 3rd 4th  
record format  
record key 2nd  
recover from errors  
Rectangle  
Rectangle Class exercise  
recursion 2nd 3rd 4th  
recursion examples and exercises  
recursion exercises  
    binary search  
    linear search  
recursion step 2nd 3rd  
recursive binary search 2nd 3rd  
recursive binary tree delete  
recursive binary tree insert  
recursive binary tree printing  
recursive call 2nd  
recursive Eight Queens 2nd  
recursive factorial function  
recursive Fibonacci function  
recursive function 2nd  
recursive function call  
recursive greatest common divisor  
recursive inorder traversal of a binary tree  
recursive linear search 2nd 3rd 4th 5th  
recursive maze traversal

recursive mergesort  
recursive postorder traversal of a binary tree  
recursive preorder traversal of a binary tree  
recursive quicksort 2nd  
recursive selection sort 2nd  
recursive solution  
recursive step  
recursive sum of two integers  
recursive Towers of Hanoi  
recursive utility function  
recursively calculate minimum value in an array  
recursively check if a string is a palindrome  
recursively determine whether a string is a palindrome  
recursively print a linked list backward  
recursively print a list backwards  
recursively print a string backward 2nd  
recursively print an array 2nd  
recursively raising an integer to an integer power  
recursively search a linked list  
recursively search a list  
redirect input symbol  
redirect input/output on UNIX, LINUX, Mac OS X and Windows systems  
redirect inputs to come from a file  
redirect output of one program to input of another program  
redirect output symbol >  
redirect outputs to a file  
redirecting input on a DOS system  
redundant parentheses 2nd  
reentrant code  
reference 2nd 3rd  
reference argument  
reference parameter 2nd  
reference to a constant 2nd  
reference to a `private` data member  
reference to an automatic variable  
reference to an `int`  
reference to an object  
references must be initialized  
referencing array elements  
referencing array elements with the array name and with pointers  
refinement process  
Refresh header

register a function for `atexit`  
register declaration  
register storage-class specifier  
`reinterpret_cast` operator 2nd 3rd  
reinventing the wheel  
relational database management system (RDBMS)  
relational operator 2nd  
relational operators `>`, `<`, `>=`, and `<=` 2nd  
release dynamically allocated memory  
reliable software  
`rem` statement in Simple 2nd 3rd  
remainder after integer division  
remote Web server  
`remove` member function of `list`  
`remove` STL algorithm 2nd  
`remove_copy` STL algorithm 2nd  
`remove_copy_if` STL algorithm 2nd 3rd  
`remove_if` STL algorithm 2nd 3rd  
`rend` member function of class `string`  
`rend` member function of containers  
`rend` member function of `vector`  
repeatability of function `rand`  
repetition  
    counter controlled 2nd 3rd  
    definite  
    indefinite  
    sentinel controlled 2nd  
repetition statement 2nd 3rd 4th 5th  
    `do...while` 2nd 3rd 4th  
    `for` 2nd 3rd 4th  
    `while` 2nd 3rd 4th 5th  
repetition terminates 2nd  
replace `==` operator with `=`  
`replace` member function of class `string` 2nd  
`replace` STL algorithm 2nd 3rd  
`replace_copy` STL algorithm 2nd 3rd  
`replace_copy_if` STL algorithm 2nd 3rd  
`replace_if` STL algorithm 2nd 3rd  
replacement node  
replacement text 2nd  
    for a macro or symbolic constant 2nd

request method  
requesting a service from an object  
requirements 2nd 3rd  
requirements document 2nd 3rd 4th  
requirements gathering  
reset attribute value (type)  
reset of bitset  
resize member function of class string  
resource leak 2nd  
restore a stream's state to "good"  
resumption model of exception handling  
rethrow an exception  
return a result 2nd  
Return key  
return message in the UML  
return statement 2nd 3rd 4th 5th  
return type  
    in a function header  
    in the UML 2nd  
    void 2nd  
returning a reference from a function  
returning a reference to a private data member  
Returning Error Indicators from Class Time's set Functions exercise  
reusability 2nd 3rd  
reusable componentry 2nd  
reusable software component  
reuse 2nd 3rd  
reusing components  
Reverse Digits exercise  
reverse order of bits in unsigned integer  
reverse STL algorithm 2nd 3rd  
reverse\_copy STL algorithm 2nd 3rd  
reverse\_iterator 2nd 3rd 4th 5th  
rfind member function of class string  
Richards, Martin  
right brace () 2nd 3rd  
right child  
right justification 2nd 3rd 4th  
right operand  
right shift operator (>>) 2nd  
right stream manipulator 2nd 3rd  
right subtree 2nd 3rd 4th 5th

right triangle 2nd  
right value (rvalue)  
right-shift operator ( $>>$ ) 2nd 3rd 4th 5th  
right-shifting a signed value is machine dependent  
right-to-left associativity 2nd  
rightmost (trailing) arguments  
rightmost node of a subtree  
rise-and-shine algorithm  
Ritchie, D.  
robust application 2nd  
Rogue Wave  
role in the UML  
role name in the UML  
rolling a die  
rolling a six-sided die 6000 times  
rolling two dice 2nd  
root node 2nd  
root node of the left subtree  
root node of the right subtree  
rotate STL algorithm 2nd  
rotate\_copy STL algorithm 2nd  
round a floating-point number for display purposes  
rounded rectangle (for representing a state in a UML state diagram)  
rounding 2nd  
row subscript  
rows  
rows attribute of element `textarea`  
rowspan attribute of element `th`  
RTTI (run-time type information) 2nd  
rule of thumb  
rules for forming structured programs  
rules of operator precedence  
Rumbaugh, James  
run debugger command  
running total  
runtime error  
runtime type information (RTTI) 2nd 3rd  
`runtime_error` class  
    what function  
`runtime_error` exception 2nd 3rd 4th  
rvalue ("right value") 2nd 3rd 4th



Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

SalariedEmployee class header file

SalariedEmployee class implementation file

SalesPerson class definition

SalesPerson class member-function definitions

savings account

SavingsAccount class

scalable

scalar 2nd

scalar quantity

scale

scaling

scaling factor 2nd

scanning images

scientific notation 2nd 3rd

scientific notation floating-point value 2nd

scientific stream manipulator 2nd

scope 2nd 3rd

block

class

file

function

function prototype

namespace

scoping example

scope of a symbolic constant or macro

scope of an identifier 2nd

scope resolution operator (:) 2nd 3rd 4th 5th 6th 7th

screen 2nd 3rd 4th

Screen class (ATM case study) 2nd 3rd 4th 5th 6th 7th 8th 9th

screen cursor

screen output

screen-manager program

script

scrutinize data

search a linked list 2nd

search algorithms

binary search

linear search

recursive binary search

recursive linear search

search engine

search functions of the string-handling library

search key 2nd 3rd

search STL algorithm 2nd

search\_n STL algorithm 2nd

searching 2nd 3rd

searching arrays

searching blocks of memory

searching data

searching performance

searching strings 2nd

second data member of pair

second pass of Simple compiler

second refinement 2nd 3rd 4th

second-degree polynomial 2nd

secondary storage device 2nd

secondary storage unit

"secret" implementation details

security 2nd 3rd

security hole

security issues involving the Common Gateway Interface

seed

seed function rand

seek direction

seek get

seek put

seekg function of istream 2nd

seekp function of ostream 2nd

segmentation violation

select a substring

select element (XHTML)

select XHTML element (form)  
selected attribute of element option  
selection 2nd  
selection sort algorithm 2nd 3rd  
selection sort with call-by-reference  
selection statement 2nd  
self assignment 2nd 3rd  
self-documenting  
self-referential class 2nd  
self-referential structure  
semicolon (;) 2nd 3rd 4th 5th  
semicolon that terminates a structure definition  
sentinel value 2nd 3rd 4th  
sentinel-controlled loop  
sentinel-controlled repetition 2nd 3rd  
separate interface from implementation  
sequence 2nd 3rd 4th 5th 6th 7th  
sequence container 2nd 3rd 4th 5th

- back function 2nd
- empty function
- front function 2nd
- insert function

  
sequence diagram in the UML 2nd  
sequence of integers  
sequence of messages in the UML  
sequence of random numbers  
sequence statement 2nd 3rd  
sequence type  
sequence-statement activity diagram  
sequential container  
sequential execution  
sequential file 2nd 3rd 4th 5th 6th 7th  
server  
server-side file 2nd 3rd  
server-side form handler  
services of a class  
set a value  
set and get functions  
set associative container  
set debugger command  
set function 2nd  
set of recursive calls to method Fibonacci

set operations of the Standard Library  
set the value of a private data member  
Set-Cookie: HTTP header  
set\_difference STL algorithm 2nd  
set\_intersection  
set\_intersection STL algorithm 2nd  
set\_new\_handler function 2nd  
set\_new\_handler specifying the function to call when new fails  
set\_symmetric\_difference STL algorithm 2nd  
set\_terminate function  
set\_unexpected function 2nd  
set\_union STL algorithm 2nd  
setbase stream manipulator  
setfill stream manipulator 2nd 3rd 4th 5th  
setprecision stream manipulator 2nd 3rd  
setw stream manipulator 2nd 3rd 4th 5th 6th  
Shakespeare, William  
Shape class hierarchy 2nd  
    exercise  
shape of a tree  
shell prompt on Linux  
shift a range of numbers  
shifted, scaled integers  
shifted, scaled integers produced by `1 + rand() % 6`  
shiftingValue  
"shipping" section of the computer  
shopping cart application 2nd  
short data type 2nd  
short int data type  
short-circuit evaluation  
showbase stream manipulator 2nd  
showpoint stream manipulator 2nd  
showpos stream manipulator 2nd 3rd  
shrink-wrapped software  
shuffle cards 2nd  
shuffling algorithm  
sibling  
side effect 2nd 3rd 4th  
sides of a right triangle  
sides of a square  
sides of a triangle

Sieve of Eratosthenes 2nd 3rd  
SIGABRT  
SIGFPE  
SIGILL  
SIGINT  
sign extension  
sign left justified  
signal  
signal function  
signal handler  
signal number  
signal value  
signal-handling library 2nd  
signals defined in header <csignal>  
signature 2nd 3rd  
signatures of overloaded prefix and postfix increment operators  
significant digits  
SIGSEGV  
SIGTERM  
Silicon Graphics Standard Template Library Programmer's Guide 2nd  
simple CGI script  
simple condition 2nd  
Simple interpreter  
Simple language  
Simple statement  
simplest activity diagram 2nd  
Simpletron Machine Language (SML) 2nd 3rd 4th 5th  
Simpletron memory location  
Simpletron Simulator 2nd 3rd 4th 5th  
Simula  
simulated deck of cards  
simulation  
Simulation: Tortoise and the Hare  
sin function  
sine  
single entry point  
single exit point  
single inheritance 2nd 3rd  
single quote (') 2nd  
single selection  
single-argument constructor 2nd 3rd 4th  
single-entry/single-exit control statement 2nd 3rd

single-line comment  
single-precision floating-point number  
single-selection `if` statement 2nd 3rd 4th  
singly linked list  
six-sided die  
`size` attribute of element `input`  
`size` function of `string`  
`size` member function of class `string`  
`size` member function of containers  
`size` member function of `priority_queue`  
`size` member function of `queue`  
`size` member function of `stack`  
`size` member function of `vector` 2nd  
`size` of a `string`  
`size` of a structure  
`size` of a variable 2nd  
`size` of an array 2nd  
`size_t` type 2nd  
`size_type`  
`sizeof` operator 2nd 3rd 4th 5th 6th 7th 8th 9th 10th  
    used to determine standard data type sizes  
`sizeof` operator when applied to an array name returns the number of bytes in the array  
`sizes` of the built-in data types  
`skip` remainder of `switch` statement  
`skip` remaining code in loop  
`skipping` white-space characters  
`skipping whitespace`  
`skipws` stream manipulator  
`small circle symbol`  
`smaller integer sizes`  
`smallest`  
`smallest of several integers`  
`"smart array"`  
`SML` 2nd  
`SML branch zero instruction`  
`SML operation code`  
`"sneakernet"`  
`software` 2nd  
`software asset`  
`software engineering` 2nd 3rd 4th  
    data hiding 2nd  
    encapsulation

reuse 2nd

separate interface from implementation

set and get functions

Software Engineering Observations

software life cycle

software reuse 2nd 3rd 4th 5th 6th 7th 8th 9th

solid circle (for representing an initial state in a UML diagram) in the UML 2nd

solid circle enclosed in an open circle (for representing the end of a UML activity diagram)

solid circle symbol

solid diamonds (representing composition) in the UML

Solution Configurations ComboBox

sort algorithms

bubble sort

bucket sort

insertion sort 2nd

merge sort

quicksort

selection sort 2nd

sort key

sort member function of list

sort standard library function

sort STL algorithm 2nd

sort\_heap STL algorithm

sorting 2nd 3rd 4th

sorting arrays

sorting data 2nd

sorting order 2nd

sorting strings

source code 2nd 3rd

source file

source-code file

space (' ')

space cannot be allocated

space-time trade-off

spaces for padding

span attribute

span attribute of element col

speaking to a computer

special character 2nd 3rd

XHTML

Special Section: Building Your Own Computer 2nd

special symbol

specialization in the UML  
specifics  
speech device  
speech synthesizer  
spelling checker  
spiral  
`splice` member function of `list`  
split the array in merge sort  
spooler  
`sqrt` function of `<cmath>` header file  
square  
square function  
square root 2nd  
squares of several integers  
`srand` function 2nd  
`srand(time(0))`  
`src` attribute of element `img` 2nd  
`stable_partition` STL algorithm  
`stable_sort` STL algorithm  
stack 2nd 3rd  
Stack  
stack 2nd 3rd 4th 5th 6th  
    unwinding 2nd 3rd 4th  
stack adapter class  
    `empty` function  
    `pop` function  
    `push` function  
    `size` function  
    `top` function  
stack class  
Stack class template 2nd 3rd 4th 5th  
    definition  
    definition with a composed `List` object  
stack frame  
stack overflow  
Stack test program  
stack unwinding 2nd 3rd 4th  
stack-of-float class  
stack-of-int class  
stack-of-string class  
`Stack<double>` 2nd 3rd  
stack<int>

S

Stack<T> 2nd  
stacked building blocks  
stacking 2nd 3rd  
stacking rule  
stacks implemented with arrays  
stacks used by compilers

"stand-alone" units  
standard algorithm  
standard class libraries  
standard data type sizes  
standard error stream (`cerr`)  
standard exception classes  
standard input 2nd  
standard input stream object (`cin`) 2nd 3rd

## Standard Library

class `string` 2nd  
container classes  
container header files  
deque class template  
exception classes  
exception hierarchy  
header files 2nd  
`list` class template  
`map` class template  
`multimap` class template  
`multiset` class template  
`priority_queue` adapter class  
queue adapter class templates  
`set` class template  
`stack` adapter class  
`vector` class template

standard output stream object (`cout`) 2nd 3rd 4th

standard signals

standard stream libraries

standard template library (STL)

Standard Template Library (STL) 2nd 3rd 4th

Programmer's Guide 2nd

"standardized, interchangeable parts"

start tag (XHTML)

starvation

state

state bits

state diagram for the ATM object  
state diagram in the UML  
state in the UML 2nd  
state machine diagram in the UML 2nd  
state of an object 2nd  
statement 2nd  
statement spread over several lines  
statement terminator (:)  
statements  
    break 2nd 3rd  
    continue 2nd 3rd  
    do...while 2nd 3rd 4th  
    for 2nd 3rd 4th  
    if 2nd 3rd  
    if...else  
    return 2nd  
    switch 2nd 3rd  
    while 2nd 3rd  
static 2nd 3rd  
static array initialization  
static array initialization and automatic array initialization  
static binding  
static data member 2nd 3rd 4th  
static data member tracking the number of objects of a class  
static data members save storage  
static keyword  
static linkage specifier  
static local object 2nd  
static local variable 2nd 3rd 4th  
static member  
static member function  
static storage class 2nd 3rd  
static storage-class specifier  
static Web content  
static\_cast (compile-time type-checked cast) 2nd 3rd  
static\_cast<int>  
status bits  
Status header  
std namespace 2nd  
std::cin 2nd  
std::cout

S

std::endl stream manipulator  
step debugger command  
Step Into command (debugger)  
Step Out command (debugger)  
Step Over command (debugger)  
Stepanov, Alexander 2nd 3rd  
stepwise refinement  
"sticky" setting 2nd  
**STL**  
**STL (Standard Template Library)**  
**STL algorithms**  
    accumulate 2nd 3rd 4th 5th  
    adjacent\_difference  
    adjacent\_find 2nd  
    binary\_search 2nd 3rd  
    copy 2nd  
    copy\_backward 2nd  
    count 2nd 3rd  
    count\_if 2nd 3rd  
    equal 2nd  
    equal\_range 2nd  
    fill 2nd  
    fill\_n 2nd  
    find 2nd 3rd  
    find\_each  
    find\_end  
    find\_first\_of  
    find\_if 2nd 3rd  
    for\_each 2nd  
    generate 2nd  
    generate\_n 2nd  
    implace\_merge  
    includes  
    inner\_product  
    inplace\_merge  
    iter\_swap 2nd 3rd  
    lexicographical\_compare 2nd  
    lower\_bound  
    make\_heap  
    max  
    max\_element 2nd

s

merge 2nd  
min  
min\_element 2nd  
mismatch 2nd 3rd  
partial\_sort  
partial\_sort\_copy  
partial\_sum  
partition  
pop\_heap  
push\_heap 2nd  
random\_shuffle 2nd 3rd  
remove 2nd  
remove\_copy 2nd  
remove\_copy\_if 2nd 3rd  
remove\_if 2nd 3rd  
replace 2nd 3rd  
replace\_copy 2nd 3rd  
replace\_copy\_if 2nd 3rd  
replace\_if 2nd 3rd  
reverse 2nd 3rd  
reverse\_copy 2nd 3rd  
rotate 2nd  
rotate\_copy 2nd  
search 2nd  
search\_n 2nd  
set\_difference 2nd  
set\_intersection 2nd  
set\_symmetric\_difference 2nd  
set\_union 2nd  
sort 2nd  
sort\_heap  
stable\_partition  
stable\_sort  
swap 2nd  
swap\_ranges 2nd 3rd  
transform 2nd 3rd  
unique 2nd 3rd  
unique\_copy 2nd 3rd  
upper\_bound 2nd  
STL container functions  
STL exception types

STL in generic programming 2nd  
STL References 2nd  
STL software  
STL tutorials 2nd  
storage alignment  
storage class 2nd 3rd  
storage class specifiers  
    auto  
    extern  
    mutable  
    register  
    static  
storage unit  
storage unit boundary  
str member function  
str member function of class ostream-stream  
straight-line form 2nd  
straight-time  
strcat function of header file <cstring> 2nd 3rd  
strchr function of header file <cstring>  
strcmp function of header file <cstring> 2nd 3rd  
strcpy function of header file <cstring> 2nd  
strcspn function of header file <cstring> 2nd 3rd  
stream base  
stream classes  
stream extraction operator >> ("get from") 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th  
stream I/O class hierarchy 2nd  
stream input 2nd  
stream input/output  
stream insertion operator << ("put to") 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th  
stream manipulator 2nd 3rd 4th 5th 6th 7th  
    boolalpha 2nd  
    boolalpha and noboolalpha  
    dec  
    endl 2nd  
    endl (end line)  
    fixed 2nd  
    hex  
    internal  
    left 2nd 3rd  
    noboolalpha

s

  noshowbase 2nd  
  noshowpoint  
  noshowpos 2nd 3rd  
  noskipws  
  nouppercase 2nd  
  oct 2nd 3rd  
  right 2nd  
  scientific  
  setbase 2nd  
  setfill 2nd 3rd 4th 5th  
  setprecision 2nd 3rd  
  setw 2nd 3rd 4th 5th 6th  
  showbase 2nd  
  showpoint 2nd  
  showpos  
  
stream of bytes  
stream of characters  
stream operation failed  
stream output  
string  
  string  
    size function  
string array  
string assignment 2nd  
string being tokenized  
string class 2nd 3rd 4th  
String class  
  string class 2nd 3rd  
    copy constructor  
    data function  
String class  
  definition with operator overloading  
string class  
  from the Standard Library  
  insertmember function  
  length member function  
String class  
  member-function and friend-function definitions  
string class  
  substr member function 2nd  
String class  
  test program

S

string comparison 2nd

string concatenation

string constant

string conversion function

atof

atoi

atol

strtod

strtol

strtoul

string copying

using array notation and pointer notation

string data type

string find member function

string input and output

string is a constant pointer

string length

string literal 2nd 3rd 4th 5th 6th

string manipulation 2nd

string object

empty string

initial value

string of characters

string processing

string search function

strchr

strcspn

struprbrk

strrchr

strspn

string search function strstr

string stream processing

string::npos (end of string)

strings as full-fledged objects

strlen function 2nd

strncat function 2nd

strncmp function 2nd

strncpy function 2nd

strong element (XHTML)

Stroustrup, Bjarne 2nd 3rd 4th 5th 6th

struprbrk function 2nd

strrchr function 2nd

strspn function 2nd  
strstr function 2nd  
strtod function 2nd 3rd  
strtok function 2nd  
strtol function 2nd 3rd 4th  
strtoul function 2nd  
struct 2nd 3rd  
structure 2nd 3rd 4th 5th  
    definition 2nd 3rd  
    member  
    member operator (.)  
    members default to private access  
    name 2nd  
structure of a system 2nd  
structured programming 2nd 3rd 4th 5th 6th 7th 8th 9th  
    summary  
structured systems analysis and design  
student-poll-analysis program  
style sheet  
sub element (XHTML)  
subclass  
submit attribute value (type)  
subobject  
    base class  
subproblem  
subscript  
subscript (XHTML)  
subscript 0 (zero)  
subscript operator  
subscript operator [ ]  
subscript operator [ ] used with strings  
subscript operator of map  
subscript out of range  
subscript range checking  
subscript through a vector  
subscripted name of an array element  
subscripted name used as an rvalue  
subscripting  
subscripting with a pointer and an offset  
substr member function of string 2nd 3rd 4th  
substring  
substring length

substring of a string  
subtract an integer from a pointer  
subtract one pointer from another  
subtraction 2nd 3rd  
suit values of cards  
sum of the elements of an array  
summary attribute of element table  
summing integers with the for statement  
sup element (XHTML)  
superclass  
supercomputer  
supermarket checkout line  
supermarket simulation  
superscript (XHTML)  
survey 2nd  
swap member function of class string  
swap member function of containers  
swap member function of list  
swap STL algorithm 2nd  
swap\_ranges STL algorithm 2nd 3rd  
swapping strings  
swapping values 2nd  
swapping values (in sorting algorithms) 2nd  
switch logic 2nd 3rd  
switch multiple-selection statement 2nd 3rd  
switch multiple-selection statement activity diagram with break statements  
switch structure  
symbol  
symbol table 2nd 3rd  
symbolic constant 2nd 3rd 4th 5th  
symbolic constant NDEBUG  
symbolic constant PI  
symmetric key encryption  
synchronize operation of an istream and an ostream  
synchronous call  
synchronous error  
synonym 2nd  
syntax  
syntax checking  
syntax error  
system  
system behavior

System Properties window

system requirements

system structure

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]

tab escape sequence \t 2nd

Tab key

tab stop

table body (XHTML table)

table body (XHTML)

table data (XHTML table)

table element (XHTML)

table foot (XHTML table)

table foot (XHTML)

table head (XHTML)

table of values

table row (XHTML table)

tabular format

tail of a list

tail of a queue 2nd

tail pointer

tails

tan function

tangent

task

tbody element (XHTML)

td element (XHTML)

telephone

Telephone number word generator

tellg function of istream

tellp function of ostream

template 2nd 3rd 4th 5th

template definition

template function 2nd

template keyword 2nd  
template parameter 2nd  
template parameter list  
templates and friends  
templates and inheritance  
Temporary Internet Files directory  
temporary location 2nd  
temporary object  
temporary String object  
temporary value 2nd  
terminal  
terminate a loop  
terminate a program 2nd  
terminate a repetition structure  
terminate function 2nd  
terminate normally  
terminate successfully  
terminating execution  
terminating null character 2nd 3rd 4th 5th 6th 7th  
terminating null character, '0', of a string  
terminating right brace () of a block  
termination condition 2nd  
termination housekeeping  
termination model of exception handling  
termination order from the operating system  
termination phase  
termination request sent to the program  
termination test  
ternary conditional operator (?:) 2nd 3rd  
test  
test  
test characters  
test state bits after an I/O operation  
Testing error states  
text analysis  
text attribute value (type)  
text box  
text editor 2nd 3rd  
text file  
text substitution  
text-based browser 2nd  
text-printing program

T

text/html MIME type  
text/plain MIME type  
textarea (XHTML)  
textarea element  
textarea element (XHTML)  
textarea XHTML element  
tfoot element (XHTML)  
th (table header column) element  
thead (table head) tag  
thead element (XHTML)  
third refinement  
this pointer 2nd 3rd 4th 5th 6th  
this pointer used explicitly  
this pointer used implicitly and explicitly to access members of an object  
Thompson, Ken  
throughput  
throw a conditional expression  
throw an exception  
throw an int  
throw exceptions derived from standard exceptions  
throw exceptions not derived from standard exceptions  
throw keyword  
throw list  
throw point  
throw standard exceptions  
throw( ) exception specification  
TicTacToe Class exercise  
tie  
tier  
tilde character (~)  
Time class  
Time class containing a constructor with default arguments  
Time class definition  
Time class definition modified to enable cascaded member-function calls  
Time class member function definitions, including const member functions  
Time class member-function definitions  
Time class member-function definitions, including a constructor that takes arguments  
Time class with const member functions  
time function 2nd  
time source file is compiled  
time-and-a-half 2nd

time\_t type  
timesharing 2nd  
timing diagram in the UML  
title bar  
title element (XHTML)  
title of a document  
title XHTML element (<title>...</title>)  
TLD (top-level domain)  
token 2nd 3rd 4th  
tokenize a sentence into separate words  
tokenizing strings 2nd  
tolower 2nd  
top  
top member function of priority\_queue  
top member function of stack  
top of a stack 2nd  
top tier  
top-down, stepwise refinement 2nd 3rd  
top-level domain (TLD)  
  cn  
  com  
  edu  
  et  
  om  
  org  
  us  
Tortoise and the Hare  
total 2nd 3rd  
toupper function (<cctype>) 2nd 3rd 4th  
Towers of Hanoi 2nd  
tr element (XHTML)  
trace HTTP request type  
trailing zeros  
transaction  
  Transaction class (ATM case study) 2nd 3rd 4th 5th 6th  
transaction file  
transaction processing  
transaction record  
transaction-processing program  
transaction-processing system  
transfer of control  
transform STL algorithm 2nd 3rd

transition  
transition arrow 2nd 3rd 4th 5th  
transition between states in the UML  
translate  
translation  
translator program  
transmit securely  
trap unexpected event  
trapezoid  
`travel.html`  
traversal  
traversals forwards and backwards  
traverse a binary tree 2nd  
traverse the left subtree  
traverse the right subtree  
traversing a container  
tree 2nd 3rd 4th  
`Tree class template` 2nd  
`Tree<int>`  
`TreeNode class template`  
trigonometric cosine  
trigonometric sine  
trigonometric tangent  
`tripleByReference`  
`tripleCallByValue`  
true  
`true` 2nd 3rd 4th  
truncate 2nd 3rd 4th 5th 6th  
truncate fractional part of a double  
truth table  
  ! (logical NOT) operator  
  && (logical AND) operator  
  || (logical OR) operator  
`try block` 2nd 3rd 4th  
`try block expires`  
`try keyword`  
Turing Machine  
turtle graphics  
two largest values  
two levels of refinement  
two-dimensional array 2nd 3rd  
two-dimensional array manipulations

two's complement  
two's position  
tying an output stream to an input stream  
type attribute 2nd  
type attribute of element o1  
type checking 2nd 3rd  
type field  
type information  
type name (enumerations)  
type of a variable 2nd  
type of the this pointer  
type parameter 2nd 3rd 4th  
type qualifier  
type template parameter  
type-safe I/O  
type-safe linkage 2nd  
type\_info class 2nd  
typedef fstream  
typedef ifstream  
typedef iostream  
typedef istream  
typedef keyword 2nd 3rd 4th 5th 6th 7th 8th  
typedef ofstream  
typedef ostream  
typedefs in first-class containers  
typeid operator 2nd  
typename keyword 2nd

 PREVNEXT 

page footer

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

U integer suffix

u integer suffix

ul element

ul element (XHTML)

UL integer suffix

u1 integer suffix

ultimate operator overloading exercise

UML

action expression 2nd 3rd

action state 2nd

activity diagram 2nd 3rd 4th

arrow

attribute

class diagram

constructor in a class diagram

data types

decision

decision symbol

diamond symbol 2nd

dotted line

final state

guard condition 2nd 3rd

guillemets (« and »)

initial state

merge symbol

minus sign ()

note

plus sign (+)

public operation

small circle symbol

solid circle symbol

String type

transition

transition arrow 2nd 3rd 4th 5th

UML (Unified Modeling Language) 2nd 3rd 4th 5th 6th 7th 8th 9th 10th

diagram

UML activity diagram

UML Activity Diagram

solid circle (for representing an initial state) in the UML

solid circle enclosed in an open circle (for representing the end of an activity) in the UML

UML class diagram

attribute compartment

constructor

operation compartment

UML Partners

UML Sequence Diagram

activation

arrowhead

lifeline

UML State Diagram

rounded rectangle (for representing a state) in the UML

solid circle (for representing an initial state) in the UML

UML Use Case Diagram

actor

use case

unary cast operator

unary decrement operator (--)

unary increment operator (++)

unary minus (-) operator

unary operator 2nd 3rd

unary operator overload 2nd

unary plus (+) operator

unary predicate function 2nd 3rd

unary scope resolution operator (:) :

unbuffered output

unbuffered standard error stream

unconditional branch

unconditional branch goto

unconditional branch instruction

unconditional goto statement in Simple

undefined area in memory

undefined value

underflow\_error exception

underlying container  
underlying data structure  
underscore (\_)  
unexpected event  
unexpected function 2nd  
unformatted I/O 2nd 3rd  
unformatted I/O using the `read`, `gcount` and `write` member functions  
unformatted output 2nd  
Unicode® 2nd 3rd  
Unified Modeling Language (UML) 2nd 3rd 4th 5th 6th 7th 8th 9th  
Uniform Resource Locator (URL)  
unincremented copy of an object  
uninitialized local reference causes a syntax error  
uninitialized variable  
`union` 2nd  
    anonymous  
`union` constructor  
`union` functions cannot be `virtual`  
`union` with no constructor  
`unique keys` 2nd 3rd  
`unique member function of list`  
`unique STL algorithm` 2nd 3rd  
`unique_copy` STL algorithm 2nd 3rd  
universal-time format  
UNIX 2nd 3rd 4th 5th 6th 7th 8th 9th  
UNIX command line 2nd  
unmodifiable  
unmodifiable lvalue 2nd  
unnamed bit field  
unnamed bit field with a zero width  
unnamed namespace  
unnamed object  
unoptimized code  
`unordered list element (ul)`  
unresolved references 2nd  
`unsigned char` data type  
`unsigned` data type 2nd 3rd  
`unsigned int` data type 2nd 3rd  
unsigned integer in bits  
`unsigned long` data type 2nd 3rd 4th  
`unsigned long int` data type 2nd  
`unsigned short` data type

unsigned short int data type  
unspecified number of arguments  
unsuccessful termination  
untie an input stream from an output stream  
unwinding the function call stack  
update a record  
update records in place  
upper\_bound function of associative container  
upper\_bound STL algorithm  
uppercase letter 2nd 3rd 4th 5th 6th  
uppercase stream manipulator 2nd 3rd  
url(uniform resource locator) 2nd  
us (top-level domain)  
use case diagram in the UML 2nd  
use case in the UML 2nd  
use case modeling  
USENIX C++ Conference  
user interface  
user-defined class name  
user-defined function 2nd  
    maximum  
user-defined termination function  
user-defined type 2nd 3rd  
user-defined types  
User-defined, nonparameterized stream manipulators  
userdata.txt  
using a dynamically allocated `ostringstream` object  
using a function template  
Using a `static` data member to maintain a count of the number of objects of a class  
using an anonymous union  
using an iterator to output a `string`  
using arrays instead of `switch`  
using command-line arguments  
using declaration 2nd  
using directive  
using function `swap` to swap two `strings`  
using functions `exit` and `atexit`  
using `goto`  
using member function `fill` and stream manipulator `setfill` to change the padding character for fields larger than the values being printed  
using namespace directive  
using signal handling

using Standard Library functions to perform a heapsort  
using stream manipulator uppercase  
using template functions  
using variable-length argument lists  
Using virtual base classes  
utility function demonstration  
utility make

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

va\_arg 2nd  
va\_end 2nd  
va\_list 2nd  
va\_start 2nd  
validation 2nd  
validity checking  
valign attribute of element th

value  
value attribute of element input  
value of a variable 2nd  
value of an array element  
value of an attribute  
value\_type 2nd

variable  
variable arguments header <cstdarg>

variable name 2nd  
    argument  
    parameter  
variable size  
variable type  
variable-length argument list 2nd

VAX VMS

vector class 2nd  
    capacity function 2nd  
    push\_back function  
    rbegin function  
    rend function  
vector class template  
vector class template element-manipulation functions

verb phrase in requirements document  
verbs in a problem statement  
verbs in a system specification  
vertical spacing 2nd  
vertical tab ('v') 2nd  
vi text editor 2nd  
video I/O  
viewcart.cgi  
virtual base class 2nd 3rd 4th  
virtual destructor  
virtual directory  
virtual function 2nd 3rd 4th 5th 6th 7th 8th  
virtual function call  
virtual function call illustrated  
virtual function table (vtable)  
virtual inheritance  
virtual memory 2nd 3rd  
virtual memory operating systems  
visibility in the UML  
visibility marker in the UML  
Visual Basic .NET  
Visual C++ .NET  
Visual C++ home page 2nd  
Visual Studio .NET  
    Quick Info box  
visualizing recursion  
VMS  
void \* 2nd  
void keyword 2nd  
void return type  
volatile qualifier 2nd  
volume of a cube  
vtable 2nd 3rd 4th  
vtable pointer

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

[◀ PREV](#)

[NEXT ▶](#)

## Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [X] [Y]  
[Z]

waiting line

walk a list

"walk off" either end of an array

"warehouse" section of the computer

warning

watch debugger command

Watch window (Visual C++ .NET debugger) 2nd

waterfall model

wchar\_t character type 2nd

"weakest" iterator type 2nd

Web address

Web browser

Web form

Web server 2nd 3rd 4th 5th

[webstore.ansi.org/ansidocstore/default.asp](http://webstore.ansi.org/ansidocstore/default.asp)

what function of class runtime\_error

what virtual function of class exception 2nd

while repetition statement 2nd 3rd 4th 5th 6th

white-space character 2nd 3rd 4th 5th 6th 7th 8th 9th 10th

    blank

    newline

    tab

whole number

whole/part relationship

width attribute of element img

width attribute of element table

width implicitly set to 0

width member function of class ios\_base 2nd

width of a bit field

width of random number range  
width setting  
width-to-height ratio  
Windows 2nd 3rd  
Windows 2000  
Withdrawal class (ATM case study) 2nd 3rd 4th 5th 6th 7th 8th 9th 10th 11th 12th 13th 14th 15th 16th  
17th 18th  
word 2nd  
word boundary  
word equivalent of a check amount  
word processing 2nd  
Wordpad  
workflow of a portion of a software system  
workflow of an object in the UML  
workstation  
World Wide Web (WWW)  
World Wide Web resources for STL  
worst-case runtime for an algorithm  
wraparound  
write function of ostream 2nd 3rd 4th  
writing data randomly to a random-access file  
[www.accu.informika.ru/resources/public/terse/cpp.htm](http://www.accu.informika.ru/resources/public/terse/cpp.htm)  
[www.acm.org/crossroads/xrds3-2/ovp32.html](http://www.acm.org/crossroads/xrds3-2/ovp32.html)  
[www.borland.com](http://www.borland.com)  
[www.borland.com/bcppbuilder](http://www.borland.com/bcppbuilder)  
[www.codearchive.com/list.php?go=0708](http://www.codearchive.com/list.php?go=0708)  
[www.compilers.net](http://www.compilers.net)  
[www.cuj.com](http://www.cuj.com)  
[www.deitel.com](http://www.deitel.com) 2nd 3rd  
[www.deitel.com/newsletter/subscribe.html](http://www.deitel.com/newsletter/subscribe.html) 2nd  
[www.devx.com](http://www.devx.com)  
[www.forum.nokia.com/main/0,6566,050\\_20,00.html](http://www.forum.nokia.com/main/0,6566,050_20,00.html)  
[www.gametutorials.com/Tutorials/GT/GT\\_Pg1.htm](http://www.gametutorials.com/Tutorials/GT/GT_Pg1.htm)  
[www.hal9k.com/cug](http://www.hal9k.com/cug)  
[www.jasc.com](http://www.jasc.com)  
[www.kai.com/C\\_plus\\_plus](http://www.kai.com/C_plus_plus)  
[www.mathtools.net/C\\_C\\_\\_/Games/](http://www.mathtools.net/C_C__/Games/)  
[www.metrowerks.com](http://www.metrowerks.com)  
[www.msdn.microsoft.com/visualc/](http://www.msdn.microsoft.com/visualc/)  
[www.omg.org](http://www.omg.org) 2nd  
[www.prenhall.com/deitel](http://www.prenhall.com/deitel)  
[www.research.att.com/~bs/homepage.html](http://www.research.att.com/~bs/homepage.html)

W

[www.symbian.com/developer/development/cppdev.html](http://www.symbian.com/developer/development/cppdev.html)  
[www.thefreecountry.com/developercity/ccompilers.shtml](http://www.thefreecountry.com/developercity/ccompilers.shtml)  
[www.uml.org](http://www.uml.org)  
[www.unicode.org](http://www.unicode.org)  
[www.w3.org/markup](http://www.w3.org/markup)  
[www.w3.org/TR/xhtml11](http://www.w3.org/TR/xhtml11)  
[www.w3schools.com/xhtml/default.asp](http://www.w3schools.com/xhtml/default.asp)  
[www.xhtml.org](http://www.xhtml.org)

 PREV

NEXT 

page footer

---

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

## Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]

XHTML (Extensible Hypertext Markup Language)

XHTML (Extensible HyperText Markup Language) 2nd 3rd

comment

document

elements

form 2nd

Recommendation

table

tag

xor operator keyword

xor\_eq operator keyword

 PREV

NEXT 

page footer

The CHM file was converted to HTML by **chm2web** software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

NEXT 

## Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)] [[Z](#)]

yellow arrow in break mode

 PREV

NEXT 

page footer

The CHM file was converted to HTML by [chm2web](#) software.

Click [here](#) to show toolbars of the Web Online Help System: [show toolbars](#)

 PREV

## Index

[[SYMBOL](#)] [[A](#)] [[B](#)] [[C](#)] [[D](#)] [[E](#)] [[F](#)] [[G](#)] [[H](#)] [[I](#)] [[J](#)] [[K](#)] [[L](#)] [[M](#)] [[N](#)] [[O](#)] [[P](#)] [[Q](#)] [[R](#)] [[S](#)] [[T](#)] [[U](#)] [[V](#)] [[W](#)] [[X](#)] [[Y](#)]

[[Z](#)]

zero-based counting

zero-width bit field

zeros and ones

zeroth element

 PREV

page footer

The CHM file was converted to HTML by **chm2web** software.