

DEITEL® DEVELOPER SERIES

C++11

for Programmers

Introducing the New
C++11 Standard

PAUL DEITEL • HARVEY DEITEL

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



C++11 FOR PROGRAMMERS
SECOND EDITION
DEITEL[®] DEVELOPER SERIES

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U. S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the U. S., please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

On file

© 2014 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

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

ISBN-10: 0-13-343985-2

Text printed in the United States at RR Donnelley in Crawfordsville, Indiana..

First printing, February 2013

C++11 FOR PROGRAMMERS

SECOND EDITION

DEITEL[®] DEVELOPER SERIES

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.



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

Trademarks

DEITEL, the double-thumbs-up bug and DIVE INTO are registered trademarks of Deitel and Associates, Inc.

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

Throughout this book, trademarks are used. Rather than put a trademark symbol in every occurrence of a trademarked name, we state that we are using the names in an editorial fashion only and to the benefit of the trademark owner, with no intention of infringement of the trademark.

To our review team:

Dean Michael Berris

Danny Kalev

Linda M. Krause

James P. McNellis

Robert C. Seacord

José Antonio González Seco

We are grateful for your guidance and expertise.

Paul and Harvey Deitel

This page intentionally left blank

Contents

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

Preface	xix
1 Introduction	1
1.1 Introduction	2
1.2 C++	2
1.3 Object Technology	3
1.4 Typical C++ Development Environment	6
1.5 Test-Driving a C++ Application	9
1.6 Operating Systems	15
1.6.1 Windows—A Proprietary Operating System	15
1.6.2 Linux—An Open-Source Operating System	15
1.6.3 Apple’s OS X; Apple’s iOS for iPhone®, iPad® and iPod Touch® Devices	16
1.6.4 Google’s Android	16
1.7 C++11 and the Open Source Boost Libraries	17
1.8 Web Resources	18
2 Introduction to C++ Programming, Input/Output and Operators	19
2.1 Introduction	20
2.2 First Program in C++: Printing a Line of Text	20
2.3 Modifying Our First C++ Program	23
2.4 Another C++ Program: Adding Integers	24
2.5 Arithmetic	28
2.6 Decision Making: Equality and Relational Operators	32
2.7 Wrap-Up	35
3 Introduction to Classes, Objects and Strings	37
3.1 Introduction	38
3.2 Defining a Class with a Member Function	38
3.3 Defining a Member Function with a Parameter	41
3.4 Data Members, <i>set</i> Member Functions and <i>get</i> Member Functions	44
3.5 Initializing Objects with Constructors	50

3.6	Placing a Class in a Separate File for Reusability	54
3.7	Separating Interface from Implementation	58
3.8	Validating Data with <i>set</i> Functions	63
3.9	Wrap-Up	68

4 Control Statements: Part I; Assignment, ++ and -- Operators 69

4.1	Introduction	70
4.2	Control Structures	70
4.3	if Selection Statement	73
4.4	if...else Double-Selection Statement	74
4.5	while Repetition Statement	78
4.6	Counter-Controlled Repetition	79
4.7	Sentinel-Controlled Repetition	85
4.8	Nested Control Statements	92
4.9	Assignment Operators	95
4.10	Increment and Decrement Operators	96
4.11	Wrap-Up	99

5 Control Statements: Part 2; Logical Operators 100

5.1	Introduction	101
5.2	Essentials of Counter-Controlled Repetition	101
5.3	for Repetition Statement	102
5.4	Examples Using the for Statement	106
5.5	do...while Repetition Statement	110
5.6	switch Multiple-Selection Statement	112
5.7	break and continue Statements	121
5.8	Logical Operators	122
5.9	Confusing the Equality (==) and Assignment (=) Operators	127
5.10	Wrap-Up	128

6 Functions and an Introduction to Recursion 129

6.1	Introduction	130
6.2	Math Library Functions	130
6.3	Function Definitions with Multiple Parameters	132
6.4	Function Prototypes and Argument Coercion	137
6.5	C++ Standard Library Headers	139
6.6	Case Study: Random Number Generation	141
6.7	Case Study: Game of Chance; Introducing enum	146
6.8	C++11 Random Numbers	151
6.9	Storage Classes and Storage Duration	152
6.10	Scope Rules	155
6.11	Function Call Stack and Activation Records	158

6.12	Functions with Empty Parameter Lists	162
6.13	Inline Functions	163
6.14	References and Reference Parameters	164
6.15	Default Arguments	167
6.16	Unary Scope Resolution Operator	169
6.17	Function Overloading	170
6.18	Function Templates	173
6.19	Recursion	175
6.20	Example Using Recursion: Fibonacci Series	179
6.21	Recursion vs. Iteration	182
6.22	Wrap-Up	184

7 Class Templates array and vector; Catching Exceptions **185**

7.1	Introduction	186
7.2	arrays	186
7.3	Declaring arrays	188
7.4	Examples Using arrays	188
7.4.1	Declaring an array and Using a Loop to Initialize the array's Elements	188
7.4.2	Initializing an array in a Declaration with an Initializer List	189
7.4.3	Specifying an array's Size with a Constant Variable and Setting array Elements with Calculations	190
7.4.4	Summing the Elements of an array	192
7.4.5	Using Bar Charts to Display array Data Graphically	193
7.4.6	Using the Elements of an array as Counters	195
7.4.7	Using arrays to Summarize Survey Results	196
7.4.8	Static Local arrays and Automatic Local arrays	198
7.5	Range-Based for Statement	200
7.6	Case Study: Class GradeBook Using an array to Store Grades	202
7.7	Sorting and Searching arrays	209
7.8	Multidimensional arrays	211
7.9	Case Study: Class GradeBook Using a Two-Dimensional array	214
7.10	Introduction to C++ Standard Library Class Template vector	221
7.11	Wrap-Up	227

8 Pointers **228**

8.1	Introduction	229
8.2	Pointer Variable Declarations and Initialization	229
8.3	Pointer Operators	231
8.4	Pass-by-Reference with Pointers	233
8.5	Built-In Arrays	238
8.6	Using const with Pointers	240
8.6.1	Nonconstant Pointer to Nonconstant Data	241
8.6.2	Nonconstant Pointer to Constant Data	241

x Contents

8.6.3	Constant Pointer to Nonconstant Data	243
8.6.4	Constant Pointer to Constant Data	243
8.7	sizeof Operator	244
8.8	Pointer Expressions and Pointer Arithmetic	247
8.9	Relationship Between Pointers and Built-In Arrays	249
8.10	Pointer-Based Strings	252
8.11	Wrap-Up	255

9 Classes: A Deeper Look; Throwing Exceptions 256

9.1	Introduction	257
9.2	Time Class Case Study	258
9.3	Class Scope and Accessing Class Members	264
9.4	Access Functions and Utility Functions	265
9.5	Time Class Case Study: Constructors with Default Arguments	266
9.6	Destructors	272
9.7	When Constructors and Destructors Are Called	272
9.8	Time Class Case Study: A Subtle Trap—Returning a Reference or a Pointer to a private Data Member	276
9.9	Default Memberwise Assignment	279
9.10	const Objects and const Member Functions	281
9.11	Composition: Objects as Members of Classes	283
9.12	friend Functions and friend Classes	289
9.13	Using the this Pointer	291
9.14	static Class Members	297
9.15	Wrap-Up	302

10 Operator Overloading; Class string 303

10.1	Introduction	304
10.2	Using the Overloaded Operators of Standard Library Class string	305
10.3	Fundamentals of Operator Overloading	308
10.4	Overloading Binary Operators	309
10.5	Overloading the Binary Stream Insertion and Stream Extraction Operators	310
10.6	Overloading Unary Operators	314
10.7	Overloading the Unary Prefix and Postfix ++ and -- Operators	315
10.8	Case Study: A Date Class	316
10.9	Dynamic Memory Management	321
10.10	Case Study: Array Class	323
	10.10.1 Using the Array Class	324
	10.10.2 Array Class Definition	328
10.11	Operators as Member vs. Non-Member Functions	336
10.12	Converting Between Types	337
10.13	explicit Constructors and Conversion Operators	338
10.14	Overloading the Function Call Operator ()	340
10.15	Wrap-Up	341

11 Object-Oriented Programming: Inheritance 342

11.1	Introduction	343
11.2	Base Classes and Derived Classes	343
11.3	Relationship between Base and Derived Classes	346
11.3.1	Creating and Using a <code>CommissionEmployee</code> Class	346
11.3.2	Creating a <code>BasePlusCommissionEmployee</code> Class Without Using Inheritance	351
11.3.3	Creating a <code>CommissionEmployee–BasePlusCommissionEmployee</code> Inheritance Hierarchy	357
11.3.4	<code>CommissionEmployee–BasePlusCommissionEmployee</code> Inheritance Hierarchy Using <code>protected</code> Data	361
11.3.5	<code>CommissionEmployee–BasePlusCommissionEmployee</code> Inheritance Hierarchy Using <code>private</code> Data	364
11.4	Constructors and Destructors in Derived Classes	369
11.5	<code>public</code> , <code>protected</code> and <code>private</code> Inheritance	371
11.6	Software Engineering with Inheritance	372
11.7	Wrap-Up	372

12 Object-Oriented Programming: Polymorphism 374

12.1	Introduction	375
12.2	Introduction to Polymorphism: Polymorphic Video Game	376
12.3	Relationships Among Objects in an Inheritance Hierarchy	376
12.3.1	Invoking Base-Class Functions from Derived-Class Objects	377
12.3.2	Aiming Derived-Class Pointers at Base-Class Objects	380
12.3.3	Derived-Class Member-Function Calls via Base-Class Pointers	381
12.3.4	Virtual Functions and Virtual Destructors	383
12.4	Type Fields and <code>switch</code> Statements	390
12.5	Abstract Classes and Pure <code>virtual</code> Functions	390
12.6	Case Study: Payroll System Using Polymorphism	392
12.6.1	Creating Abstract Base Class <code>Employee</code>	393
12.6.2	Creating Concrete Derived Class <code>SalariedEmployee</code>	397
12.6.3	Creating Concrete Derived Class <code>CommissionEmployee</code>	399
12.6.4	Creating Indirect Concrete Derived Class <code>BasePlusCommissionEmployee</code>	401
12.6.5	Demonstrating Polymorphic Processing	403
12.7	(Optional) Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood”	407
12.8	Case Study: Payroll System Using Polymorphism and Runtime Type Information with Downcasting, <code>dynamic_cast</code> , <code>typeid</code> and <code>typeid</code>	410
12.9	Wrap-Up	414

13 Stream Input/Output: A Deeper Look 415

13.1	Introduction	416
------	--------------	-----

13.2	Streams	417
13.2.1	Classic Streams vs. Standard Streams	417
13.2.2	iostream Library Headers	418
13.2.3	Stream Input/Output Classes and Objects	418
13.3	Stream Output	420
13.3.1	Output of char * Variables	421
13.3.2	Character Output Using Member Function put	421
13.4	Stream Input	422
13.4.1	get and getline Member Functions	422
13.4.2	istream Member Functions peek, putback and ignore	425
13.4.3	Type-Safe I/O	425
13.5	Unformatted I/O Using read, write and gcount	425
13.6	Introduction to Stream Manipulators	426
13.6.1	Integral Stream Base: dec, oct, hex and setbase	427
13.6.2	Floating-Point Precision (precision, setprecision)	427
13.6.3	Field Width (width, setw)	429
13.6.4	User-Defined Output Stream Manipulators	430
13.7	Stream Format States and Stream Manipulators	431
13.7.1	Trailing Zeros and Decimal Points (showpoint)	432
13.7.2	Justification (left, right and internal)	433
13.7.3	Padding (fill, setfill)	435
13.7.4	Integral Stream Base (dec, oct, hex, showbase)	436
13.7.5	Floating-Point Numbers; Scientific and Fixed Notation (scientific, fixed)	437
13.7.6	Uppercase/Lowercase Control (uppercase)	438
13.7.7	Specifying Boolean Format (boolalpha)	438
13.7.8	Setting and Resetting the Format State via Member Function flags	439
13.8	Stream Error States	440
13.9	Tying an Output Stream to an Input Stream	443
13.10	Wrap-Up	443

14 File Processing **444**

14.1	Introduction	445
14.2	Files and Streams	445
14.3	Creating a Sequential File	446
14.4	Reading Data from a Sequential File	450
14.5	Updating Sequential Files	456
14.6	Random-Access Files	456
14.7	Creating a Random-Access File	457
14.8	Writing Data Randomly to a Random-Access File	462
14.9	Reading from a Random-Access File Sequentially	464
14.10	Case Study: A Transaction-Processing Program	466
14.11	Object Serialization	473
14.12	Wrap-Up	473

15 Standard Library Containers and Iterators 474

15.1	Introduction	475
15.2	Introduction to Containers	476
15.3	Introduction to Iterators	480
15.4	Introduction to Algorithms	485
15.5	Sequence Containers	485
15.5.1	vector Sequence Container	486
15.5.2	list Sequence Container	494
15.5.3	deque Sequence Container	498
15.6	Associative Containers	500
15.6.1	multiset Associative Container	501
15.6.2	set Associative Container	504
15.6.3	multimap Associative Container	505
15.6.4	map Associative Container	507
15.7	Container Adapters	509
15.7.1	stack Adapter	509
15.7.2	queue Adapter	511
15.7.3	priority_queue Adapter	512
15.8	Class bitset	513
15.9	Wrap-Up	515

16 Standard Library Algorithms 517

16.1	Introduction	518
16.2	Minimum Iterator Requirements	518
16.3	Algorithms	520
16.3.1	fill, fill_n, generate and generate_n	520
16.3.2	equal, mismatch and lexicographical_compare	522
16.3.3	remove, remove_if, remove_copy and remove_copy_if	524
16.3.4	replace, replace_if, replace_copy and replace_copy_if	527
16.3.5	Mathematical Algorithms	529
16.3.6	Basic Searching and Sorting Algorithms	533
16.3.7	swap, iter_swap and swap_ranges	537
16.3.8	copy_backward, merge, unique and reverse	538
16.3.9	inplace_merge, unique_copy and reverse_copy	541
16.3.10	Set Operations	543
16.3.11	lower_bound, upper_bound and equal_range	546
16.3.12	Heapsort	548
16.3.13	min, max, minmax and minmax_element	551
16.4	Function Objects	553
16.5	Lambda Expressions	556
16.6	Standard Library Algorithm Summary	557
16.7	Wrap-Up	559

17 Exception Handling: A Deeper Look 560

17.1	Introduction	561
------	--------------	-----

17.2	Example: Handling an Attempt to Divide by Zero	561
17.3	Rethrowing an Exception	567
17.4	Stack Unwinding	568
17.5	When to Use Exception Handling	570
17.6	Constructors, Destructors and Exception Handling	571
17.7	Exceptions and Inheritance	572
17.8	Processing new Failures	572
17.9	Class <code>unique_ptr</code> and Dynamic Memory Allocation	575
17.10	Standard Library Exception Hierarchy	578
17.11	Wrap-Up	579

18 Introduction to Custom Templates 581

18.1	Introduction	582
18.2	Class Templates	582
18.3	Function Template to Manipulate a Class-Template Specialization Object	587
18.4	Nontype Parameters	589
18.5	Default Arguments for Template Type Parameters	589
18.6	Overloading Function Templates	589
18.7	Wrap-Up	590

19 Class `string` and String Stream Processing: A Deeper Look 591

19.1	Introduction	592
19.2	<code>string</code> Assignment and Concatenation	593
19.3	Comparing strings	595
19.4	Substrings	598
19.5	Swapping strings	598
19.6	<code>string</code> Characteristics	599
19.7	Finding Substrings and Characters in a <code>string</code>	601
19.8	Replacing Characters in a <code>string</code>	603
19.9	Inserting Characters into a <code>string</code>	605
19.10	Conversion to Pointer-Based <code>char *</code> Strings	606
19.11	Iterators	607
19.12	String Stream Processing	609
19.13	C++11 Numeric Conversion Functions	612
19.14	Wrap-Up	613

20 Bits, Characters, C Strings and structs 615

20.1	Introduction	616
20.2	Structure Definitions	616
20.3	<code>typedef</code>	618
20.4	Example: Card Shuffling and Dealing Simulation	618
20.5	Bitwise Operators	621

20.6	Bit Fields	630
20.7	Character-Handling Library	633
20.8	C String-Manipulation Functions	639
20.9	C String-Conversion Functions	646
20.10	Search Functions of the C String-Handling Library	651
20.11	Memory Functions of the C String-Handling Library	655
20.12	Wrap-Up	659

21 Other Topics **660**

21.1	Introduction	661
21.2	<code>const_cast</code> Operator	661
21.3	<code>mutable</code> Class Members	663
21.4	namespaces	665
21.5	Operator Keywords	668
21.6	Pointers to Class Members (<code>.*</code> and <code>->*</code>)	670
21.7	Multiple Inheritance	672
21.8	Multiple Inheritance and <code>virtual</code> Base Classes	677
21.9	Wrap-Up	681

22 ATM Case Study, Part 1: Object-Oriented Design with the UML **682**

22.1	Introduction	683
22.2	Introduction to Object-Oriented Analysis and Design	683
22.3	Examining the ATM Requirements Document	684
22.4	Identifying the Classes in the ATM Requirements Document	691
22.5	Identifying Class Attributes	698
22.6	Identifying Objects' States and Activities	703
22.7	Identifying Class Operations	707
22.8	Indicating Collaboration Among Objects	714
22.9	Wrap-Up	721

23 ATM Case Study, Part 2: Implementing an Object-Oriented Design **725**

23.1	Introduction	726
23.2	Starting to Program the Classes of the ATM System	726
23.3	Incorporating Inheritance into the ATM System	732
23.4	ATM Case Study Implementation	739
23.4.1	Class ATM	740
23.4.2	Class Screen	747
23.4.3	Class Keypad	749
23.4.4	Class CashDispenser	750
23.4.5	Class DepositSlot	752

23.4.6	Class Account	753
23.4.7	Class BankDatabase	755
23.4.8	Class Transaction	759
23.4.9	Class BalanceInquiry	761
23.4.10	Class Withdrawal	763
23.4.11	Class Deposit	768
23.4.12	Test Program ATMCASEStudy.cpp	771
23.5	Wrap-Up	771

A Operator Precedence and Associativity 774

B ASCII Character Set 777

C Fundamental Types 778

D Number Systems 780

D.1	Introduction	781
D.2	Abbreviating Binary Numbers as Octal and Hexadecimal Numbers	784
D.3	Converting Octal and Hexadecimal Numbers to Binary Numbers	785
D.4	Converting from Binary, Octal or Hexadecimal to Decimal	785
D.5	Converting from Decimal to Binary, Octal or Hexadecimal	786
D.6	Negative Binary Numbers: Two's Complement Notation	788

E Preprocessor 790

E.1	Introduction	791
E.2	#include Preprocessing Directive	791
E.3	#define Preprocessing Directive: Symbolic Constants	792
E.4	#define Preprocessing Directive: Macros	792
E.5	Conditional Compilation	794
E.6	#error and #pragma Preprocessing Directives	795
E.7	Operators # and ##	796
E.8	Predefined Symbolic Constants	796
E.9	Assertions	797
E.10	Wrap-Up	797

Index 799

Online Chapters and Appendices

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

F	C Legacy Code Topics	F-1
G	UML 2: Additional Diagram Types	G-1
H	Using the Visual Studio Debugger	H-1
I	Using the GNU C++ Debugger	I-1
J	Using the Xcode Debugger	J-1
K	Test Driving a C++ Program on Mac OS X	K-1

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

This page intentionally left blank

Preface

“The chief merit of language is clearness ...”

—Galen

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

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

www.deitel.com/books/cpp11fp

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

deitel@deitel.com

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

Features

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

C++11 Standard

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



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

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

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

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

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

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

www.informit.com/title/9780133439854

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

Object-Oriented Programming

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

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

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

Pedagogic Features

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

Other Features

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

Secure C++ Programming

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

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

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

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

www.securecoding.cert.org

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

Training Approach

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

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

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

```

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

```

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

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

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

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

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

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



Good Programming Practice

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



Common Programming Error

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



Error-Prevention Tip

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



Performance Tip

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



Portability Tip

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



Software Engineering Observation

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

Online Chapter and Appendices

The following chapter and appendices are available online:

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

To access the online chapter and appendices, go to:

www.informit.com/register

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

Obtaining the Software Used in C++11 for Programmers

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

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

www.microsoft.com/express

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

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

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

www.deitel.com/livelessons

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

Acknowledgments

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

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

Reviewers

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

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

deitel@deitel.com

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

Paul Deitel

Harvey Deitel

About the Authors

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

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

Deitel® Dive-Into® Series Corporate Training

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

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

deitel@deitel.com

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

www.deitel.com/training

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

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

www.informit.com/store/sales.aspx

This page intentionally left blank

2

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

Objectives

In this chapter you'll:

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

2.1 Introduction	2.5 Arithmetic
2.2 First Program in C++: Printing a Line of Text	2.6 Decision Making: Equality and Relational Operators
2.3 Modifying Our First C++ Program	2.7 Wrap-Up
2.4 Another C++ Program: Adding Integers	

2.1 Introduction

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

Compiling and Running Programs

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

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

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

```

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

```

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

Fig. 2.1 | Text-printing program.

Comments

Lines 1 and 2

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

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

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

#include Preprocessing Directive

Line 3

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

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



Common Programming Error 2.1

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

Blank Lines and White Space

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

The main Function

Line 5

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

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

Line 6

```
int main()
```

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

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

An Output Statement

Line 8

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

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

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



Good Programming Practice 2.1

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



Good Programming Practice 2.2

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

The `std` Namespace

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

The Stream Insertion Operator and Escape Sequences

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

Escape sequence	Description
<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.
<code>\'</code>	Single quote. Used to print a single quote character.
<code>\"</code>	Double quote. Used to print a double quote character.

Fig. 2.2 | Escape sequences.

The return Statement

Line 10

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

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

2.3 Modifying Our First C++ Program

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

Printing a Single Line of Text with Multiple Statements

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

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

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

```

5 // function main begins program execution
6 int main()
7 {
8     std::cout << "Welcome ";
9     std::cout << "to C++!\n";
10 } // end function main

```

```

Welcome to C++!

```

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

Printing Multiple Lines of Text with a Single Statement

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

```

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

```

```

Welcome
to

C++!

```

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

2.4 Another C++ Program: Adding Integers

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

```

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

```

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

```

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

```

```

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

```

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

Variable Declarations

Lines 9–11

```

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

```

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



Error-Prevention Tip 2.1

Although it's not always necessary to initialize every variable explicitly, doing so will help you avoid many kinds of problems.

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

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

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

**Good Programming Practice 2.3**

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

Fundamental Types

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

Identifiers

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

**Portability Tip 2.1**

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

**Good Programming Practice 2.4**

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

**Good Programming Practice 2.5**

Avoid using abbreviations in identifiers. This improves program readability.

**Good Programming Practice 2.6**

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

Placement of Variable Declarations

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

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

could have been placed immediately before line 14

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

Obtaining the First Value from the User

Line 13

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

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

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

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

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

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

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

Obtaining the Second Value from the User

Line 16

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

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

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

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

Calculating the Sum of the Values Input by the User

The assignment statement in line 19

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

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



Good Programming Practice 2.7

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

Displaying the Result

Line 21

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

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

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

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

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

thus eliminating the need for the variable `sum`.

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

2.5 Arithmetic

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

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

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

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

Fig. 2.6 | Arithmetic operators.

Arithmetic Expressions in Straight-Line Form

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

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

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

Parentheses for Grouping Subexpressions

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

Rules of Operator Precedence

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

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

$$(a * (b + c))$$

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

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

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

$$a + b + c$$

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

Operator(s)	Operation(s)	Order of evaluation (precedence)
()	Parentheses	Evaluated first. If the parentheses are <i>nested</i> , such as in the expression $a * (b + c / d + e)$, the expression in the <i>innermost</i> pair is evaluated first. [Caution: If you have an expression such as $(a + b) * (c - d)$ in which two sets of parentheses are not nested, but appear "on the same level," the C++ Standard does <i>not</i> specify the order in which these parenthesized subexpressions will be evaluated.]
*	Multiplication	Evaluated second. If there are several, they're evaluated left to right.
/	Division	
%	Modulus	
+	Addition	Evaluated last. If there are several, they're evaluated left to right.
-	Subtraction	

Fig. 2.7 | Precedence of arithmetic operators.

Sample Algebraic and C++ Expressions

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

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

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

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

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

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

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

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

6
1
2
4
3
5

The circled numbers indicate the order in which C++ applies the operators. The multiplication, modulus and division are evaluated *first* in left-to-right order (i.e., they associate from

left to right) because they have *higher precedence* than addition and subtraction. The addition and subtraction are applied next. These are also applied left to right. The assignment operator is applied *last* because its precedence is *lower* than that of any of the arithmetic operators.

Evaluation of a Second-Degree Polynomial

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

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

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

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

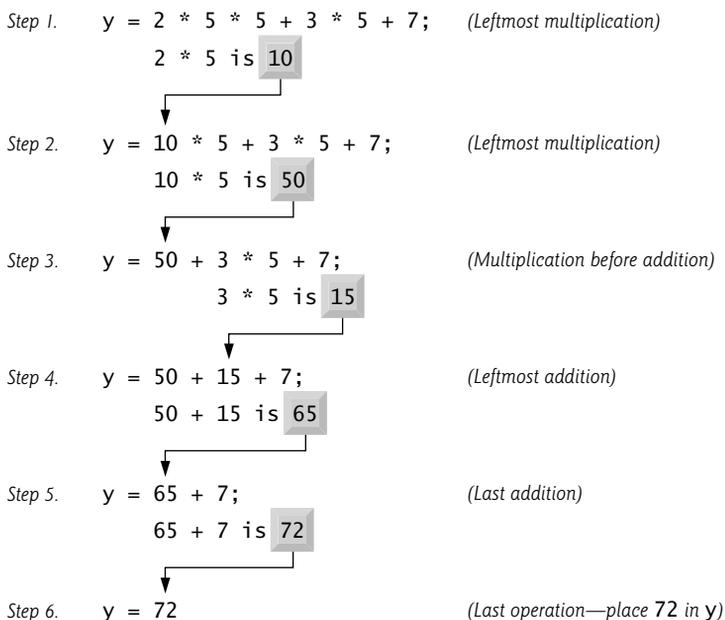


Fig. 2.8 | Order in which a second-degree polynomial is evaluated.

Redundant Parentheses

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

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

2.6 Decision Making: Equality and Relational Operators

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

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

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

Fig. 2.9 | Relational and equality operators.



Common Programming Error 2.2

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



Common Programming Error 2.3

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

Using the **if** Statement

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

```
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 = 0; // first integer to compare (initialized to 0)
14     int number2 = 0; // second integer to compare (initialized to 0)
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 } // end function main
```

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

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

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

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

using Declarations

Lines 6–8

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

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

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

```
using namespace std;
```

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

Variable Declarations and Reading the Inputs from the User

Lines 13–14

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

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

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

Comparing NumbersThe `if` statement in lines 19–20

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

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

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

**Common Programming Error 2.4**

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

White Space

Recall that whitespace characters, such as tabs, newlines and spaces, are normally ignored by the compiler. So, statements may be split over several lines and may be spaced according to your preferences. It's a syntax error to split identifiers, strings (such as "he11o") and constants (such as the number 1000) over several lines.



Good Programming Practice 2.8

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

Operator Precedence

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

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

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



Good Programming Practice 2.9

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

2.7 Wrap-Up

You learned many important basic features of C++ in this chapter, including displaying data on the screen, inputting data from the keyboard and declaring variables of fundamen-

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

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

This page intentionally left blank

Index

Symbols

-- postfix decrement operator 96
-- prefix decrement operator 96
^ (bitwise exclusive OR operator) 621, 669
^= (bitwise exclusive OR assignment operator) 515, 629, 669
, (comma operator) 104
:: (binary scope resolution operator) 298
:: (scope resolution operator) 60
:: (unary scope resolution operator) 668, 672
:: (unary scope resolution operator) 169
! (logical NOT operator) 123, 124, 668
 truth table 125
!= (inequality operator) 32, 668
?: (ternary conditional operator) 75, 181
.* (pointer-to-member operator) 670, 672
. * and ->* operators 670
'\0', null character 253
'\n', newline character 252
[] (operator for map) 507
* (multiplication operator) 29
* (pointer dereference or indirection operator) 232, 232
*= (multiplication assignment operator) 96
/ (division operator) 29
/* */ (C-style multiline comment) 21
// (single-line comment) 20
/= (division assignment operator) 96
\' (single-quote-character) escape sequence 23
\" (double-quote-character) escape sequence 23
\\ (backslash-character) escape sequence 23
\a (alert) escape sequence 23
\n (newline) escape sequence 23
\r (carriage-return) escape sequence 23
\t (tab) escape sequence 23
& (address operator) 669
& (bitwise AND) 621
& in a parameter list 166
& to declare reference 165
&, address operator 231, 232
&& (logical AND operator) 123, 181, 668
 truth table 123
&= (bitwise AND assignment operator) 515, 629, 669
preprocessor operator 791, 796
preprocessor operator 796
#pragma preprocessing directive 796
#undef preprocessing directive 794
% (modulus operator) 29

%= (modulus assignment operator) 96
+ (addition operator) 27, 29
++ (postfix increment operator) 96
 on an iterator 480
++ (prefix increment operator) 96
 on an iterator 480
+= (addition assignment operator) 95
 string concatenation 595
< (less-than operator) 32
<< (left-shift operator) 621
<< (stream insertion operator) 22, 28
<<= (left-shift assignment operator) 629
<= (less-than-or-equal-to operator) 32
<algorithm> header 558
<algorithm> header 558
<forward_list> header 494
<unordered_map> header 505, 507
<unordered_set> header 501, 504
= 27, 35, 279, 308, 480
= (assignment operator) 27, 29, 125
-= (subtraction assignment operator) 96
== (equality operator) 32, 125
> (greater-than operator) 32
-> (member selection via pointer) 672
->* (pointer-to-member operator) 670
>= (greater-than-or-equal-to operator) 32
>> (right shift operator) 621
>>= (right shift with sign extension assignment operator) 629
| (bitwise inclusive OR operator) 621, 669
|= (bitwise inclusive OR assignment operator) 515, 629, 669
|| (logical OR operator) 123, 124, 668
 truth table 124
|| logical OR operator 181
~ (bitwise complement operator) 621, 669

Numerics

0x 432
0x 432
2-D array 211

A

abbreviating assignment expressions 95
abort 797
abort function 272, 574
absolute value 131
abstract base class 390, 391, 738
abstract class 390, 391, 392, 407
abstract operation in the UML 734
access a global variable 169
access function 265

access non-`static` class data members and member functions 301
access `private` member of a class 47
access privileges 241, 243
access specifier 39, 47, 291, 726
 `private` 47
 `protected` 258
 `public` 47
access the caller's data 164
access violation 476
accessor 49
Account class (ATM case study) 690, 693, 697, 699, 700, 708, 715, 716, 718, 720, 732, 772
accounts-receivable system 446
accumulate algorithm 529, 532, 553, 555, 559
accumulated outputs 28
action 74, 75
action expression 71
action expression in the UML 705
action of an object 704
action state 71
action state in the UML 705
action state symbol 71
activation in a UML sequence diagram 720
activation record 159
activity diagram 70, 71, 78, 105
 `do...while` statement 111
 `for` statement 106
 `if` statement 74
 `if...else` statement 75
 in the UML 690, 704, 706, 723
 sequence statement 71
 `switch` statement 120
 `while` statement 79
activity in the UML 690, 703, 707
activity of a portion of a software system 71
actor in use case in the UML 689
adapter 509
add a new account to a file 472
add an integer to a pointer 247
addition 29
addition assignment operator (`+=`) 95
addition program that displays the sum of two numbers 24
address of a bit field 633
address operator (`&`) 231, 232, 234, 308, 672
addressable storage unit 633
adjacent_difference algorithm 559
adjacent_find algorithm 558
aggregation 264, 696
aiming a derived-class pointer at a base-class object 381

- airline reservation system 456
- alert escape sequence ('\a') 23, 637
- algebraic expression 29
- <algorithm> header 141, 492
- algorithms 476, 485
 - accumulate 529, 532
 - all_of 533, 536
 - any_of 533, 536
 - binary_search 209, 533, 536
 - copy_backward 539
 - copy_n 541
 - count 529, 532
 - count_if 529, 532
 - equal 523
 - equal_range 546, 548
 - fill 520, 521
 - fill_n 520, 521
 - find 533, 535
 - find_if 533, 536
 - find_if_not 533, 537
 - for_each 529, 533
 - generate 520, 521
 - generate_n 520, 521
 - includes 544
 - inplace_merge 541, 542
 - is_heap 551
 - is_heap_until 551
 - iter_swap 537, 538
 - lexicographical_compare 524
 - lower_bound 548
 - make_heap 550
 - max 552
 - max_element 529, 532
 - merge 538, 540
 - min 552
 - min_element 529, 532
 - minmax 552
 - minmax_element 529, 532, 553
 - mismatch 522, 524
 - move 540
 - move_backward 540
 - none_of 533, 536
 - pop_heap 551
 - push_heap 551
 - random_shuffle 529, 531
 - remove 524, 526
 - remove_copy 526
 - remove_copy_if 524, 527, 541
 - remove_if 524, 526
 - replace 529
 - replace_copy 527, 529
 - replace_copy_if 527, 529
 - replace_if 527, 529
 - reverse 538, 541
 - reverse_copy 541, 542
 - separated from container 519
 - set_difference 543, 545
 - set_intersection 543, 545
 - set_symmetric_difference 543, 545
 - set_union 543, 546
 - sort 209, 533, 536
 - sort_heap 550
 - swap 537, 538
 - swap_ranges 537, 538
 - transform 529, 533
 - unique 538, 540
 - unique_copy 541, 542
 - upper_bound 548
- alias 166, 167, 609
 - for the name of an object 276
- alignment 617
- all 514
- all_of algorithm 533, 536, 558
- allocate 321
- allocate dynamic memory 575
- allocate memory 140, 321
- allocator 493
- allocator_type 479
- alphabetizing strings 643
- alter the flow of control 121
- ambiguity problem 672, 677
- American National Standards Institute (ANSI) 2
- American Standard Code for Information Interchange (ASCII) 116
- analysis stage of the software life cycle 688
- analyzing a project's requirements 683
- and operator keyword 668
- and_eq operator keyword 669
- "ANDed" 623
- Android 16
 - operating system 16
 - smartphone 16
- angle brackets (< and >) 173, 791
- angle brackets (< and >) in templates 583
- anonymous function objects 518
- ANSI (American National Standards Institute) 2
- ANSI/ISO 9899: 1990 2
- any 514
- any_of algorithm 533, 536, 558
- Apache Software Foundation 15
- append 595
- append data to a file 447, 448
- Apple Inc. 16
- Apple Macintosh 16
- argument coercion 138
- argument for a macro 792
- argument to a function 41
- arguments in correct order 136
- arguments passed to member-object constructors 283
- arithmetic assignment operators 95, 96
- arithmetic calculations 28
- arithmetic mean 30
- arithmetic operator 28
- arithmetic overflow 84, 570
- arithmetic overflow error 579
- arithmetic underflow error 579
- "arity" of an operator 309
- <array> header 140
- array
 - built-in 229, 238
 - name 250
 - notation for accessing elements 251
 - subscripting 251
- array 187
 - bounds checking 198
- Array class 324
- Array class definition with overloaded operators 328
- Array class member-function and friend function definitions 329
- array class template 186
- Array class test program 324
- array subscript operator ([]) 328
- arrow 71
- arrow member selection operator (->) 265
- arrow operator (->) 293
- arrowhead in a UML sequence diagram 720
- ASCII (American Standard Code for Information Interchange)
 - Character Set 116, 252, 422
- assert 797
- assign member function of class
 - string 593
- assign member function of list 498
- assign one iterator to another 484
- assigning addresses of base-class and derived-class objects to base-class and derived-class pointers 378
- assigning class objects 279
- assignment operator functions 333
- assignment operators 27, 35, 95, 279, 308, 480
 - *= multiplication assignment operator 96
 - /= division assignment operator 96
 - %= modulus assignment operator 96
 - += addition assignment operator 95
 - = subtraction assignment operator 96
- assignment statement 27, 97
- associate from left to right 35, 98
- associate from right to left 35, 98, 116
- association 508
- association (in the UML) 694, 695, 696, 728, 729
 - name 695
- associative container 480, 483, 500, 503
 - ordered 476, 500
 - unordered 476, 500
- associative container functions
 - count 503
 - equal_range 503
 - find 503
 - insert 503, 507
 - lower_bound 503
 - upper_bound 503
- associativity 124, 126
- associativity chart 35
- associativity not changed by overloading 309
- associativity of operators 29, 35
- asterisk (*) 28
- asynchronous call 717
- asynchronous event 570
- at member function 493, 514
 - class string 308, 595
 - class vector 226
- ATM (automated teller machine) case study 683, 688
- ATM class (ATM case study) 693, 694, 695, 699, 702, 703, 708, 714, 715, 716, 717, 718, 727
- ATM system 689, 690, 691, 692, 693, 698, 703, 707, 726
- atof 647
- atoi 647
- atol 648
- attribute 45, 728, 729
 - compartment in a class diagram 701
 - declaration in the UML 701, 703

attribute (cont.)
 in the UML 5, 41, 693, 698, 700, 701, 703, 707, 736
 name in the UML 701
 of a class 4
 of an object 5
 attributes of a variable 152
 auto keyword 213
 auto_ptr object manages dynamically allocated memory 577
 automated teller machine 456
 automated teller machine (ATM) 683, 684, 688
 user interface 684
 automatic array 188
 automatic array initialization 199
 automatic local array 199
 automatic local variable 153, 156, 167
 automatic object 571
 automatic storage class 152, 200
 automatic storage duration 153
 automatically destroyed 156
 average 30
 average calculation 79, 85
 avoid naming conflicts 291
 avoid repeating code 270

B

back member function of queue 511
 back member function of sequence containers 492
 back_inserter function template 540, 542
 backslash (\) 22, 794
 backslash escape sequence (\\) 23
 backward traversal 607
 bad member function 442
 bad_alloc exception 493, 572, 574, 578
 bad_cast exception 578
 bad_typeid exception 578
 badbit 449
 badbit of stream 422, 442
 BalanceInquiry class (ATM case study) 693, 696, 699, 700, 702, 704, 708, 715, 716, 717, 718, 727, 733, 734, 735
 Bank account program 466
 BankDatabase class (ATM case study) 693, 697, 699, 708, 715, 716, 717, 718, 720, 727, 729
 banking system 456
 bar chart 193, 194
 bar chart printing program 194
 bar of asterisks 193, 194
 base 2 621
 base case(s) 175, 179, 182
 base class 343, 345, 733
 base-class catch 578
 base-class constructor 370
 base-class exception 578
 base-class member accessibility in derived class 371
 base-class pointer to a derived-class object 389
 base-class private member 361
 base-class subobject 678
 base *e* 132

base specified for a stream 436
 base-10 number system 132, 432
 base-16 number system 432
 base-8 number system 432
 base-class initializer syntax 359
 base-class member function redefined in a derived class 368
 BasePlusCommissionEmployee class header 402
 BasePlusCommissionEmployee class implementation file 402
 BasePlusCommissionEmployee class represents an employee who receives a base salary in addition to a commission 352
 BasePlusCommissionEmployee class test program 355
 BasePlusCommissionEmployee class that inherits from class CommissionEmployee, which does not provide protected data 367
 basic searching and sorting algorithms of the Standard Library 533
 basic_ofstream class template 420, 446
 basic_ifstream class template 420, 445
 basic_ios class template 418, 677
 basic_iostream class template 418, 420, 446, 677
 basic_istream class template 418, 446, 677
 basic_istream class template 609
 basic_ofstream class template 446
 basic_ostringstream class template 420, 446, 677
 basic_ostringstream class template 609
 basic_string class template 592
 begin function 240
 begin iterator 608
 begin member function of class string 608
 begin member function of containers 479
 begin member function of first-class containers 480
 beginning of a file 452
 beginning of a stream 452
 behavior 707
 of a class 4
 behavior of the system 703, 704, 707, 716
 bell 23
 Bell Laboratories 2
 bidirectional iterator 482, 483, 494, 501, 504, 505, 518, 540, 541, 542
 bidirectional iterator operations 484
 bidirectional navigability in the UML 727
 binary (base 2) number system 781
 binary arithmetic operator 90
 binary function 553
 binary function object 553
 binary number 634
 binary number system 650
 binary operator 27, 28
 binary predicate function 497, 523, 532, 536, 540, 545, 550

binary_search algorithm 209, 533, 536, 558
 bit 616
 bit field 621, 630, 633
 bit-field manipulation 633
 bit-field member of structure 631
 bit fields save space 633
 bit manipulation 621
 bitand operator keyword 669
 bitor operator keyword 669
 "bits-and-bytes" level 621
 bitset 477, 513, 514, 515
 <bitset> header 140
 bitwise AND assignment 669
 bitwise AND assignment operator (&=) 629
 bitwise AND operator (&) 621, 621, 624, 626
 bitwise AND, bitwise inclusive-OR, bitwise exclusive-OR and bitwise complement operators 624
 bitwise assignment operator keywords 669
 bitwise assignment operators 515, 629
 bitwise complement 622, 669
 bitwise complement operator (~) 621, 624, 627, 629, 788
 bitwise exclusive OR 669
 bitwise exclusive OR assignment operator (\wedge)= 629
 bitwise exclusive OR operator (\wedge) 621, 624, 627
 bitwise inclusive OR 669
 bitwise inclusive OR assignment operator (\mid)= 629
 bitwise inclusive OR operator (\mid) 461, 621, 624, 626
 bitwise left-shift operator (<<) 304, 627
 bitwise logical OR 515
 bitwise operator keywords 669
 bitwise operators 621, 622, 629
 bitwise right-shift operator (>>) 304
 bitwise shift operator 627
 BlackBerry OS 15
 block 34, 44, 66, 77, 89, 153, 155, 156
 block is active 153
 block is exited 153
 block of data 655
 block of memory 498, 655
 block scope 155, 265
 variable 265
 body of a class definition 39
 body of a function 21, 22, 40
 body of a loop 102, 105
 Booch, Grady 684
 bool data type 74
 bool value false 74
 bool value true 74
 boolalpha stream manipulator 125, 432, 438
 Boolean attribute in the UML 699
 Boost C++ Libraries 17
 boundary of a storage unit 633
 bounds checking 198
 braces ({}), 22, 34, 66, 77, 89, 117
 braces in a do...while statement 112
 bracket ([]) 187
 break statement 118, 121

break statement exiting a **for** statement 121
 brittle software 364
 buffer is filled 420
 buffer is flushed 420
 buffer overflow 198
 buffered output 420
 buffered standard error stream 418
 buffering 443
 building-block approach 3
 built-in array 229, 238
 business-critical computing 565
 byte 621

C

.C extension 6
 C legacy code 791, 792, 797
 C-like pointer-based array 477
 C string 252
c_str member function of class **string** 607
 C++ 2
 C++ compiler 7
 C++ development environment 7, 8
 C++ preprocessor 7, 21
 C++ Standard Library 3
 <**string**> file 43
 class template **vector** 221
 header location 57
 headers 139
string class 43
 C++11 17, 178
all_of algorithm 536
 anonymous function objects 518
any_of algorithm 536
 associative container keys are immutable 477
auto keyword 213, 503
begin function 240, 489
cbegin container member function 489
end container member function 489
 compiler fix for types ending in >> 502
copy_n algorithm 541
crbegin container member function 490
crend container member function 490
 default type arguments for function template type parameters 589
 delegating constructor 271
end function 240, 489
find_if_not algorithm 537
forward_list class template 476, 494
 in-class initializer 120, 260
insert container member function (now returns an iterator) 493, 494
iota algorithm 559
is_heap algorithm 551
is_heap_until algorithm 551
 list initialization 94, 507
 list initialization of a return type 507
 list initialization of associative container 507

C++11 (cont.)
 list initializer 271
 list initializers 490
minmax algorithm 552
minmax_element algorithm 532, 553
move algorithm 540
 move assignment operator 478
 move constructor 478
move_backward algorithm 540
noexcept 571
none_of algorithm 536
 non-member container **swap** function 478
nullptr constant 230
override 384
 random-number generation 195
scoped_enum 150
shrink_to_fit container member function for **vector** and **deque** 490
 specifying an enum's integral type 150
stod function 612
stof function 612
stoi function 612
stol function 612
stold function 612
stoll function 612
stoul function 612
stoull function 612
to_string function 612
 trailing return types for functions 175
unique_ptr class template 575, 575, 578
unordered_multimap class template 477
unordered_multiset class template 477
unordered_set class template 477
 calculations 28, 70
 call a function 41
 call stack 242
 calling function (caller) 40, 47
 calling functions by reference 233
 camel case 39
capacity member function
 of **string** 601
 of **vector** 488
 capacity of a string 599
 Card Shuffling and Dealing simulation 616, 618, 620
 carriage return ('\r') escape sequence 23, 634, 637
 carry bit 788
 cascading member function calls 293, 294, 296
 cascading stream insertion operations 28
 case label 117, 118
 case sensitive 26
CashDispenser class (ATM case study) 693, 695, 699, 700, 708, 720
 casino 146
 <**cassert**> header 141, 797
cast 249
 downcast 383
 cast away **const**-ness 662
 cast expression 794

cast operator 85, 90, 139, 337, 338
 cast operator function 337
 cast variable visible in debugger 792
catch a base class object 578
catch all exceptions 579
catch block 226
catch clause (or handler) 566, 570
catch handler 564
catch related errors 572
catch(...) 579
cbegin member function of containers 479
cbegin member function of **vector** 489
 <**cctype**> header 140, 634
 CD 445
ceil function 131
ceil member function of containers 479
end member function of **vector** 489
cerr (standard error stream) 9, 66, 418, 419, 445
 <**cmath**> header 141
 chaining stream insertion operations 28
char** 648
char data type 26, 116, 139, 606, 621
char16_t 418
char32_t 418
 character 616
 character array 253, 606
 character constant 252
 character-handling functions 634
isdigit, **isalpha**, **isalnum** and **isxdigit** 635
islower, **isupper**, **tolower** and **toupper** 636
isspace, **isctrl**, **ispunct**, **isprint** and **isgraph** 637
 character presentation 141
 character sequences 456
 character set 120
 character string 22, 188
 character's numerical representation 116
 characters represented as numeric codes 644
 character-string manipulation 633
 checked access 595
cin (standard input stream) 9, 27, 418, 419, 445, 449
 function **getline** 254
cin.clear 442
cin.eof 422, 442
cin.get function 115, 116, 423
cin.tie function 443
 circular include 730
 class 4, 701, 707, 711, 726, 792
 attribute 45
 client-code programmer 62
 constructor 50
 data member 5, 45
 default constructor 51, 53
 define a constructor 52
 define a member function 38
 implementation programmer 62
 instance of 46
 interface 58
 interface described by function prototypes 58
 member function 38
 member-function implementations in a separate source-code file 59

- class (cont.)
 - name 728
 - naming convention 39
 - object of 46
 - public services 58
 - services 49
- class average on a quiz 79
- class average problem 79
- class definition 39
- class development 324
- class diagram
 - for the ATM system model 697, 722
 - in the UML 690, 693, 696, 698, 701, 708, 726, 729, 735, 736, 737
- class diagram (UML) 41
- class hierarchy 344, 389, 391
- class-implementation programmer 62
- class keyword 173, 583
- class libraries 109
- class library 372
- class members default to private access 616
- class scope 155, 261, 264
- class-scope variable is hidden 265
- class template 186, 582, 592
 - auto_ptr 575
 - definition 582
 - scope 585
 - specialization 582
 - Stack 583, 585
- class variable 207
- Classes 3
 - Array 324
 - bitset 477, 513, 514, 515
 - deque 485, 498
 - exception 562
 - forward_list 485, 494
 - invalid_argument 578
 - list 485, 494
 - multimap 505
 - out_of_range exception class 226
 - priority_queue 512
 - queue 511, 511
 - runtime_error 562, 570
 - set 504
 - stack 509
 - string 43
 - unique_ptr 575
 - vector 221
- classic stream libraries 417
- clear function of ios_base 442
- clear member function of containers 479
- clear member function of first-class containers 494
- client code 376
- client-code programmer 62
- client of a class 707, 717
- client of an object 48
- <limits> header 141
- clog (standard error buffered) 418, 419, 445
- close member function of ofstream 450
- <cmath> header 109, 140
- code 5
- CodeLite 7
- coin tossing 141
- collaboration 714, 715, 717
- collaboration diagram in the UML 691, 716
- collaboration in the UML 714
- colon (:) 155, 286, 675
- column 211
- column headings 189
- column subscript 211
- comma operator (,) 104, 181
- comma-separated list
 - of parameters 136
 - 25, 35, 104, 230
- command-line argument 240
- comma-separated list of base classes 675
- comment 20
- CommissionEmployee class header 399
- CommissionEmployee class implementation file 400
- CommissionEmployee class represents an employee paid a percentage of gross sales 347
- CommissionEmployee class test program 350
- CommissionEmployee class uses member functions to manipulate its private data 365
- Common Programming Errors overview xxiii
- communication diagram in the UML 691, 716, 717
- commutative 336
- commutative operation 336
- comparator function object 501, 505
- comparator function object less 501, 512
- compare iterators 484
- compare member function of class string 597
- comparing
 - strings 639, 642
- comparing blocks of memory 655
- comparing strings 595
- compilation error 95
- compilation unit 667
- compile 7
- compiler 21, 90
- compiling
 - multiple-source-file program 63
- comp1 operator keyword 669
- complement operator (~) 621
- complex conditions 123
- component 3
- composition 264, 283, 287, 343, 346, 695, 696, 721
- compound interest 108
 - calculation with for 108
- compound statement 34
- computing the sum of the elements of an array 193
- concatenate 595
 - stream insertion operations 28
 - strings 641
- concrete class 390
- concrete derived class 395
- condition 32, 73, 75, 111, 122
- conditional compilation 791, 794
- conditional execution of preprocessing directives 791
- conditional expression 75
- conditional operator (?:) 75
- conditional preprocessing directives 794
- conditionally compiled output statement 795
- confusing equality (==) and assignment (=) operators 32, 127
- conserving memory 153
- consistent state 65
- const 281, 313, 792
- const keyword 163
- const member function 40, 281
- const member function on a const object 282
- const member function on a non-const object 282
- const object 192, 282
- const object must be initialized 192
- const objects and const member functions 282
- const qualifier 191, 240, 661
- const qualifier before type specifier in parameter declaration 166
- const variables must be initialized 192
- const version of operator[] 335
- const with function parameters 240
- const_cast
 - cast away const-ness 662
 - const_cast demonstration 662
 - const_cast operator 661, 662, 663
 - const_iterator 479, 480, 481, 483, 484, 503, 505, 608
 - const_pointer 479
 - const_reference 479
 - const_reverse_iterator 479, 480, 484, 490, 608
 - constant integral expression 119
 - constant pointer
 - to an integer constant 243
 - to constant data 241, 243, 244
 - to nonconstant data 241, 243
 - constant reference 334
 - constant reference parameter 166
 - constant variable 191
 - "const-ness" 663
 - constructed inside out 288
 - constructor 50
 - cannot be virtual 389
 - cannot specify a return type 50
 - conversion 337, 339
 - copy 332
 - default 53
 - default arguments 269
 - defining 52
 - explicit 339
 - function prototype 59
 - in a UML class diagram 54
 - inherit 370
 - inherit from base class 370
 - naming 52
 - parameter list 52
 - single argument 339, 340
 - constructors and destructors called automatically 272
 - container 140, 475, 476
 - container adapter 476, 477, 483, 509
 - priority_queue 512
 - queue 511
 - stack 509

- container adapter functions
 - pop 509
 - push 509
- container class 265, 328, 476
- containers
 - begin function 479
 - cbegin function 479
 - cend function 479
 - clear function 479
 - crbegin function 479
 - crend function 479
 - empty function 478
 - end function 479
 - erase function 479
 - insert function 478
 - max_size function 478
 - rbegin function 479
 - rend function 479
 - size function 478
 - swap function 478
- continue statement 121
- continue statement terminating a single iteration of a for statement 122
- control characters 637
- control statement 70, 73, 74
 - nesting 73
 - stacking 73
- control statements 73
 - do...while 110, 111
 - do...while repetition statement 72
 - for 102, 104
 - for repetition statement 72
 - if 32
 - if single-selection statement 71
 - if...else double-selection statement 72
 - nested if...else 77
 - nesting 92
 - repetition statement 73
 - selection statement 73
 - sequence statement 73
 - switch 112, 119
 - while 102, 110
 - while repetition statement 72
- control variable 102
- control-variable name 104
- controlling expression 117
- converge on the base case 182
- conversion constructor 337, 339
- conversion operator 337
 - explicit 340
- conversions among fundamental types
 - by cast 337
- convert a binary number to decimal 786
- convert a hexadecimal number to decimal 786
- convert among user-defined types and built-in types 337
- convert an octal number to decimal 786
- convert between types 337
- convert lowercase letters 140
- convert strings to floating-point types 612
- convert strings to integral types 612
- converting strings to C-style strings and character arrays 606
- copy algorithm 492, 558
- copy assignment 279
- copy constructor 280, 287, 327, 332, 334, 370, 478, 480

- copy member function of class string
 - 461, 607
- copy of the argument 241
- copy_backward algorithm 539, 558
- copy_if algorithm 558
- copy_n algorithm 541, 558
- copy-and-paste approach 356
- copying strings 640
- correct number of arguments 136
- correct order of arguments 136
- correctly initializing and using a constant variable 191
- cos function 131
- cosine 131
- count algorithm 529, 532, 558
- count function of associative container 503
- count_if algorithm 529, 532, 558
- counter 154
- counter-controlled repetition 79, 88, 92, 101, 182
- counter-controlled repetition with the for statement 103
- counter variable 82
- counting loop 102
- cout (<<) (the standard output stream) 418, 419, 445
- cout (standard output stream) 9
- cout (the standard output stream) 21, 24, 27
- cout.put 421
- cout.write 426
- plusplus predefined symbolic constant 797
- .cpp extension 6
- CPU (central processing unit) 8
- craps simulation 146, 147, 150
- crbegin member function of containers 479
- crbegin member function of vector 490
- create an object (instance) 40
- create your own data types 28
- CreateAndDestroy class
 - definition 273
 - member-function definitions 274
- creating a random access file 457
- Creating a random-access file with 100 blank records sequentially 461
- Creating a sequential file 447
- creating an association 508
- Credit inquiry program 453
- credit processing program 458
- crend member function of containers 479
- crend member function of vector 490
- <cstdio> header 141
- <csdtlib> header 574
- <cstdlib> header 140, 141, 646
- <cstring> header 140, 640
- <ctime> header 140, 146
- <Ctrl>-d 116, 429, 449
- Ctrl key 116
- <Ctrl>-z 116, 429, 449
- <csdtlib> header 142
- current position in a stream 452
- cursor 22
- .cxx extension 6

D

- dangerous pointer manipulation 407
- dangling-else problem 77
- dangling pointer 333
- dangling reference 167
- data hiding 47, 49
- data member 5, 45, 46, 726
- data member function of class string 607
- data members 38
- data persistence 445
- data structures 475
- data types
 - bool 74
 - char 116, 139
 - double 85, 108
 - float 85, 138
 - int 25
 - long 120
 - long double 138
 - long int 120, 139
 - long long 120, 138
 - long long int 138
 - short 120
 - short int 120
 - unsigned 145
 - unsigned char 139
 - unsigned int 139, 145
 - unsigned long 138
 - unsigned long int 138
 - unsigned long long 138
 - unsigned long long int 138
 - unsigned short 139
 - unsigned short int 139
- data types in the UML 44
- Date class 283, 316
- Date class definition 283
- Date class definition with overloaded increment operators 316
- Date class member function definitions 284
- Date class member-function and friend-function definitions 317
- Date class test program 319
- __DATE__ predefined symbolic constant 797
- date source file is compiled 797
- deallocate 321
- deallocate memory 321, 575
- debugger 792
- debugging 144
- debugging aid 795
- debugging tool 797
- dec stream manipulator 427, 432, 436
- decimal (base 10) number system 649, 650, 781
- decimal (base-10) number system 432
- decimal numbers 436, 634
- decimal point 85, 90, 91, 109, 421, 432
- decision 74
- decision in the UML 705
- decision symbol 74
- declaration 25
- declaration of a function 59
- declaring a static member function
 - const 301
- decrement
 - a pointer 247
- decrement a control variable 101

- decrement operator (`--`) 96
 - decrement operators 315
 - default access mode for class is `private` 47
 - default argument 167, 266
 - default arguments with constructors 266
 - `default` case 117, 118, 144
 - default constructor 51, 53, 267, 288, 316, 327, 332, 370, 478
 - provided by the compiler 53
 - provided by the programmer 53
 - default copy constructor 287
 - default delimiter 425
 - default memberwise assignment 279
 - default memberwise copy 332
 - default precision 90
 - default to decimal 436
 - default to `public` access 616
 - default type argument for a type parameter 589
 - default type arguments for function template type parameters 589
 - `default_random_engine` 151
 - `#define` 794, 796
 - define a constructor 52
 - define a member function of a class 38
 - Define class `GradeBook` with a member function `displayMessage`, create a `GradeBook` object, and call its `displayMessage` function 38
 - Define class `GradeBook` with a member function that takes a parameter, create a `GradeBook` object and call its `displayMessage` function 42
 - `#define NDEBUG` 797
 - `#define PI 3.14159` 792
 - `#define` preprocessing directive 792
 - `#define` preprocessor directive 259
 - defining occurrence 9
 - definition 101
 - Deitel Buzz Online* newsletter 18
 - delegating constructor 271
 - `delete` 333, 575, 577
 - `delete []` (dynamic array deallocation) 323
 - delete a record from a file 472
 - `delete` operator 321, 389
 - deleting dynamically allocated memory 333
 - delimiter 254, 644
 - delimiter (with default value `'\n'`) 423
 - delimiting characters 644
 - `Deposit` class (ATM case study) 693, 696, 699, 700, 708, 715, 716, 724, 727, 733, 734
 - `DepositSlot` class (ATM case study) 693, 695, 699, 708, 716, 728
 - `<deque>` header 140, 499
 - `deque` class template 485, 498
 - `push_front` function 499
 - `shrink_to_fit` member function 490
 - dereference
 - a `const` iterator 482
 - a null pointer 232
 - a pointer 232, 235, 241
 - an iterator 480, 481, 484
 - an iterator positioned outside its container 490
 - dereferencing operator (`*`) 232
 - derive one class from another 264
 - derived class 343, 345, 372, 733, 734
 - indirect 401
 - derived-class `catch` 578
 - descriptive words and phrases 699, 701
 - deserialized object 473
 - design process 5, 683, 689, 708, 713
 - design specification 689
 - destructor 272, 357, 478
 - called in reverse order of constructors 272
 - destructor in a derived class 370
 - destructors called in reverse order 370
 - Dev C++ 7
 - diagnostics that aid program debugging 141
 - diamond inheritance 677
 - diamond symbol 71, 74
 - dice game 146
 - die rolling
 - using an array instead of `switch` 195
 - `difference_type` 480
 - digit 26, 253, 781
 - direct base class 345
 - directly reference a value 229
 - disk 8, 9
 - disk drive 417
 - disk I/O completion 570
 - disk space 449, 573, 574
 - displacement 410
 - display screen 417, 419
 - divide by zero 9
 - `DivideByZeroException` 566
 - `divides` function object 554
 - division 29
 - `do...while` repetition statement 72, 110, 111
 - dollar amount 109
 - `dot (.)` operator 40
 - `dot` operator (`.`) 265, 293, 384, 576
 - dotted line 71
 - `double` 26
 - `double` data type 85, 108, 138
 - double-ended queue 498
 - double-precision floating-point number 89
 - double quote 22, 23
 - double-selection statement 72
 - double-word boundary 617
 - “doubly initializing” member objects 288
 - doubly linked list 494, 477
 - downcasting 383
 - driver program 55
 - dummy value 85
 - duplicate keys 500, 505
 - DVD 445
 - dynamic binding 384, 406, 407, 410
 - dynamic casting 411
 - dynamic data structure 229
 - dynamic memory 575
 - dynamic memory management 321
 - dynamic storage duration 153
 - `dynamic_cast` 413, 578
 - dynamically allocate array of integers 328
 - dynamically allocated memory 279, 280, 333, 389, 575
 - allocate and deallocate storage 272
 - dynamically allocated storage 332
 - dynamically determine function to execute 383
- ## E
- Eclipse 7
 - Eclipse Foundation 15
 - edit 6
 - edit a program 6
 - editor 6
 - element of an array 186
 - elided UML diagram 694
 - `#elif` 795
 - emacs 6
 - embedded parentheses 29
 - embedded system 15
 - `Employee` class 283
 - definition showing composition 285
 - definition with a static data member to track the number of `Employee` objects in memory 299
 - header 394
 - implementation file 395
 - member-function definitions 286, 299
 - empty member function
 - of containers 478
 - of `priority_queue` 513
 - of `queue` 511
 - of sequence container 493
 - of `stack` 509
 - of `string` 307, 601
 - empty parentheses 40, 41, 43
 - empty statement 78
 - empty string 48, 601
 - encapsulate 47
 - encapsulation 5, 49, 262, 278, 288
 - `end` function 240
 - end line 28
 - `end` member function of class `string` 608
 - `end` member function of containers 479
 - `end` member function of first-class container 480
 - `end` of a sequence 536
 - `end` of a stream 452
 - “end of data entry” 85
 - end-of-file 116, 117, 254, 442
 - `#endif` preprocessing directive 795
 - `#endif` preprocessor directive 259
 - `endl` 28, 90
 - end-of-file 449
 - end-of-file indicator 449
 - end-of-file key combination 449
 - end-of-file marker 445
 - `Enter` key 27
 - enter key 117, 118
 - `enum`
 - scoped 150
 - specifying underlying integral type 150
 - unscoped 150
 - `enum class` 150
 - `enum` keyword 149
 - `enum struct` 150
 - enumeration 149, 792
 - enumeration constant 149, 794
 - EOF 116, 422, 425, 634

eof member function 422, 442
 eofbit of stream 442
 equal algorithm 523, 558
 equal to 32
 equal_range algorithm 546, 548, 558
 equal_range function of associative container 503
 equal_to function object 554
 equality and relational operators 33
 equality operator (==) 324, 480
 equality operators 32, 33
 equality operators (== and !=) 74, 123
 equation of straight line 30
 erase 603
 erase member function of class string 603
 erase member function of containers 479
 erase member function of first-class containers 493
 e-reader device 16
 #error preprocessing directive 795
 error
 off-by-one 104
 error bits 425
 error detected in a constructor 571
 error state of a stream 422, 440, 441
 escape character 22
 escape early from a loop 121
 escape sequence 22, 24
 escape sequences
 \ ' (single-quote character) 23
 \ " (double-quote character) 23
 \\ (backslash character) 23
 \a (alert) 23
 \n (newline) 23
 \r (carriage return) 23
 \t (tab) 23, 118
 event 703
 <exception> header 140
 exception 225, 561
 handler 225
 handling 221
 parameter 226
 exception class 562, 578
 what virtual function 562
 exception classes derived from common base class 572
 exception handling 140, 561
 out_of_range exception class 226
 what member function of an exception object 226
 <exception> header 562, 578
 exception object 566
 exception parameter 564
 exceptional condition 118
 Exceptions 226
 bad_alloc 572
 bad_cast 578
 bad_typeid 578
 length_error 578
 logic_error 578
 out_of_range 226, 578
 overflow_error 578
 underflow_error 579
 executable image 8
 executable program 7
 execute a program 6, 8
 execution-time error 9

execution-time overhead 407
 exit 449
 exit function 23
 exit function 272, 273, 574
 EXIT_FAILURE 449
 EXIT_SUCCESS 449
 exp function 131
 expand a macro 793
 explicit constructor 339
 explicit conversion 90
 explicit keyword 52, 339
 conversion operators 340
 exponential “explosion” of calls 182
 exponential complexity 182
 exponential function 131
 exponentiation 31, 108
 expression 74, 75, 90, 104
 extensibility 376
 extensibility of C++ 314
 extensible language 179
 extensible markup language (XML) 473
 extensible programming language 40
 extern keyword 154
 extern storage-class specifier 152

F

fabs function 131
 Facebook 15
 factorial 176, 177, 179
 fail member function 442
 failbit 449
 failbit of stream 422, 426, 442
 false 32
 false 74, 75, 182, 438
 fatal logic error 32
 fatal runtime error 9
 fault-tolerant programs 225, 561
 Fibonacci series 179, 182
 field width 110, 189, 426, 429
 fields larger than values being printed 435
 FIFO 477, 498
 FIFO (first-in, first-out) 511
 file 445, 451
 file of *n* bytes 445
 file open mode 447, 450
 file open modes
 ios::app 447
 ios::ate 448
 ios::binary 448, 461, 464
 ios::in 448, 450
 ios::out 447
 ios::trunc 448
 __FILE__ predefined symbolic constant 796
 file processing 417, 420
 file scope 155, 264
 filename 447, 450
 filename extensions 6
 file-position pointer 452, 464, 472
 file-processing classes 420
 fill algorithm 520, 521, 558
 fill character 261, 426, 429, 434, 435
 fill member function 433, 435
 fill member function of basic_ios 442
 fill_n algorithm 520, 521, 558
 final
 class 389
 member function 389
 final state 71
 final state in the UML 705
 final value of a control variable 101, 105
 find algorithm 533, 535, 558
 find function of associative container 503
 find member function of class string 601, 603
 find_end algorithm 558
 find_first_not_of member function of class string 603
 find_first_of algorithm 558
 find_first_of member function of class string 603
 find_if algorithm 533, 536, 558
 find_if_not algorithm 533, 537, 558
 find_last_of member function of class string 603
 finding strings and characters in a string 601
 first data member of pair 503
 first-in, first-out (FIFO) data structure 511
 first-class container 477, 479, 480, 483, 489, 494
 begin member function 480
 clear function 494
 end member function 480
 erase function 493
 first-in, first-out (FIFO) 477, 498
 fixed notation 421, 432, 437
 fixed-point format 91
 fixed-point value 110
 fixed stream manipulator 91, 432, 433, 437
 fixed-size data structure 238
 flag value 85
 flags member function of ios_base 439
 flash drive 445
 float data type 85, 138
 floating point 432, 437
 floating-point arithmetic 304
 floating-point division 90
 floating-point literal 89
 double by default 89
 floating-point number 85, 90
 double data type 85
 double precision 89
 float data type 85
 single precision 89
 floating-point size limits 141
 floating-point number in scientific format 437
 floor function 131
 flow of control 78, 88
 flow of control in the if...else statement 75
 flow of control of a virtual function call 408
 flush buffer 443
 flush output buffer 28
 flushing stream 426
 fmod function 131
 fmtflags data type 439
 for repetition statement 72, 102, 104
 for repetition statement examples 105
 for_each algorithm 529, 533, 558
 force a decimal point 421

forcing a plus sign 434
 form feed ('\f') 634, 637
 formal parameter 136
 formal type parameter 173
 format error 442
 format of floating-point numbers in scientific format 437
 format state 426, 439
 format-state stream manipulators 432
 formatted data file processing 445
 formatted I/O 417
 formatted input/output 456
 formatted text 456
 forward declaration 730
 forward iterator 482, 518, 521, 529, 536, 538, 540
 forward iterator operations 484
 forward iterators 494
 <forward_list> header 140
 forward_list class template 476, 485, 494
 splice_after member function 498
 fractional parts 90
 fragile software 364
 free function (global function) 262
 free store 321
 friend function 289, 346
 friend of a derived class 675
 friends are not member functions 291
 Friends can access private members of class 289
 friendship granted, not taken 289
 front member function of queue 511
 front member function of sequence containers 492
 front_inserter function template 540
 <fstream> header 140
 fstream 446, 462, 466, 471
 <fstream> header 445
 function 3, 9, 21, 137
 argument 41
 call 136
 call overhead 163
 call stack 159
 declaration 137
 definition 136, 156
 empty parentheses 40, 41, 43
 header 40
 local variable 44
 multiple parameters 44
 name 154
 overloading 170
 parameter 41, 43
 parameter list 43
 prototype 58, 136, 137, 156, 165, 235
 return a result 47
 signature 137, 171
 that takes no arguments 162
 trailing return type 175
 function body 40
 function call 41
 function call operator () 340, 410
 function call stack 242
 function object 501, 505, 518, 553
 binary 553
 divides 554
 equal_to 554

function object (cont.)
 greater 554
 greater_equal 554
 less 554
 less_equal 554
 logical_end 554
 logical_not 554
 logical_or 554
 minus 554
 modulus 554
 multiplies 554
 negate 554
 not_equal_to 554
 plus 554
 predefined in the STL 553
 function object less< int > 501
 function object less< T > 505, 512
 function overhead 793
 function overloading 416
 function pointer 407, 410, 518, 553
 function prototype 58, 109, 289, 792
 parameter names optional 59
 function prototype scope 155, 156
 function scope 155, 155
 function template 173, 582, 589
 function template specialization 173
 <functional> header 141, 553
 functional structure of a program 22
 functions 3
 functions for manipulating data in the standard library containers 141
 functions with empty parameter lists 162
 function-template specialization 582
 fundamental type 26
 fundamental types
 unsigned int 82

G
 game of chance 146
 game of craps 147
 game playing 141
 gcount function of istream 426
 general class average problem 85
 general utilities library <cstdlib> 797
 generalities 376
 generalization in the UML 733
 generalized numeric operations 557
 general-utilities library <cstdlib> 646
 generate algorithm 520, 521, 558
 generate_n algorithm 520, 521, 558
 generating values to be placed into elements of an array 191
 generator function 520
 generic algorithms 519
 generic programming 582
 get a value 49
 get and set functions 48
 get member function 422, 423
 get pointer 452
 getline function for use with class string 593
 getline function of cin 424
 getline function of the string header 43, 48
 gets the value of 32
 global 60
 global function 130
 global identifier 665

global namespace 667
 global namespace scope 155, 156, 272, 298
 global object constructors 272
 global scope 272, 274, 667
 global variable 154, 156, 158, 169, 667
 golden mean 179
 golden ratio 179
 good function of ios_base 442
 Good Programming Practices overview xxiii
 goodbit of stream 442
 goto elimination 70
 goto statement 70
 GradeBook.cpp 80, 86
 GradeBook.h 80, 85
 graph information 194
 Graphical User Interface (GUI) 16
 greater function object 554
 greater_equal function object 554
 greater-than operator 32
 greater-than-or-equal-to operator 32
 guard condition 74
 guard condition in the UML 705
 GUI (Graphical User Interface) 16
 guillemets (« and ») in the UML 54

H
 .h filename extension 54
 half-word 617
 handle on an object 264
 hard disk 445
 hardcopy printer 9
 hardware platform 2
 has-a relationship 343, 695, 283
 header 54, 62, 139, 259, 372, 791
 Headers
 <algorithm> 492, 558
 <cmath> 109
 <deque> 499
 <exception> 562
 <forward_list> 494
 <fstream> 445
 <functional> 553
 <iomanip.h> 90
 <iostream> 21, 116
 <list> 494
 <map> 505, 507
 <memory> 575
 <numeric> 559
 <queue> 511, 512
 <set> 501
 <stack> 509
 <stdexcept> 562, 578
 <string> 43
 <typeinfo> 413
 <unordered_map> 505, 507
 <unordered_set> 501, 504
 <vector> 221
 how they are located 57
 name enclosed in angle brackets (< >) 57
 name enclosed in quotes (" ") 57
 heap 321, 512, 548, 551
 heapsort sorting algorithm 548
 helper function 265
 hex stream manipulator 427, 432, 436

hexadecimal 650
 integer 232
 hexadecimal (base-16) number 421, 427, 432, 436, 649, 781
 hexadecimal notation 421
 hexadecimal number system 634
 hide implementation details 288
 hide names in outer scopes 155
 hierarchy of exception classes 578
 hierarchy of shapes 390
 "highest" type 138
 high-level I/O 417
 horizontal tab ('\t') 23, 634, 637
 host object 283

I

IDE (integrated development environment) 6
 identifier 26, 72, 156
 identifiers for variable names 152
 IEC (International Electrotechnical Commission) 2
 #if 795
 #if preprocessing directive 795
 if single-selection statement 71, 74
 if statement 32, 74
 if statement activity diagram 74
 if...else double-selection statement 72, 74, 75
 if...else statement activity diagram 75
 #ifdef preprocessing directive 795
 #ifndef preprocessor directive 259
 #ifndef preprocessing directive 795
 ifstream 446, 450, 451, 464
 ifstream constructor function 450
 ignore 312
 ignore function of istream 425
 implementation inheritance 393
 implementation of a member function changes 270
 implementation phase 737
 implementation process 708, 726
 implicit conversion 90, 338, 339
 via conversion constructors 339
 implicit first argument 291
 implicit handle 264
 implicit, user-defined conversions 338
 implicitly virtual 384
 improper implicit conversion 338
 in-class initializers 260
 in-memory formatting 609
 in-memory I/O 609
 in-class initializer (C++11) 120
 #include 791, 791
 #include "filename" 791
 include guard 257, 259
 #include <iomanip> 90
 #include <iostream> 21
 #include preprocessing directive 791
 #include preprocessor directive 137
 includes algorithm 543, 544, 559
 including a header multiple times 259
 increment
 a pointer 247
 increment a control variable 101, 104, 105
 increment an iterator 484
 increment operator 315

increment operator (++) 96
 indentation 73, 76
 independent software vendor (ISV) 3
 index 186
 indexed access 498
 indirect base class 345
 indirect derived class 401
 indirection 229, 407
 indirection operator (*) 232, 234
 indirectly reference a value 229
 inequality 668
 inequality operator (!=) 324
 inequality operator keywords 668
 infinite loop 89, 104, 176
 information hiding 5
 inherit constructors 370
 inherit constructors from base class 370
 inherit interface 390
 inherit members of an existing class 343
 Inheritance
 hierarchy for university `CommunityMembers` 344
 inheritance 5, 258, 264, 343, 345, 732, 733, 736, 737
 examples 344
 hierarchy 384
 implementation vs. interface inheritance 393
 multiple 672
 relationships of the I/O-related classes 420, 446
 virtual base class 679
 initial state 71
 initial state in the UML 703, 705
 initial value of a control variable 101, 103
 initial value of an attribute 702
 initialize a pointer 230
 initializer 189
 initializer list 189, 253
 initializer_list 552
 initializer_list class template 336
 initializing an array's elements to zeros and printing the array 189
 initializing multidimensional arrays 212
 initializing the elements of an array with a declaration 190
 inline 164, 335, 794
 inline function 163
 inline function 792, 793
 inline function to calculate the volume of a cube 164
 inline keyword 163
 inner block 155
 inner_product algorithm 559
 innermost pair of parentheses 29
 inplace_merge algorithm 542, 559
 input a line of text 424
 Input and output stream iterators 481
 input from string in memory 140
 input iterator 482, 484, 523, 526, 529, 532, 540, 545
 input line of text into an array 254
 input/output library functions 141
 input/output of objects 473
 input/output operations 71
 input/output stream header <iostream> 21
 input sequence 481
 input stream 422, 423

input stream iterator 481
 input stream object (cin) 27
 inputting from strings in memory 609
 insert function of associative container 503, 507
 insert member function of class string 605
 insert member function of containers 478
 insert member function of sequence container 493
 inserter function template 540
 insertion at back of vector 486
 instance 4
 instance of a class 46
 instant access processing 466
 instant-access application 456
 instruction 8
 int 21, 26, 138
 int & 165
 int operands promoted to double 90
 integer 21, 25
 integer arithmetic 304
 Integer class definition 575
 integer division 28, 90
 integer promotion 90
 integers prefixed with 0 (octal) 436
 integers prefixed with 0x or 0X (hexadecimal) 436
 integral constant expression 112
 integral size limits 141
 integrated development environment (IDE) 6
 interaction diagram in the UML 716
 interactions among objects 714, 717
 interest rate 108
 interface 58
 Interface Builder 16
 interface inheritance 393
 interface of a class 58
 internal spacing 434
 internal stream manipulator 432, 434
 International Electrotechnical Commission (IEC) 2
 International Standards Organization (ISO) 2
 invalid_argument class 578
 invalid_argument exception 493
 invalid_argument exception class 260
 invoking a non-const member function on a const object 281
 <iomanip> header 140, 791, 418, 427
 <iomanip.h> header 90
 iOS 15
 ios_base base class 440
 ios_base class
 precision function 427
 width member function 429
 ios::app file open mode 447
 ios::ate file open mode 448
 ios::beg seek direction 452
 ios::binary file open mode 448, 461, 464
 ios::cur seek direction 452
 ios::end seek direction 452
 ios::in file open mode 448, 450
 ios::out file open mode 447
 ios::trunc file open mode 448

`<iostream>` header 21, 139, 418, 419, 791, 116, 445
iota algorithm 559
iPod Touch 16
is a 675
is-a relationship (inheritance) 343, 372
is_heap algorithm 551, 559
is_heap_until algorithm 551, 559
is_partitioned algorithm 558
is_permutation algorithm 558
is_sorted algorithm 558
is_sorted_until algorithm 558
isalnum 634
isalpha 634
iscntrl 634, 637
isdigit 634, 636
isgraph 634, 637
islower 634, 636
ISO 2
isprint 634, 637
ispunct 634, 637
isspace 634, 637
istream 420
istream class 452, 457, 464, 471, 473, 609
 peek function 425
 seekg function 452
 tellg function 452
istream member function ignore 312
istream_iterator 481
istreamstring 611
istreamstring class 609, 611
isupper 634, 636
isxdigit 634, 634
iter_swap algorithm 537, 538, 558
iterating 82
iteration 182, 184
Iterative factorial solution 183
iterative model 688
iterative solution 176, 184
<iterator> header 141, 540, 542
iterator 198, 475, 607, 608
iterator 479, 480, 481, 484, 503, 608
iterator invalidation 519
iterator operations 484
iterator pointing to first element past the end of container 480
iterator pointing to the first element of the container 480
iterator typedef 483
iterator-category hierarchy 483

J

Jacobson, Ivar 684
Jacopini, G. 70
Java programming language 16
Jobs, Steve 16
justified field 435

K

key 500
keyboard 9, 27, 115, 417, 419, 445
Keypad class (ATM case study) 690, 693, 695, 708, 715, 716, 720, 728
key-value pair 477, 505, 507, 508
keywords 72
 and 668

keywords (cont.)

and_eq 669
auto 213
bitand 669
bitor 669
class 173, 583
compl 669
const 163
enum 149
enum class 150
enum struct 150
explicit 52, 339
extern 154
inline 163
mutable 663
namespace 665, 667
not 668
not_eq 668
or 668
or_eq 669
private 47
public 39
static 154
table of keywords 72
template 583
throw 566
typedef 418
typename 173, 583
void 40
xor 669
xor_eq 669

L

label 155
labels in a switch structure 155
lambda expression 518, 556
lambda function 556
lambda introducer 557
large object 166
last-in, first-out (LIFO) 158
 data structure 477, 509
 order 582, 587
late binding 384
leading 0 436
leading 0x and leading 0X 432, 436
left brace { } 21, 24
left justification 110, 434
left-shift operator (<<) 304, 621, 621, 627, 628
left side of an assignment 128, 187, 276, 328
left stream manipulator 110, 432, 433, 433
left-to-right associativity 98
left value 128
left-shift assignment operator (<<=) 629
left-shift operator (<<) 419
left-to-right associativity 35
left-to-right evaluation 29, 30
legacy C code 792
legacy code 797
length member function of class string 593
length of a string 254
length of a substring 340
length_error exception 493, 578, 599
less function object 554
less_equal function object 554

less< double > 505
less< int > 501, 505
less-than operator 32, 480
less-than-or-equal-to operator 32
lexicographical 597
Lexicographical_compare algorithm 522, 524, 559
lifetime of an object in a UML sequence diagram 719
LIFO (last-in, first-out) 158, 477, 509
 order 582, 587
<limits> header 141
line 30
line number 796
line of communication with a file 448, 450
line of text 424
__LINE__ predefined symbolic constant 796
link 6
linkage 152, 667
linker 7
Linux 15
 shell prompt 9
Linux operating system 15
<list> header 140
list class 485, 494
list functions
 assign 498
 merge 498
 pop_back 498
 pop_front 498
 push_front 497
 remove 498
 sort 497
 splice 497
 swap 498
 unique 498
<list> header 494
list initialization 94
list initializer 227, 271
 dynamically allocated array 322
 vector 490
literal
 floating point 89
live-code approach xxiii
load 6
loader 8, 8
local automatic object 275
local variable 44, 154, 156
<locale> header 141
log function 132
log10 function 132
logarithm 132
logic error 6, 32
logic_error exception 578
logical AND 668
logical AND (&&) 123
logical negation 123, 124
logical NOT (!) 123, 124, 668
logical operator keywords 668
logical operators 123
logical OR (||) 123, 124, 626, 668
logical_and function object 554
logical_not function object 554
logical_or function object 554
long data type 120
long double data type 138
long int 177

long int data type 120, 139
 long long data type 120, 138
 long long int data type 138
 loop 72, 79
 loop-continuation condition 72, 101, 102, 104, 105, 110, 111
 loop counter 101
 loop-continuation condition fails 182
 looping statement 72
 loss of data 442
 lower_bound algorithm 548, 558
 lower_bound function of associative container 503
 lowercase letter 634, 636
 lowercase letters 26, 140
 "lowest type" 138
 low-level I/O capabilities 417
 lvalue ("left value") 128, 167, 187, 231, 232, 276, 328, 335, 500
 lvalues as rvalues 128

M

m-by-n array 211
 Mac OS X 15, 16
 machine code 110
 machine dependent 247
 machine language 153
 Macintosh 16
 macro 139
 macro argument 793
 macro definition 796
 macro expansion 793
 macro-identifier 792
 macros 791
 magnitude 434
 magnitude right justified 432
 main 21, 24
 "make your point" 146
 make_heap algorithm 550, 559
 make_pair 507
 mandatory function prototypes 137
 mangled function name 171
 manipulating individual characters 633
 manipulator 109
 manipulators 446
 many-to-one relationship 697
 <map> header 140, 505, 507
 mapped values 500
 mask 623
 "masked off" 623
 matching catch block 564
 math library 140
 math library functions 109, 131

- ceil 131
- cos 131
- exp 131
- fabs 131
- floor 131
- fmod 131
- log 132
- log10 132
- pow 132
- sin 132
- sqrt 132
- tan 132

 mathematical algorithms 529
 mathematical algorithms of the Standard Library 529

max algorithm 552, 559
 max_element algorithm 529, 532, 559
 max_size member function of a string 601
 max_size member function of containers 478
 maximum length 132
 maximum length of a string 601
 maximum size of a string 599
 mean 30
 member function 4, 38, 39, 726

- argument 41
- implementation in a separate source-code file 59
- parameter 41

 member function automatically inlined 261
 member function call 5
 member function calls for const objects 281
 member function calls often concise 262
 member function defined in a class definition 261
 member function definitions of class Integer 576
 member functions that take no arguments 262
 member-initializer list 52, 283, 286, 675
 member object

- default constructor 288
- initializer 287

 member selection operator (.) 265, 293, 384, 576
 memberwise assignment 279, 308
 memberwise copy 332
 memchr 656, 658
 memcmp 656, 657
 memcpy 655, 656
 memmove 656, 657
 <memory> header 140
 memory 153
 memory address 229
 memory consumption 407
 memory functions of the string-handling library 655
 memory handling

- function memchr 658
- function memcmp 658
- function memcpy 656
- function memmove 657
- function memset 659

 <memory> header 575
 memory leak 322, 476, 575, 577, 607

- prevent 577

 memory-access violation 476
 memset 656, 659
 merge algorithm 538, 540, 558
 merge in the UML 705
 merge member function of list 498
 merge symbol 79
 message in the UML 714, 716, 717, 720
 message passing in the UML 719
 Microsoft

- Visual C++ 6

 Microsoft Visual C++ 669
 Microsoft Windows 116
 min algorithm 552, 559
 min_element algorithm 529, 532, 559
 minmax algorithm 552, 559

minmax_element algorithm 529, 532, 553, 559
 minus function object 554
 minus sign (-) indicating private visibility in the UML 726
 minus sign, - (UML) 50
 mismatch algorithm 522, 524, 558
 mission-critical computing 565
 mixed-type expression 138
 model of a software system 694, 702, 735
 modifiable lvalue 308, 328, 335
 modify a constant pointer 243
 modify address stored in pointer variable 243
 modulus function object 554
 modulus operator (%) 28, 29, 142, 146
 monetary formats 141
 most derived class 681
 move algorithm 540, 558
 move assignment operator 334, 480
 move constructor 334, 370, 478, 480
 move semantics 334, 480
 move_backward algorithm 540, 558
 Mozilla Foundation 15
 multidimensional array 211
 multimap associative container 505
 multiple 28
 multiple inheritance 345, 418, 672, 673, 674, 675, 677
 multiple inheritance demonstration 673
 multiple parameters to a function 44
 multiple-selection statement 72, 112
 multiple-source-file program

- compilation and linking process 62

 multiple-statement body 34
 multiplication 28, 29
 multiplicative operators (*, /, %) 90
 multiplicity 694
 multiplies function object 554
 mutable

- data member 663, 663, 664
- demonstration 664
- keyword 152, 663

 mutable data member 664
 mutating sequence algorithms 558
 mutator 49

N

name decoration 171
 name function of class type_info 413
 name handle 264

- on an object 264

 name mangling 171
 name mangling to enable type-safe linkage 172
 name of a control variable 101
 name of a source file 796
 name of a user-defined class 39
 name of a variable 152
 name of an array 187
 named constant 191
 namespace 22

- alias 668
- global 667
- nested 667
- qualifier 668
- unnamed 667

- namespace 665
 - namespace alias 668
 - namespace keyword 665, 667
 - namespace member 665
 - namespace scope 155
 - namespaces 665
 - naming conflict 291, 665
 - narrowing conversion 95
 - natural logarithm 132
 - navigability arrow in the UML 726
 - NDEBUG 797
 - near container 477
 - negate function object 554
 - nested blocks 155
 - nested control statement 92
 - nested for statement 194, 214, 219
 - nested if...else statement 76, 77
 - nested message in the UML 718
 - nested namespace 667
 - nested namespace 667
 - nested parentheses 29
 - NetBeans 7
 - network connection 417
 - network message arrival 570
 - new 332
 - new calls the constructor 321
 - new failure handler 574
 - <new> header 572
 - new operator 321
 - new returning 0 on failure 573
 - new stream manipulators 430
 - new throwing bad_alloc on failure 572, 573
 - newline ('\n') escape sequence 22, 28, 35, 118, 252, 421, 637
 - next_permutation algorithm 559
 - NeXTSTEP operating system 16
 - noboolalpha stream manipulator 438
 - noexcept keyword 571
 - non-const member function 282
 - non-const member function called on a const object 282
 - non-const member function on a non-const object 282
 - nonconstant pointer to constant data 241
 - nonconstant pointer to nonconstant data 241
 - noncontiguous memory layout of a deque 499
 - nondeterministic random numbers 144
 - none_of algorithm 533, 536, 558
 - nonfatal logic error 32
 - nonfatal runtime error 9
 - non-member, friend function 313
 - non-member function to overload an operator 336
 - nonmodifiable function code 264
 - nonmodifiable lvalue 226, 308
 - nonmodifying sequence algorithms 557, 558
 - nonparameterized stream manipulator 90
 - nonrecoverable failures 442
 - non-static member function 291, 301, 337
 - nontype template parameter 589
 - nonzero treated as true 127
 - noshowbase stream manipulator 432, 436
 - noshowpoint stream manipulator 432
 - noshowpos stream manipulator 432, 434
 - noskipws stream manipulator 432
 - NOT (!; logical NOT) 123
 - not equal 32
 - not operator keyword 668
 - not_eq operator keyword 668
 - not_equal_to function object 554
 - note 71
 - nothrow object 573
 - nothrow_t type 573
 - noun phrase in requirements specification 692, 698
 - nouppercase stream manipulator 432, 438
 - nth_element algorithm 558
 - NULL 231
 - null character ('\0') 253, 254, 426, 640, 645
 - null pointer (0) 230, 232, 449, 639
 - null statement 78
 - null terminated 606
 - null-terminated string 255, 421
 - nullptr constant 230
 - number of arguments 136
 - number of elements in an array 245
 - <numeri> 532
 - numeric algorithms 553, 559
 - <numeri> header 559
 - numerical data type limits 141
- O**
- object 2, 3
 - object (or instance) 5, 717
 - object code 7, 62
 - object leaves scope 272
 - object of a derived class 377, 380
 - object of a derived class is instantiated 369
 - object-oriented analysis and design (OOAD) 6, 683
 - object-oriented design (OOD) 683, 690, 692, 698, 702, 707, 726
 - object-oriented language 6
 - object-oriented programming (OOP) 2, 6, 16, 258, 343, 2
 - object serialization 473
 - object's vtable pointer 410
 - Objective-C 16
 - objects contain only data 264
 - oct stream manipulator 427, 432, 436
 - octal (base-8) number system 427, 432, 649
 - octal number 421, 436, 634, 650
 - octal number system (base 8) 781
 - off-by-one error 104
 - offset 410
 - offset from the beginning of a file 452
 - offset to a pointer 250
 - ofstream 446, 448, 450, 451, 461, 464, 466
 - constructor 448
 - open function 448
 - one's complement 627, 788
 - one's complement operator (~) 621
 - one-pass algorithm 482
 - ones position 781
 - one-to-many mapping 477
 - one-to-many relationship 505, 697
 - one-to-one mapping 477, 507
 - one-to-one relationship 697
 - OOAD (object-oriented analysis and design) 6, 683
 - OOD (object-oriented design) 683, 690, 692, 698, 702, 707
 - OOP (object-oriented programming) 2, 6, 343
 - open a file for input 448
 - open a file for output 448
 - open a nonexistent file 449
 - open function of ofstream 448
 - Open Handset Alliance 16
 - open source 15, 16
 - opened 445
 - operand 22, 27, 28, 75
 - operating system 15, 16
 - operation (UML) 41
 - operation compartment in a class diagram 708
 - operation in the UML 41, 694, 707, 708, 711, 728, 731, 736
 - operation parameter in the UML 44, 708, 711, 712
 - operator
 - associativity 126
 - overloading 28, 173, 304, 416, 621
 - precedence 29, 98, 126, 629
 - precedence and associativity chart 35
 - operator keywords 669
 - operator keywords 308, 668, 669
 - operator keywords demonstration 669
 - operator overloading
 - decrement operators 315
 - increment operators 315
 - operator void* 452
 - operator void* member function 442
 - operator void* member function of ios 449
 - operator! member function 314, 442, 449
 - operator!= 335
 - operator() overloaded operator 553
 - operator[]
 - const version 335
 - non-const version 335
 - operator+ 308
 - operator++ 315, 321
 - operator++(int) 315
 - operator<< 313, 331
 - operator= 333, 478
 - operator== 334, 523
 - operator>> 312, 331
- operators
- ! (logical NOT operator) 123, 124
 - != (inequality operator) 32
 - .* and -* 670
 - () (parentheses operator) 29
 - * (multiplication operator) 29
 - * (pointer dereference or indirection) 232, 232
 - *= multiplication assignment 96
 - / (division operator) 29
 - /= division assignment 96
 - && (logical AND operator) 123
 - % (modulus operator) 29
 - %= modulus assignment 96
 - + (addition operator) 27, 29
 - += 595
 - += addition assignment 95

operators (cont.)

- < (less-than operator) 32
- << (stream insertion operator) 22, 28
- <= (less-than-or-equal-to operator) 32
- = (assignment operator) 27, 29, 125
- = subtraction assignment 96
- == (equality operator) 32, 125
- > (greater-than operator) 32
- >= (greater-than-or-equal-to operator) 32
- || (logical OR operator) 123, 124
- addition assignment (+=) 95
- address (&) 232
- arithmetic 95
- arrow member selection (->) 265
- assignment 95
- conditional (?) 75
- const_cast 661
- decrement (--) 96, 97
- delete 321
- dot (.) 40
- increment (++) 96
- member selection (.) 265
- multiplicative (*, /, %) 90
- new 321
- parentheses () 90
- postfix decrement 96
- postfix increment 96, 98
- prefix decrement 96
- prefix increment 96, 98
- scope resolution (::) 60
- sizeof 244, 245
- static_cast 90
- ternary 75
- typeid 413
- unary minus (-) 90
- unary plus (+) 90
- unary scope resolution (::) 169
- optimizations on constants 281
- optimizing compiler 110, 154
- OR (||; logical OR) 123
- or operator keyword 668
- or_eq operator keyword 669
- order in which constructors and destructors are called 274
- order in which destructors are called 272
- order in which operators are applied to their operands 180
- order of evaluation 181
- ordered associative containers 476, 500
- original format settings 440
- OS X 16
- ostream 452, 457, 466, 473
- ostream class 418
 - seekp function 452
 - tellp function 452
- ostream_iterator 481
- ostringstream class 609
- other character sets 592
- out-of-range array subscript 570
- out-of-range element 328
- out of scope 158
- out_of_bounds exception 493
- out_of_range class 335
- out_of_range exception 493, 514, 578, 595
- out_of_range exception class 226
- outer block 155

- outer for structure 214
- out-of-bounds array elements 198
- output a floating-point value 432
- output buffering 443
- output data items of built-in type 419
- output format of floating-point numbers 437
- output iterator 482, 484, 521, 529, 542, 545
- output of char * variables 421
- output of characters 420
- output of floating-point values 421
- output of integers 421
- output of standard data types 420
- output of uppercase letters 421
- output sequence 481
- output stream 492
- output to string in memory 140
- outputting to strings in memory 609
- overflow 570
- overflow_error exception 578
- overhead of a function call 793
- overload the addition operator (+) 308
- overload unary operator ! 314
- overloaded [] operator 328
- overloaded << operator 314
- overloaded addition assignment operator (+=) 316
- overloaded assignment (=) operator 327, 333
- overloaded binary operators 309
- overloaded cast operator function 337
- overloaded constructors 371
- overloaded equality operator (==) 327, 334
- overloaded function 171, 589
- overloaded function call operator () 340
- overloaded function definitions 170
- overloaded increment operator 316
- overloaded inequality operator 327, 335
- overloaded operator
 - () 553
- overloaded operator += 320
- overloaded operator[] member function 335
- overloaded postfix increment operator 316, 320
- overloaded prefix increment operator 316, 320
- overloaded stream insertion and stream extraction operators 311
- overloaded stream insertion operator 675
- overloaded subscript operator 328, 335
- overloading 28, 170
 - constructor 271
- overloading + 309
- overloading << and >> 173
- overloading binary operator < 310
- overloading binary operators 309
- overloading function call operator () 340
- overloading operators 173
- overloading postfix increment operator 315, 321
- overloading prefix and postfix decrement operators 315
- overloading prefix and postfix increment operators 315
- overloading resolution 589

- overloading stream insertion and stream extraction operators 310, 316, 320, 327, 331
- overloading template functions 589
- overloading the stream insertion operator 473
- override a function 383
- override keyword 384

P

- pad with specified characters 421
- padding 633
- padding characters 429, 432, 433, 435
- padding in a structure 633
- pair 503
- pair of braces {} 34, 66
- parameter 41, 43
- parameter in the UML 44, 708, 711, 712
- parameter list 43, 52
- parameterized stream manipulator 90, 109, 418, 427, 452
- parameterized type 582
- parentheses operator () 29, 90
- parentheses to force order of evaluation 35
- partial_sort algorithm 558
- partial_sort_copy algorithm 558
- partial_sum algorithm 559
- partition algorithm 558
- partition_copy algorithm 558
- partition_point algorithm 558
- Pascal case 39
- pass-by-reference 164, 229, 235, 237
 - with a pointer parameter used to cube a variable's value 235
 - with pointer parameters 233
 - with reference parameters 165, 233
- pass-by-reference with pointers 166
- pass-by-value 164, 165, 233, 234, 236, 243
 - used to cube a variable's value 234
- passing arguments by value and by reference 165
- passing large objects 166
- passing options to a program 240
- "past the end" iterator 532
- peek function of istream 425
- percent sign (%) (modulus operator) 28
- perform a task 40
- perform an action 22
- performance 3
- PI 792, 793
- plus function object 554
- plus sign 434
- plus sign (+) indicating public visibility in the UML 726
- plus sign, + (UML) 41
- pointer 247
- pointer 479
- pointer arithmetic 247, 248, 250, 489
 - machine dependent 247
- pointer assignment 249
- pointer-based strings 252
- pointer comparison 249
- pointer dereference (*) operator 232
- pointer expression 247, 250
- pointer handle 264
- pointer manipulation 407

- pointer notation 251
 - pointer operators `&` and `*` 232
 - pointer to a function 407
 - pointer to an object 242
 - pointer to `void` (`void *`) 249
 - pointer variable 575
 - pointer/offset notation 250
 - pointer/subscript notation 250
 - pointer-based string 606
 - pointers and array subscripting 249, 250
 - pointers and arrays 249
 - pointers declared `const` 243
 - pointers to dynamically allocated storage 293, 334
 - pointer-to-member operators
 - `.*` 670
 - `->*` 670
 - point-of-sale system 456
 - poll analysis program 196
 - polymorphic exception processing 572
 - polymorphic programming 390, 410
 - polymorphic screen manager 376
 - polymorphically invoking functions in a derived class 678
 - polymorphism 373, 375
 - polymorphism and references 407
 - polynomial 31
 - `pop` 587
 - `pop` function of container adapters 509
 - `pop` member function of
 - `priority_queue` 512
 - `pop` member function of `queue` 511
 - `pop` member function of `stack` 509
 - `pop_back` member function of `list` 498
 - `pop_front` 494, 495, 500, 511
 - `pop_heap` algorithm 551, 559
 - Portability Tips overview xxiv
 - portable 2
 - position number 186
 - positional notation 781
 - positional value 781, 782
 - positional values in the decimal number system 782
 - postdecrement 96, 98
 - postfix decrement operator 96
 - postfix increment operator 96, 98
 - postincrement 96, 320
 - postincrement an iterator 484
 - `pow` function 108, 110, 132
 - power 132
 - precedence 29, 31, 35, 98, 104, 124, 180
 - precedence chart 35
 - precedence not changed by overloading 309
 - precedence of the conditional operator 75
 - precision 90, 421, 426
 - format of a floating-point number 91
 - `precision` function of `ios_base` 427
 - precision of floating-point value 85
 - precision of floating-point numbers 427
 - precision setting 428
 - predecrement 96, 98
 - predefined function objects 553
 - predefined symbolic constants 796
 - predicate function 265, 497, 523, 526, 529, 532, 536, 537, 540, 545, 550
 - prefix decrement operator 96, 97
 - prefix increment operator 96, 98
 - preincrement 96, 320
 - preprocessing directives 7, 21
 - preprocessor 6, 7, 137, 791
 - preprocessor directives
 - `#ifndef` 259
 - `#define` 259
 - `#endif` 259
 - `prev_permutation` algorithm 559
 - prevent memory leak 577
 - preventing headers from being included more than once 259
 - primary memory 8
 - primitive data type promotion 90
 - principal 108
 - principle of least privilege 153, 239, 240, 243, 281, 450, 483
 - print a line of text 20
 - printer 9, 417
 - printing
 - line of text with multiple statements 23
 - multiple lines of text with a single statement 24
 - unsigned integer in bits 622
 - `priority_queue` adapter class 512
 - empty function 513
 - `pop` function 512
 - `push` function 512
 - size function 513
 - top function 513
 - `private`
 - access specifier 47, 726
 - base class 372
 - base-class data cannot be accessed from derived class 358
 - inheritance 345
 - members of a base class 345
 - static data member 298
 - private libraries 7
 - probability 141
 - program development environment 6
 - program execution stack 159
 - program in the general 375
 - program in the specific 375
 - program termination 275, 276
 - programmer-defined function
 - maximum 132
 - promotion 90
 - promotion hierarchy for built-in data types 138
 - promotion rules 138
 - prompt 27, 88
 - prompting message 443
 - proprietary classes 372
 - `protected` 361
 - `protected` access specifier 258
 - `protected` base class 372
 - `protected` base-class data can be accessed from derived class 363
 - `protected` inheritance 345, 372
 - pseudorandom numbers 144
 - `public`
 - keyword 726, 731
 - method 260
 - `public` access specifier 39
 - `public` base class 372
 - `public` inheritance 343, 345
 - `public` keyword 39
 - `public` member of a derived class 346
 - `public` services of a class 58
 - `public static` class member 298
 - `public static` member function 298
 - punctuation mark 644
 - pure specifier 391
 - pure virtual function 391, 407
 - `push` 587
 - `push` function of container adapters 509
 - `push` member function of
 - `priority_queue` 512
 - `push` member function of `queue` 511
 - `push` member function of `stack` 509
 - `push_back` member function of class
 - template vector 227
 - `push_back` member function of `vector` 488
 - `push_front` member function of `deque` 499
 - `push_front` member function of `list` 497
 - `push_heap` algorithm 551, 559
 - put file-position pointer 457, 462
 - `put` member function 421, 422
 - put pointer 452
 - putback function of `istream` 425
- ## Q
- qualified name 369
 - `<queue>` header 140
 - `queue` adapter class 511
 - `back` function 511
 - empty function 511
 - `front` function 511
 - `pop` function 511
 - `push` function 511
 - size function 511
 - `<queue>` header 511, 512
 - quotation marks 22
- ## R
- radians 131
 - raise to a power 132
 - `rand` function 141, 142
 - `RAND_MAX` symbolic constant 141
 - random integers in range 1 to 6 142
 - random number 144
 - `random_shuffle` algorithm 529, 531, 558
 - random-access file 445, 457, 458, 464, 466
 - random-access iterator 482, 483, 498, 501, 518, 524, 531, 536, 550, 551
 - random-access iterator operations 484
 - randomizing 144
 - randomizing the die-rolling program 145
 - range 481, 532
 - range checking 324, 595
 - range-based `for` 595, 608
 - range-based `for` statement 200
 - Rational Software Corporation 690
 - Rational Unified Process™ 690
 - raw data 456
 - raw data processing 445
 - `rbegin` member function of class
 - `string` 608
 - `rbegin` member function of containers 479
 - `rbegin` member function of `vector` 490

- rdstate function of `ios_base` 442
 - read 457, 464
 - read a line of text 43
 - read characters with `getline` 43
 - read data sequentially from a file 450
 - read function of `istream` 425
 - read member function 426
 - read member function of `istream` 457, 471
 - read-only variable 191
 - Reading a random-access file sequentially 464
 - Reading and printing a sequential file 450
 - real number 85
 - record 446, 466
 - record format 459
 - recover from errors 442
 - recursion 175, 182, 183
 - recursion step 176, 179
 - recursive call 176, 179
 - recursive function 175
 - recursive function `factorial` 178
 - recursive solution 184
 - redundant parentheses 31, 123
 - reference 229, 416, 479
 - reference argument 233
 - reference parameter 164, 165, 165
 - reference to a constant 166
 - reference to a `private` data member 276
 - reference to an automatic variable 167
 - reference to an `int` 165
 - referencing array elements 251
 - referencing array elements with the array name and with pointers 251
 - register declaration 154
 - register storage-class specifier 152
 - regular expression 17, 609
 - `reinterpret_cast` operator 249, 458, 461, 464
 - reinventing the wheel 3
 - relational operator 32, 33
 - relational operators `>`, `<`, `>=`, and `<=` 122
 - release dynamically allocated memory 333
 - remainder after integer division 28
 - remove algorithm 526, 558
 - remove member function of `list` 498
 - `remove_copy` algorithm 524, 526, 558
 - `remove_copy_if` algorithm 524, 527, 541, 558
 - `remove_if` algorithm 524, 526, 558
 - `rend` member function of class `string` 608
 - `rend` member function of `containers` 479
 - `rend` member function of `vector` 490
 - repetition
 - counter controlled 79, 88
 - repetition statement 70, 73
 - `do...while` 110, 111
 - for 102, 104
 - `while` 78, 102, 110
 - repetition terminates 78
 - replace 603
 - replace `==` operator with `=` 127
 - `replace` algorithm 527, 529, 558
 - `replace` member function of class `string` 603, 605
 - `replace_copy` algorithm 527, 529, 558
 - `replace_copy_if` algorithm 527, 529, 558
 - `replace_if` algorithm 527, 529, 558
 - for a macro or symbolic constant 792, 794
 - requirements 5, 683, 688
 - requirements document 688, 689
 - requirements gathering 688
 - requirements specification 684
 - reset 514
 - `resize` member function of class `string` 601
 - resource leak 572
 - restore a stream's state to "good" 442
 - resumption model of exception handling 565
 - rethrow an exception 567
 - return a result 137
 - Return* key 27
 - return message in the UML 720
 - `return` statement 23, 47, 137, 176
 - return type 40
 - `void` 40, 48
 - return type in a function header 137
 - return type in the UML 708, 713
 - returning a reference from a function 167
 - returning a reference to a `private` data member 276
 - reusability 582
 - reusable software components 3
 - reuse 4, 54, 264
 - reverse algorithm 538, 541, 558
 - `reverse_copy` algorithm 541, 542, 558
 - `reverse_iterator` 479, 480, 484, 490, 608
 - `rfind` member function of class `string` 603
 - right brace `}` 21, 23, 89
 - right justification 110, 432, 433
 - right operand 22
 - right shift `>>` 621
 - right shift operator `>>` 304
 - right shift with sign extension assignment operator `>>=` 629
 - right stream manipulator 110, 432, 433
 - right-to-left associativity 98
 - right value 128
 - rightmost (trailing) arguments 167
 - right-shift operator `>>` 419, 621, 622, 628
 - right-shifting a signed value is machine dependent 629
 - right-to-left associativity 35
 - Ritchie, Dennis 2
 - robust application 561, 565
 - role in the UML 695
 - role name in the UML 695
 - rolling a die 142
 - rolling a six-sided die 6000 times 143
 - rolling two dice 146, 147
 - `rotate` algorithm 558
 - `rotate_copy` algorithm 558
 - round a floating-point number for display purposes 91
 - rounded rectangle (for representing a state in a UML state diagram) 703
 - rounding numbers 91, 131
 - row subscript 211
 - rows 211
 - RTTI (runtime type information) 411, 414
 - rules of operator precedence 29
 - Rumbaugh, James 684
 - runtime error 9
 - runtime type information (RTTI) 411, 414
 - `runtime_error` class 562, 570, 578
 - `what` function 567
 - rvalue* ("right value") 128, 167, 328
- ## S
- `SalariedEmployee` class header 397
 - `SalariedEmployee` class implementation file 398
 - savings account 108
 - scaling 142
 - scaling factor 142, 146
 - scientific notation 91, 421, 437
 - scientific notation floating-point value 438
 - scientific stream manipulator 432, 437
 - scope 104, 155, 665
 - scope of a symbolic constant or macro 794
 - scope of an identifier 152, 154
 - scope resolution operator `::` 60, 298, 585, 665, 668, 672, 677
 - scoped enum 150
 - scope-resolution operator `::` 150
 - scopes
 - class 155
 - file 155
 - function 155
 - function prototype 155
 - namespace 155
 - scoping example 156
 - screen 9, 21
 - Screen class (ATM case study) 693, 695, 708, 714, 715, 716, 718, 720, 728
 - screen-manager program 376
 - scrutinize data 259
 - search algorithm 558
 - search functions of the string-handling library 651
 - search key 500
 - `search_n` algorithm 558
 - searching 533
 - searching arrays 209
 - searching blocks of memory 655
 - searching strings 639, 646
 - second data member of `pair` 503
 - second-degree polynomial 31
 - secondary storage device 445
 - secondary storage devices
 - CD 445
 - DVD 445
 - flash drive 445
 - hard disk 445
 - tape 445
 - second-degree polynomial 31
 - "secret" implementation details 663
 - security flaws 198
 - seed 146
 - seed function `rand` 144
 - seek direction 452
 - seek get 452

- seek put 452
- seekg function of `istream` 452, 472
- seekp function of `ostream` 452, 462
- select a substring 340
- selection statement 70, 73
- self-assignment 333
- self-assignment 293
- self-documenting 26
- semicolon (;) 22, 34, 78, 791
- semicolon that terminates a structure definition 617
- send a message to an object 5
- sentinel value 85, 89, 116
- separate interface from implementation 58
- sequence 210, 481, 538, 540
- sequence container 476, 483, 485, 493, 497
 - back function 492
 - empty function 493
 - front function 492
 - insert function 493
- sequence diagram in the UML 691, 716
- sequence of messages in the UML 717
- sequence of random numbers 144
- sequence statement 70, 71, 73
- sequence-statement activity diagram 71
- sequential execution 70
- sequential file 445, 446, 447, 450, 456
- serialized object 473
- services of a class 49
- <set> header 140
- set a value 49
- set and get functions 48
- set associative container 504
- set function 288
- <set> header 501, 504
- set_intersection 545
- set_new_handler function 572, 574
- set of recursive calls to method `Fibonacci` 181
- set operations of the Standard Library 543
- set_difference algorithm 543, 545, 559
- set_intersection algorithm 543, 545, 559
- set_new_handler specifying the function to call when new fails 574
- set_symmetric_difference algorithm 543, 545, 559
- set_union algorithm 543, 546, 559
- setbase stream manipulator 427
- setfill stream manipulator 261, 433, 435
- setprecision stream manipulator 90, 109, 427
- setw 189, 312
- setw parameterized stream manipulator 109
- setw stream manipulator 254, 429, 433
- Shape class hierarchy 345
- shell prompt on Linux 9
- shift a range of numbers 142
- shifted, scaled integers 142
- shifted, scaled integers produced by `1 + rand() % 6` 142
- shiftingValue 146
- short-circuit evaluation 124
- short data type 120
- short int data type 120
- showbase stream manipulator 432, 436
- showpoint stream manipulator 91, 432
- showpos stream manipulator 432, 434
- shrink_to_fit member function of classes `vector` and `deque` 490
- shuffle algorithm 558
- shuffling algorithm 619
- side effect 164
- side effect of an expression 154, 164, 181
- sign extension 622
- sign left justified 432
- sign value 85
- signature 137, 171, 315
- signatures of overloaded prefix and postfix increment operators 315
- significant digits 433
- simple condition 122, 124
- sin function 132
- sine 132
- single-argument constructor 339, 340
- single-entry/single-exit control statement 73, 74
- single inheritance 345, 677
- single-line comment 21
- single-precision floating-point number 89
- single quote 23
- single quote (') 252
- single-selection if statement 72, 76
- singly linked list 476, 494
- six-sided die 142
- size function of `string` 461
- size member function of `array` 187
- size member function of class `string` 64, 593
- size member function of containers 478
- size member function of `priority_queue` 513
- size member function of `queue` 511
- size member function of `stack` 509
- size member function of `vector` 224
- size of a `string` 599
- size of a variable 152
- size of an array 244
- size_t 189, 458
- size_t type 244
- size_type 480
- sizeof 464, 651, 794
- sizeof operator 244, 245, 291
 - used to determine standard data type sizes 245
- sizeof operator when applied to an array name returns the number of bytes in the array 245
- skip remainder of `switch` statement 121
- skip remaining code in loop 122
- skipping whitespace 426, 432
- skipws stream manipulator 432
- small circle symbol 71
- smart pointer `xxi`, 18
- smartphone 16
- software engineering 58
 - data hiding 47, 49
 - encapsulation 49
 - reuse 54, 57
 - separate interface from implementation 58
 - set and get functions 48
 - Software Engineering Observations overview `xxiv`
 - software life cycle 688
 - software reuse 3, 343, 582, 672
 - solid circle (for representing an initial state in a UML diagram) in the UML 703, 705
 - solid circle enclosed in an open circle (for representing the end of a UML activity diagram) 705
 - solid circle symbol 71
 - solid diamonds (representing composition) in the UML 695
 - sort algorithm 209, 533, 536, 558
 - sort member function of `list` 497
 - sort_heap algorithm 550, 559
 - sorting 446, 533
 - sorting and related algorithms 557
 - sorting arrays 209
 - sorting order 536, 540
 - sorting strings 141
 - source code 6, 372
 - source-code file 54
 - SourceForge 15
 - spaces for padding 435
 - space-time trade-off 466
 - special character 253
 - special characters 26
 - specialization in the UML 734
 - spiral 179
 - splice member function of `list` 497
 - splice_after member function of class template `forward_list` 498
 - sqrt function of <cmath> header 132
 - square function 139
 - square root 132, 428
 - srand function 144
 - srand(time(0)) 146
 - <sstream> header 140, 609, 609
 - stable_partition algorithm 558
 - stable_sort algorithm 558
 - <stack> header 140
 - stack 158, 582
 - stack adapter class 509
 - empty function 509
 - pop function 509
 - push function 509
 - size function 509
 - top function 509
 - Stack class template 582, 589
 - stack frame 159
 - <stack> header 509
 - stack overflow 159, 176
 - stack unwinding 566, 569, 571
 - Stack<double> 585, 588
 - Stack<int> 588
 - Stack<T> 587
 - standard data type sizes 245
 - standard error stream (`cerr`) 9
 - standard exception classes 578
 - standard input object (`cin`) 27
 - standard input stream (`cin`) 9, 418
 - standard input stream object (`cin`) 445
 - Standard Library
 - class `string` 305
 - container classes 476
 - deque class template 499
 - exception classes 579
 - exception hierarchy 578

- Standard Library (cont.)
 - headers 141, 791
 - list class template 495
 - map class template 508
 - multimap class template 506
 - multiset class template 501
 - priority_queue adapter class 513
 - queue adapter class templates 512
 - set class template 504
 - stack adapter class 509
 - vector class template 487
- standard output object (cout) 22, 418
- standard output stream (cout) 9
- standard output stream object (cout) 445
- standard stream libraries 418
- Standard Template Library 475
- state 690
- state bits 422
- state diagram for the ATM object 703
- state diagram in the UML 703
- state in the UML 690, 705
- state machine diagram in the UML 690, 703
- state of an object 698, 703
- statement 22, 40
- statement spread over several lines 35
- statement terminator (;) 22
- statements
 - break 118, 121
 - continue 121
 - do...while 110, 111
 - for 102, 104
 - if 32
 - return 23
 - switch 112, 119
 - throw 261
 - try 226
 - while 102, 110
- static array initialization 198, 199
- static array initialization and automatic array initialization 199
- static binding 384
- static_cast<int> 116
- static data member 207, 297, 298
- static data member tracking the number of objects of a class 300
- static data members save storage 298
- static keyword 154
- static linkage specifier 667
- static local object 273, 275
- static local variable 156, 158, 198, 521
- static member 298
- static member function 298
- static storage class 152
- static storage duration 153, 154, 155
- static storage-class specifier 152
- static_cast 98, 126
- static_cast (compile-time type-checked cast) 188
- static_cast operator 90
- status bits 442
- std namespace 592
- std::cin 27
- std::cout 21
- std::endl stream manipulator 28
- __STDC__ predefined symbolic constant 797
- <stdexcept> header 140, 562, 578
- StepStone 16
- “sticky” setting 261
- sticky setting 110, 125
- STL 475
- STL algorithms
 - accumulate 553
- STL exception types 493
- stod function 612
- stof function 612
- stoi function 612
- stol function 612
- stold function 612
- stoll function 612
- storage alignment 617
- storage duration 152
 - automatic 153
 - dynamic 153
 - static 153
 - thread 153
- storage unit 633
- storage-class specifiers 152
 - extern 152
 - mutable 152
 - register 152
 - static 152
- storage-unit boundary 633
- stoul function 612
- stoull function 612
- str member function 610
- str member function of class ostream-stream 609
- straight-line form 29, 30
- strcat function of header <cstring> 639, 641
- strchr 651
- strcmp function of header <cstring> 639, 642
- strcpy function of header <cstring> 639, 640
- strcspn 651, 652
- stream base 427
- stream extraction operator 419
- stream extraction operator >> (“get from”) 27, 34, 173, 304, 310, 332, 419, 422, 473
- stream I/O class hierarchy 446
- stream input 419, 422
- stream input/output 21
- stream insertion operator << (“put to”) 22, 23, 28, 173, 304, 310, 332, 419, 420, 449, 675
- stream manipulator 28, 109, 426, 434, 452
- stream manipulators 90
 - boolalpha 125, 438
 - dec 427
 - fixed 91, 437
 - hex 427
 - internal 434
 - left 110, 433
 - noboolalpha 438
 - noshowbase 436
 - noshowpoint 432
 - noshowpos 432, 434
 - nouppercase 432, 438
 - oct 427
 - right 110, 433
 - scientific 437
 - setbase 427
 - setfill 261, 435
- stream manipulators (cont.)
 - setprecision 90, 109, 427
 - setw 109, 254, 429
 - showbase 436
 - showpoint 91, 432
 - showpos 434
 - std::endl (end line) 28
- stream of bytes 417
- stream of characters 22
- stream operation failed 442
- stream output 419
- <string> header 140
- string 477
 - size function 461
- string assignment 593, 594
- string assignment and concatenation 594
- string being tokenized 645
- string class 43, 304, 307, 593
 - at member function 308
 - size member function 64
 - substr member function 66, 307
- string class copy constructor 592
- string class from the Standard Library 140
- string comparison 595
- string concatenation 593
- string constant 253
- string-conversion function 646
 - atof 647
 - atoi 648
 - atol 648
 - strtod 649
 - strtoul 649
 - strtoul 650
- string find member function 601
- string find member functions 601
- <string> header 43, 592, 56
- string insert functions 605
- string insert member function 605
- string length 645
- string literal 22, 253
- string object
 - empty string 48
 - initial value 48
- string of characters 22
- string-search function
 - strchr 652
 - strcspn 652
 - strpbrk 653
 - strrchr 653
 - strspn 654
 - strstr 655
- string stream processing 609
- string::npos 603
- strings as full-fledged objects 252
- strlen function 640, 645
- strncat function 639, 641
- strncmp function 640, 642
- strcpy function 639, 640
- Stroustrup, B. 2
- strpbrk 651, 653
- strrchr 651, 653
- strspn 651, 654
- strstr 651, 654
- strtod 646, 648
- strtok function 640, 644
- strtoul 647, 649, 649
- strtoul 647, 650

struct 616
 structure 616, 792
 structure definition 616, 630
 structure members default to **private**
 access 616
 structure name 616
 structure of a system 702, 703
 structure type 616
 structured programming 70
 student-poll-analysis program 196
 subclass 343
 subclass of a base class 678
 subproblem 176
 subscript 186
 subscript 0 (zero) 186
 subscript operator 500
 subscript operator [] 595
 subscript operator [] used with **strings**
 593
 subscript operator for **map** 507
 subscript out of range 493
 subscript through a **vector** 493
 subscripted name used as an *rvalue* 328
 subscripting 498
 subscripting with a pointer and an offset
 251
substr 598
substr member function of class
 string 66, 598
substr member function of **string** 307
 substring 340
 substring length 340
 substring of a **string** 598
 subtract one pointer from another 247
 subtraction 29
 sum of the elements of an **array** 193
 summing integers with the **for** statement
 107
 superclass 343
 survey 196, 198
 swap algorithm 538, 558
 swap member function of class **string**
 598
 swap member function of **containers** 478
 swap member function of **list** 498
 swap_ranges algorithm 537, 538, 558
 swapping **strings** 598
 swapping two **strings** 598
 switch logic 390
 switch multiple-selection statement
 112, 119
 switch multiple-selection statement ac-
 tivity diagram with **break** statements
 120
 symbol 592
 symbol values 781
 symbolic constant 791, 792, 794, 796
 symbolic constant **NDEBUG** 797
 symbolic constant **PI** 793
 synchronize operation of an **istream** and
 an **ostream** 443
 synchronous call 717
 synchronous error 570
 system 690
 system behavior 690
 system requirements 688
 system structure 690

T

tab 35
 tab escape sequence `\t` 118
 Tab key 22
 tab stop 23
 table of values 211
 tablet computer 16
 tabular format 189
 tails 142
 tan function 132
 tangent 132
 tape 445
 tellg function of **istream** 452
 tellp function of **ostream** 452
 template 792
 default type argument for a type pa-
 rameter 589
 template definition 174
 template function 173
template keyword 173, 583
 template parameter 583
 template parameter list 173
 temporary object 337
 temporary value 90, 139
 terminate a program 574
 terminate normally 449
 terminate successfully 23
 terminating condition 177
 terminating null character 253, 254, 607,
 640, 645
 terminating right brace (}) of a block 155
 termination condition 198
 termination housekeeping 272
 termination model of exception handling
 565
 termination test 182
 ternary conditional operator (?:) 181
 ternary operator 75
test 514
 test characters 140
 test state bits after an I/O operation 422
 text editor 450
 text file 466
 text-printing program 20
 text substitution 793
 this pointer 291, 293, 301, 334
 this pointer used explicitly 291
 this pointer used implicitly and explic-
 itly to access members of an object 292
 thread storage duration 153
 throw an exception 226, 260, 261, 564
 throw exceptions derived from standard
 exceptions 579
 throw exceptions not derived from stan-
 dard exceptions 579
 throw keyword 566
 throw point 565
 throw standard exceptions 579
 tie an input stream to an output stream
 443
 tilde character (~) 272
 Time class containing a constructor with
 default arguments 266
 Time class definition 258
 Time class definition modified to enable
 cascaded member-function calls 294
 Time class member-function definitions
 259

Time class member-function definitions,
 including a constructor that takes ar-
 guments 267
time function 146
 __TIME__ predefined symbolic constant
 797
 time source file is compiled 797
 to_string function 612
 token 640, 644
 tokenizing strings 639, 644
 toLower 634, 636
 top member function of
 priority_queue 513
 top member function of **stack** 509
 total 154
 toupper 634, 636
 trailing return type 557
 trailing return type (function) 175
 trailing return types 175
 trailing zeros 432
Transaction class (ATM case study)
 733, 734, 735, 738, 772
 transaction processing 505
 transaction-processing program 466
 transaction-processing system 456
 transfer of control 70
transform algorithm 529, 533, 558
 transition 71
 transition arrow 71, 74, 78, 79
 transition between states in the UML 703
 translation 7
 translation unit 667
 traversal 607
 trigonometric cosine 131
 trigonometric sine 132
 trigonometric tangent 132
 true 32
 true 73, 74, 75
 truncate 28, 83, 90, 448
 truncate fractional part of a **double** 138
 truth table 123
 ! (logical NOT) operator 125
 && (logical AND) operator 123
 || (logical OR) operator 124
 try block 226, 564, 567, 570, 571
 try block expires 565
 try statement 226
 Turing Machine 70
 two-dimensional array 211, 215
 two-dimensional array manipulations
 215
 two's complement 788
 notation 788
 twos position 783
 tying an output stream to an input stream
 443
 type checking 793, 794
 type field 473
 type information 473
 type name (enumerations) 149
 type of a variable 152
 type of the this pointer 292
 type parameter 173, 583, 589
 type-safe linkage 171
 type_info class 413
typedef 418, 592, 609, 618
 fstream 420
 ifstream 420
 in first-class containers 479

typedef (cont.)

- `istream` 418
- `istream` 418
- `ofstream` 420
- `ostream` 418
- `typeid` 413, 578
- `<typename>` header 140, 413
- typename keyword 173, 583
- type-safe I/O 425

U

- UML (Unified Modeling Language) 6, 41, 683, 684, 690, 694, 701, 702, 733
 - action expression 71
 - action state 71
 - activity diagram 70, 71, 78
 - arrow 71
 - attribute 41
 - class diagram 41
 - constructor in a class diagram 54
 - data types 44
 - decision symbol 74
 - diagram 690
 - diamond symbol 71, 74
 - dotted line 71
 - final state 71
 - guard condition 74
 - guillemets (« and ») 54
 - initial state 71
 - merge symbol 79
 - minus sign (–) 50
 - note 71
 - plus sign (+) 41
 - public operation 41
 - Resource Center
 - (www.deitel.com/UML/) 691
 - small circle symbol 71
 - solid circle symbol 71
 - `String` type 44
 - transition 71
 - transition arrow 71, 74, 78, 79
- UML activity diagram 105
 - solid circle (for representing an initial state) in the UML 705
 - solid circle enclosed in an open circle (for representing the end of an activity) in the UML 705
- UML class diagram
 - attribute compartment 701
 - constructor 54
 - operation compartment 708
- UML sequence diagram
 - activation 720
 - arrowhead 720
 - lifeline 719
- UML state diagram
 - rounded rectangle (for representing a state) in the UML 703
 - solid circle (for representing an initial state) in the UML 703
- UML use case diagram
 - actor 689
 - use case 689
- unary decrement operator (–) 96
- unary increment operator (++) 96
- unary minus (–) operator 90
- unary operator 90, 124, 231
- unary operator overload 309, 314

- unary plus (+) operator 90
- unary predicate function 497, 526, 529
- unary scope resolution operator (::) 169
- unbuffered output 420
- unbuffered standard error stream 418
- `#undef` preprocessing directive 794, 796
- undefined area in memory 618
- `underflow_error` exception 579
- underlying container 509
- underlying data structure 512
- underscore (–) 26
- unformatted I/O 417, 418, 425
- unformatted output 420, 422
- Unicode 592
- Unicode character set 417
- Unified Modeling Language (UML) 6, 683, 684, 690, 694, 701, 702, 733
- uniform initialization 94
- `uniform_int_distribution` 151
- unincremented copy of an object 321
- unique algorithm 538, 540, 558
- unique keys 500, 504, 507
- unique member function of `list` 498
- `unique_copy` algorithm 541, 542, 558
- `unique_ptr` class 575
 - built-in array 578
- universal-time format 260
- UNIX 449
- unnamed bit field 633
- unnamed bit field with a zero width 633
- unnamed namespace 667
- unordered associative containers 476, 477, 500
 - `<unordered_map>` header 140
 - `unordered_map` class template 477, 507
 - `unordered_multimap` class template 477, 505
 - `unordered_multiset` class template 477, 501
 - `<unordered_set>` header 140
 - `unordered_set` class template 477, 504
- unscoped `enum` 150
- `unsigned char` data type 139
- `unsigned data` type 139, 145
- `unsigned int` 91
- `unsigned int` data type 139, 145
- `unsigned int` fundamental type 82
- `unsigned integer` in bits 622
- `unsigned long` 178, 650
- `unsigned long data` type 138
- `unsigned long int` 177, 178
- `unsigned long int data` type 138
- `unsigned long long data` type 138
- `unsigned long long int` 178
- `unsigned long long int data` type 138
- `unsigned short data` type 139
- `unsigned short int data` type 139
- untie an input stream from an output stream 443
- unwinding the function call stack 568
- update records in place 456
- `upper_bound` algorithm 548, 558
- `upper_bound` function of associative container 503
- uppercase letter 26, 140, 634, 636
- uppercase stream manipulator 432, 436, 438
- use case diagram in the UML 689, 690
- use case in the UML 688, 689

- use case modeling 688
 - user-defined class name 39
 - user-defined function 132
 - user-defined type 40, 149, 337
 - using a dynamically allocated `ostream` object 610
 - using a function template 173
 - Using a `static` data member to maintain a count of the number of objects of a class 299
 - using an iterator to output a `string` 608
 - using arrays instead of `switch` 195
 - `using` declaration 34
 - in headers 56
 - `using` directive 34, 665
 - in headers 56
 - using function `swap` to swap two strings 598
 - using Standard Library functions to perform a heap sort 548
 - Using `virtual` base classes 680
 - `<utility>` header 141
 - utility function 265
- V**
- validation 64
 - validity checking 64
 - value 27
 - value initialize 238
 - value of a variable 152
 - value of an array element 187
 - `value_type` 479
 - variable 25
 - variable name
 - argument 44
 - parameter 44
 - variadic template 504
 - `<vector>` header 140
 - `vector` class 221
 - capacity function 488, 488
 - `cbegin` function 490
 - `crend` function 490
 - `push_back` function 488
 - `push_front` function 488
 - `rbegin` function 490
 - `rend` function 490
 - `vector` class template 186, 486
 - `push_back` member function 227
 - `shrink_to_fit` member function 490
 - `vector` class template element-manipulation functions 490
 - `<vector>` header 221
 - verb phrase in requirements specification 707
 - vertical spacing 102
 - vertical tab ('v') 634, 637
 - vi 6
 - `virtual` base class 661, 678, 679, 680, 681
 - `virtual` destructor 389
 - `virtual` function 375, 383, 407, 409, 678
 - call 409
 - call illustrated 408
 - table (*vtable*) 407
 - `virtual` inheritance 679
 - virtual memory 573, 574

visibility in the UML 726
 visibility marker in the UML 726
 Visual Studio 2012 Express Edition 7
void * 249, 655
void keyword 40, 48
void return type 138
volatile qualifier 661
 volume of a cube 164
vtable 407, 409, 410
vtable pointer 410

W

“walk off” either end of an array 323
 warning message 66
 waterfall model 688
wchar_t 592
wchar_t character type 418
 “weakest” iterator type 482, 519
what member function of an exception object 226
what virtual function of class **exception** 562, 567, 573

while repetition statement 72, 78, 102, 110
while statement activity diagram 79
 whitespace characters 21, 22, 35, 422, 423, 426, 634, 637, 791, 796
 whole/part relationship 695
 width implicitly set to 0 429
width member function of class **ios_base** 429
 width of a bit field 630
 width of random number range 146
 width setting 429
 Windows 15, 116
 Windows operating system 15
 Windows Phone 7 15
Withdrawal class (ATM case study) 693, 694, 695, 696, 699, 700, 705, 708, 715, 716, 719, 720, 728, 729, 731, 733, 734, 735, 737, 738, 739
 word 617
 word boundary 617
 workflow of a portion of a software system 71
 workflow of an object in the UML 704

Wozniak, Steve 16
 wraparound 320
write 457, 462
write function of **ostream** 421, 425
 writing data randomly to a random-access file 462

X

Xcode 7
 Xerox PARC (Palo Alto Research Center) 16
 XML (extensible markup language) 473
xor operator keyword 669
xor_eq operator keyword 669

Z

zeroth element 186
 zero-width bit field 633